






CoVeriTeam: On-Demand Composition of Cooperative Verification Systems

Dirk Beyer   and Sudeep Kanav 

LMU Munich, Munich, Germany

Abstract. There is no silver bullet for software verification: Different techniques have different strengths. Thus, it is imperative to combine the strengths of verification tools via combinations and cooperation. CoVeriTeam is a language and tool for on-demand composition of cooperative approaches. It provides a systematic and modular way to combine existing tools (without changing them) in order to leverage their full potential. The idea of cooperative verification is that different tools help each other to achieve the goal of correctly solving verification tasks. The language is based on verification artifacts (programs, specifications, witnesses) as basic objects and verification actors (verifiers, validators, testers) as basic operations. We define composition operators that make it possible to easily describe new compositions. Verification artifacts are the interface between the different verification actors. CoVeriTeam consists of a language for composition of verification actors, and its interpreter. As a result of viewing tools as components, we can now create powerful verification engines that are beyond the possibilities of single tools, avoiding to develop certain components repeatedly. We illustrate the abilities of CoVeriTeam on a few case studies. We expect that CoVeriTeam will help verification researchers and practitioners to easily experiment with new tools, and assist them in rapid prototyping of tool combinations.

Keywords: Cooperative Verification · Tool Development · Software Verification · Automatic Verification · Verification Tools · Tool Composition · Tool Reuse

1 Introduction

As research in the field of formal verification advanced, the complexity of the programs under verification also kept on increasing. As a result, despite its successful application to the source code of large industrial and open-source projects [2, 3, 23, 27, 36], the current techniques fall short on solving many important verification tasks. It seems essential to combine the strengths of different verification techniques and tools to solve these tasks.

The verification community successfully applies different approaches to combine ideas: integrated approaches (source-code-based), where different pieces of source code are integrated into one tool [28], and off-the-shelf approaches (executable-based), where different executables from existing tools are combined

without changing them. The latter can be further classified into sequential and parallel portfolio [33], algorithm selection [37], and cooperative approaches [22].

The integrated approaches require development effort for adaptation or implementation of integrated components instead of building on existing mature implementations—the combination is very tight. On the other hand, the standard off-the-shelf approaches (portfolio [33] and selection [37]) let the tools run in isolation and the individual tools do not cooperate at all. The components do not benefit from the knowledge that is produced by other tools in the combination—the combination is very loose. In this work, we focus on cooperative verification, which is neither as tight as source-code integration nor as loose as portfolio and selection approaches—somewhere in between the two extremes.

Cooperative verification [22] is an approach to combine different tools for verification in such a way that they help each other solving a verification task, where the combinations are neither too tight nor too loose. Implementations include using a shared data base to exchange information (e.g., there are cooperative SAT solvers that use a shared set of learned clauses [34], and cooperative software verifiers that use a shared set of reached abstract states [14]) or pass information from one tool to the other (e.g., conditional model checkers [13, 25]). Cooperative verification aims to combine the individual strengths of these technologies to achieve better results. Our thesis is that *programming* (meta) verification systems based on *combination* and *cooperation* could be a promising solution. CoVeriTeam provides a framework to achieve this.

Developing such a tool is not straightforward. Various concerns that need to be addressed for developing a robust solution can be broadly divided in two categories: *concepts* and *execution*. (1) Concepts deal with defining the interfaces for tools, and with the mechanism for their combination. Before tools can cooperate, we need a common definition of tools based on their behavior. We need to categorize *what a tool does*, *what inputs it consumes*, and *what outputs it produces*, before we can use it in a cooperative framework with ease. In CoVeriTeam, we categorize tools in various types of verification *actors*, and the inputs and outputs produced by these actors in verification *artifacts*. The actors can be combined with the help of composition operators that define the mechanism of cooperation. (2) Execution is concerned with all issues during the execution of a tool. Actors first need to execute to cooperate. This opens another dimension of challenges and opportunities to improve the cooperation. To give two examples: a tool might have a too high resource consumption, thus, resources must be controlled and limited, and tools might interfere with other executing processes, thus, tools must be executed in isolated containers.

This paper presents CoVeriTeam, a language and tool for on-demand composition of cooperative verification systems that solves the above mentioned challenges. We contribute a domain-specific language and an execution engine. In the CoVeriTeam language, we can compose new actors based on existing ones using built-in composition operators. The existing components are not changed, but taken *off-the-shelf* from actor providers (technically: tool archives). We do not change existing software components: the composition is done *on-demand*

(when needed by the user) and *on-the-fly* (it does not compile a new tool from the components). In other words, existing verification tools are viewed as off-the-shelf components, and can be used in a larger context to construct more powerful verification compositions. Our approach does not require writing code in programming languages used to develop the underlying components. In the CoVeriTeam language, the user can execute tools without fearing that they interact with the host system or with other tools in an unspecified way. The execution environment, as well as input and output, are controlled using the Linux features cgroups, name spaces, and overlay file systems. We use the BENCHEXEC [20] system as library for convenient access to those OS features via a Python API.

Contributions. We make the following contributions:

1. a language to compose new verification tools based on existing ones,
2. an execution engine for on-the-fly execution of these compositions,
3. case studies implementing combinations in CoVeriTeam that were previously achieved only via hard-wired combinations, and
4. an open-source implementation and an artifact for reproduction.

In addition to the above mentioned contributions, CoVeriTeam provides the following features to the end user: (1) CoVeriTeam takes care of downloading and installing specified verification tools on the host system. (2) There is no need to learn command-line parameters of a verification tool because CoVeriTeam takes care of translating the input to the arguments for the underlying tool. This provides a uniform interface to a number of similar tools. (3) The off-the-shelf components (i.e., tools) are executed in a container, with resource limits, such that the execution cannot change (or even damage) the host system.

These features in turn enable a researcher or practitioner to easily experiment with new tools, and rapidly prototype new verification combinations. CoVeriTeam liberates the researcher who uses tool combinations from maintaining scripts that combine tools executions, and worrying about downloading, installing, and figuring out the command to execute a verification tool.

Impact. CoVeriTeam has already found use cases in the verification community: (1) It was used in a modular implementation of CEGAR [26] using off-the-shelf components [12]. (2) It was used for construction and evaluation of various verifier combinations [17]. (3) CoVeriTeam (wrapped in a service) was used in the software-verification competition 2021 and 2022 to help the participants debug issues with their tools (see Sect. 3 in [7]). Also, according to SV-COMP rules, a team is granted points only for those tasks whose result can be validated using a validator. Thus, a verifier-validator combination might be interesting for participants. With the help of CoVeriTeam such combinations can be easily constructed.

Also, the advent of many high-quality verifiers should lead to a certain level of standardization of the API and provided features. For example, tools for SMT or SAT solving are easy to use because of their standardized input language (e.g., SMTLIB for SMT solvers [4]). Consequently, such tools can be easily integrated into larger architectures as components. Our vision is that soon verifiers will be seen also as components that can be used in larger architectures just like SMT solvers are now integrated into verification tools.

Example 1 Witness Validation

Input: Program p , Specification s **Output:** Verdict

```

1: verifier := Verifier("Ultimate Automizer")
2: validator := Validator("CPAchecker")
3: result := verifier.verify(p, s)
4: if result.verdict  $\in$  {TRUE, FALSE} then
5:   result = validator.validate (p, s, result.witness)
6: return (result.verdict, result.witness)

```

2 Running Example

We explain the idea of CoVeriTeam using a short example. Verifiers are complex software systems and might have bugs. Therefore, for more assurance a user might want to validate the result of a verifier based on the verification witness that the verifier produces [10]. Such a procedure is sketched in [Example 1](#).

The user wanting to execute the procedure sketched in [Example 1](#) would need to download the tools (verifier and validator), execute the verifier, check the result of the verifier, and then if needed connect the outputs of the verifier with the inputs of the validator. The user would quite possibly write a shell script to do this, which is cumbersome and difficult to maintain.

CoVeriTeam takes care of all the above issues. In the next section, we discuss the types, namely artifacts and actors, that are used in the CoVeriTeam language. After this, we explain the design and usage of the CoVeriTeam execution engine, and discuss the CoVeriTeam program for our validating verifier in [Listing 1](#).

3 Design and Implementation of CoVeriTeam

We now explain details about the design and implementation of CoVeriTeam. First we discuss conceptual notions of actors, artifacts, and compositions; then we discuss execution concerns that a cooperative verification tool needs to handle. Then we delve deeper into implementation details where we discuss how an actor is created and executed. Last, we briefly explain the API that CoVeriTeam exposes and extensibility of this API.

3.1 Concepts

This section describes the language that we have designed for cooperative verification and on-demand composition. At first we describe the notion of artifacts and actors, and then the composition language to compose components to new actors.

Artifacts and Actors. Verification artifacts provide the means of information (and knowledge) exchange between the verification actors (tools). [Figure 1](#) shows a hierarchy of artifacts, restricted to those that we have used in the case studies for evaluating our work. On a high level we divide verification artifacts in

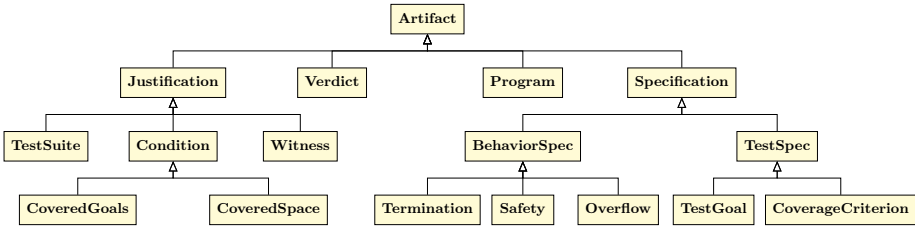


Fig. 1: Hierarchy of Artifacts (arrows indicate an is-a relation)

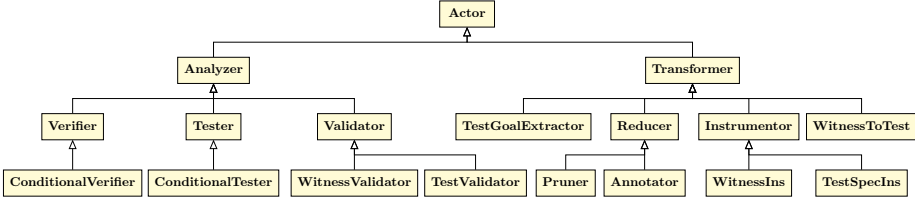


Fig. 2: Hierarchy of Actors (arrows indicate an is-a relation)

the following kinds: *Programs*, *Specifications*, *Verdicts*, and *Justifications*. *Programs* are behavior models (can be further classified into programs, control-flow graphs, timed automata, etc.). *Specifications* include behavioral specifications (for formal verification) and test specifications (coverage criteria for test-case generation). *Verdicts* are produced by actors signifying the class of result obtained (TRUE, FALSE, UNKNOWN, TIMEOUT, ERROR). *Justifications* for the verdict are produced by an actor; they include test suites to justify an obtained coverage, or verification witnesses to justify a verification result.

Verification actors act on the artifacts and as a result either produce new artifacts or transform a given artifact for consumption by some other actor. Figure 2 shows a hierarchy of actors, restricted to those that we have used in the case studies for evaluating our work. We divide verification actors in the following types: *Analyzers* and *Transformers*. *Analyzers* create new knowledge, e.g., verifiers, validators, and test generators. *Transformers* instrument, refine, or abstract artifacts.

Composition. Actors can be composed to create new actors. Our language supports the following compositions: *sequence*, *parallel*, *if-then-else*, and *repeat*.

COVERTTEAM infers types and type-checks the compositions, and then either constructs a new actor or throws a type error. In the following, we use the notation I_a for the input parameter set of an actor a and O_a for the output parameter set of a . A parameter is a pair of name and *artifact type*. A *name clash* between two sets A and B exists if there is a name in A that is mapped to a different artifact type in B , more formally: $\exists(a, t_1) \in A, (a, t_2) \in B : t_1 \neq t_2$. The *actor type* is a mapping from input parameter set to output parameter set ($I_a \rightarrow O_a$).

Sequential. Given two actors a_1 and a_2 , the sequential composition SEQUENCE(a_1, a_2) (Fig. 3a) constructs an actor that executes a_1 and a_2 in sequence, i.e., one after another. The composition is well-typed if there is no name clash between I_{a_1} and $(I_{a_2} \setminus O_{a_1})$. This means that we allow same artifact to be passed to the second actor in sequence, but disallow the confusing scenario

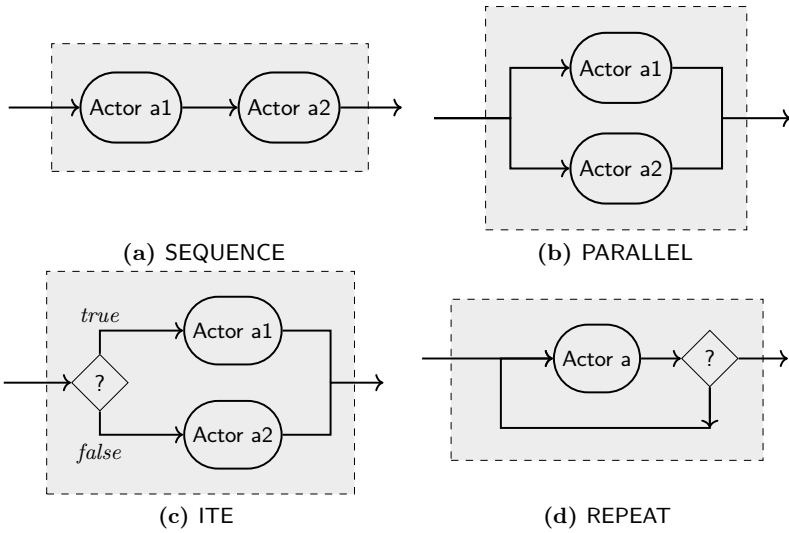


Fig. 3: Compositions in CoVeriTEm

where both actors expect an artifact with the same name but different type. The inferred type of the composition is $I_{a1} \cup (I_{a2} \setminus O_{a1}) \rightarrow O_{a2}$.

Parallel. Given two actors $a1$ and $a2$, the parallel composition PARALLEL ($a1, a2$) (Fig. 3b) constructs an actor that executes the actors $a1$ and $a2$ in parallel. The composition is well-typed if (a) there is no name clash between I_{a1} and I_{a2} and (b) the names of O_{a1} and O_{a2} are disjoint. The inferred type of the composition is $I_{a1} \cup I_{a2} \rightarrow O_{a1} \cup O_{a2}$.

ITE. Given a predicate $cond$ and two actors $a1$ and $a2$, the *if-then-else* composition ITE ($cond, a1, a2$) (Fig. 3c) constructs an actor that executes the actor $a1$ if predicate $cond$ evaluates to *true*, and the actor $a2$ otherwise. The composition is well-typed if (a) there is no name clash between $cond$, I_{a1} , and I_{a2} , and (b) the output parameters are the same ($O_{a1} = O_{a2}$). The inferred type of the composition is $I_{a1} \cup I_{a2} \cup vars(cond) \rightarrow O_{a1}$, where $vars$ maps the variables used in a predicate to their artifact types. This allows us to define the condition $cond$ using artifacts other than the inputs of I_{a1} and I_{a2} .

There are situations where $a2$ is not required and its explicit specification only increases complexity. So, we have relaxed the type checker and made $a2$ optional.

Repeat. Given a set fp and an actor a , the *repeat* composition REPEAT(fp, a) (Fig. 3d) constructs an actor that repeatedly executes actor a until a fixed-point of set fp is reached, that is, fp did not change in the last execution of a . The *repeat* composition feeds back the output of a from iteration n to a for iteration $n + 1$. Let us partition $I_a \cup O_a$ into three sets: $I_a \setminus O_a$, $O_a \setminus I_a$, and $I_a \cap O_a$. The parameters in $I_a \setminus O_a$ do not change their value and the parameters in $O_a \setminus I_a$ are accumulated if accumulatable, otherwise their value after the execution of the composition is the value from the last iteration. The composition is well-typed if $fp \subseteq dom(I_a \cap O_a)$, where dom returns the names of a parameter set. The inferred type of the composition is $I_a \rightarrow O_a$.

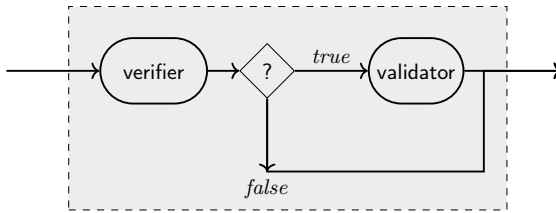


Fig. 4: CoVeriTeam implementation of the validating verifier from Example 1

Figure 4 shows the pictorial representation of our running example using these compositions. First a verifier is executed, then the validator is executed if the verifier returned TRUE or FALSE, otherwise (in case of UNKNOWN, TIMEOUT, ERROR) the validator is not executed and the output of the verifier is forwarded.

3.2 Execution Concerns

A tool for cooperative verification orchestrates the execution of verification tools. This means it needs to assemble the command for the tool, as well as handle the output produced by the tool. A verification tool might consume a lot of resources and a user might want to limit this. It might crash during execution, might interfere with other processes. CoVeriTeam needs to handle all these concerns.

Instead of developing our own infrastructure to handle these concerns, we reuse some of the features provided by BENCHEXEC [20]: we use tool-info modules to assemble command lines and parse log output, RUNEXEC (a component of BENCHEXEC) to execute tools in a container and limit resource consumption.

Tool-Info modules are integration modules of the benchmarking framework BENCHEXEC [20]. A typical tool-info module is a few lines of code used for assembling a command line and parsing the log output produced by the tool. It takes only a few hours to create one.¹ CoVeriTeam uses tool-info modules to pass artifacts to atomic actors (assemble command-line) and extract artifacts from the output produced by the atomic actor. Using tool-info modules gave us integration of more than 80 tools without effort, because such integration modules exist for most well-known verifiers, validators, and testers (as many researchers use BENCHEXEC and provide such integration modules for their tools).

CoVeriTeam uses RUNEXEC to isolate tool execution to prevent interference with the execution environment and enforce resource limits. We also report back to the user the resources consumed by the tool execution as measured by RUNEXEC.

3.3 CoVeriTeam

Figure 5 provides an abstract view of the system. CoVeriTeam takes as input a program written in the CoVeriTeam language and artifacts. At first, the code generator converts this input program to Python code. This transformed

¹ We claim this based on our experience with tool developers creating their tool-info modules, which is a prerequisite for participating in SV-COMP or Test-Comp.

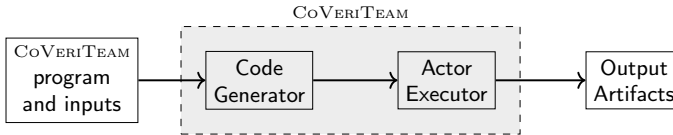


Fig. 5: Abstract view of the CoVeriTeam tool

```

1 verifier = ActorFactory.create(ProgramVerifier,
2     "actors/uautomizer.yml");
3
4 // Use validator if verdict is true or false
5 condition = ELEMENTOF(verdict, {TRUE, FALSE});
6 second_component = ITE(condition, validator);
7 // Verifier and second component to be executed in sequence
8 validating_verifier = SEQUENCE(verifier, second_component);
9
10 // Prepare example inputs
11 prog = ArtifactFactory.create(CProgram, prog_path);
12 spec = ArtifactFactory.create(BehaviorSpecification, spec_path);
13 inputs = {'program':prog, 'spec':spec};
14 // Execute the new component on the inputs
15 res = execute(validating_verifier, inputs);
16 print(res);
  
```

Listing 1: CoVeriTeam implementation of the validating verifier from Example 1

code uses the internal API of CoVeriTeam. Then this Python code is executed, which means the actor executor is called on the specified actor. This in turn produces output artifacts on successful execution of the actor.

There are four key parts of executing a CoVeriTeam program: creation of *atomic actors*, composition of *actors* (atomic or composite), creation of *artifacts*, and execution of the actors. We now give a brief explanation of these parts with the help of our running example. Listing 1 shows a CoVeriTeam implementation of the running example (Example 1).

Creation of an Atomic Actor. Atomic actors in CoVeriTeam provide an interface for external tools. CoVeriTeam uses the information provided in an actor-definition file to construct an atomic actor. Lines 1 and 2 in Listing 1 show the creation of atomic actors `verifier` and `validator` using the *ActorFactory* by providing the *ActorType* and the actor-definition file. Once constructed, this actor can be executed.

An *actor definition* is specified in a file in YAML format. It contains the information necessary for executing the actor: location from where to download the tool, the name of the tool-info module to assemble the command line and parse tool output, additional command-line parameters for the tool, resource limitations to enforce, etc. Listing 2 shows the actor definition file for UAutomizer [32]: the actor name is `uautomizer`, the identifier for the BENCHEXEC tool-info module is


```

1 actor_name: uautomizer
2 toolinfo_module: "ultimateautomizer.py"
3 archive:
4   doi: "10.5281/zenodo.3813788"
5   spdx_license_identifier: "LGPL-3.0-or-later"
6 options: ['--full-output', '--architecture', '32bit']
7 resource_limits:
8   memlimit: "15 GB"
9   timelimit: "15 min"
10  cpuCores: "8"
11 format_version: '1.1'

```

Listing 2: Definition of atomic actor in YAML format

`ultimateautomizer`, the DOI of the tool archive (or the URL for obtaining the tool archive), the SPDX license identifier, the options passed by CoVeriTeam to UAutomizer, and resource limits for the execution of the actor. Once an atomic actor has been constructed using an actor definition, CoVeriTeam has all the information necessary to execute the underlying tool with the provided artifacts.

Composition of an Actor. The second key part is the composition of an actor. Lines 6 and 8 in Listing 1 create composite actors using ITE and SEQUENCE, respectively. It is these compositions that create the validating verifier of our running example. Verification actors in CoVeriTeam can exchange information (artifacts) with other actors and cooperate through compositions.

Creation of an Artifact. The notion of artifact in CoVeriTeam is a *file* wrapped in an *artifact type*. The underlying files are the basis of an artifact—exchangeable information. Lines 11 and 12 in Listing 1 create artifacts using the *ArtifactFactory* by providing the *ArtifactType* and the artifact file. These artifacts would then be provided to the executor that then executes the actors on them.

Code Generation. The code generator of CoVeriTeam translates the input program to Python code that uses the internal API of CoVeriTeam. It is a canonical transformation in which the statements for creation of actors and artifacts are converted to Python statements instantiating corresponding classes from the API. Similarly, statements for composition and execution of actors are also transformed.

Execution. Analogously to the construction of actors, the execution of an actor in CoVeriTeam is also divided in two: atomic and composition. Line 15 in Listing 1 executes the actor `validating_verifier` on the given input artifacts.

Figure 6 shows the actor executor for both atomic and composite actors. It executes an actor on the provided artifacts. At first it type checks the inputs, i.e., check if the input types provided to actor comply with the expected input types of the actor. It then calls the executor for atomic or composite actor depending on the actor type. Thereafter, it type checks the outputs, and at last returns the artifacts.

Execution of an atomic actor means the execution of the underlying tool on the provided artifacts. At first, the executor downloads the tool if necessary. CoVeriTeam downloads and unzips the archive that contains the tool on the first execution of an atomic actor. It keeps the tool available in cache for later

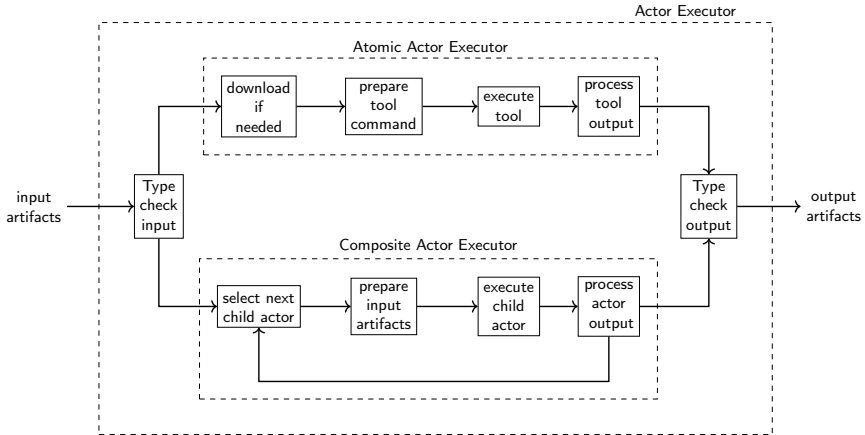


Fig. 6: Abstract view of an actor execution in CoVeriTeam

executions. After this step, the command line for the tool is prepared using the tool-info module. It then executes the tool in a container, and then processes the tool output, i.e., extracts the artifacts from the tool output and saves them.

Execution of a composition means executing the composed actors—making information produced by one available to other during the execution—as per the rules of composition. The composite-actor executor at first selects the next child actor to execute. It then computes the inputs for this selected actor. Then it executes this actor, which can be atomic or another composite actor, on these inputs. It then processes the outputs produced by the execution of the selected child actor. This processing could be temporarily saving, filtering, or modifying the produced artifacts. If needed, it then proceeds to execute the next child actor, otherwise exits the composition execution.

Output. CoVeriTeam collects all the artifacts produced during the execution of an actor, and saves them. The output can be divided into three parts: execution trace, artifacts, and log files. An execution trace is an XML file containing information about the artifacts consumed and produced by each actor, and also the resources consumed by atomic actors (as measured by `BENCHEXEC`) during the execution. CoVeriTeam also saves the artifacts produced during the execution of an actor. Additionally, for each atomic actor execution, it also saves a log file containing the command which was actually executed and the messages printed on `stdout`.

3.4 API

In addition to the above described features, CoVeriTeam exposes an API that is extensible. We expose actors, artifacts, utility actors, and compositions through Python packages. In this section, we briefly discuss this API.

Library of Actors and Compositions. CoVeriTeam provides a library of some actors and a few compositions that can be instantiated with suitable

actors. We considered actors based on the tools participating in the competitions on software verification and testing [5, 6] (available in the replication archives), because those are known to be mature and stable.

The library of compositions contains a validating verifier, an execution-based validator [11], a reducer-based construction of a conditional model checker [15], `CONDTEST` [18], and `METAVAL` [21]. These are present in the `examples/` directory of the `CoVeriTeam` repository. We discuss some of these constructions in [Sect. 4.1](#).

New Actors, Artifacts, and Tools. New actors, artifacts, and tools can be integrated easily in `CoVeriTeam`. The integration of a new atomic actor requires only creating a YAML actor definition and, if not already available, implementing a tool-info module. The integration of a new actor type in the language requires (1) creating a class for the actor specifying its input and output artifact types, (2) preparing the parameters to be passed to tool-info module, that in turn would create a command line for the tool execution, using the options from the YAML actor definition, and (3) creating output artifacts from the output files produced by the execution of an atomic actor of that type.

Integration of a new artifact requires creating a new class for the artifact. A basic artifact requires a path containing the artifact. Some artifacts support special features, for example, a test suite is a mergeable artifact (i.e., two test suites for a given input program can be merged into one test suite).

Integrating a new tool in the framework requires: (1) creating the tool-info module for it, (2) creating an actor definition for the tool, (3) providing a self-contained archive that can be executed on a Ubuntu machine.

At present, `CoVeriTeam` supports all verifiers and validators that are listed on the 2021 competition web sites of `SV-COMP`² and `Test-Comp`³. One needs only a few hours to create a new tool-info module and an actor-definition file. Within a couple of hours we were able to create the actor definitions for about 40 tools participating in `SV-COMP` and `Test-Comp`.

4 Evaluation

We now present our evaluation of `CoVeriTeam`. It consists of a few case studies, and insights from the experiments to measure performance overhead.

4.1 Case Studies

We evaluated `CoVeriTeam` on four more case studies, as indicated in the fourth column of [Table 1](#). We now explain two of these case studies using figures for compositions. The programs and explanations for all of the case studies are also available in our project repository (linked from the last column of [Table 1](#)).

Conditional Testing à la `CONDTEST`. Conditional testing [18] allows cooperation between different test generators (testers) by sharing the details of the

² <https://sv-comp.sosy-lab.org/2021/systems.php>

³ <https://test-comp.sosy-lab.org/2021/systems.php>

Table 1: Examples of cooperative techniques in the literature

Technique	Year	Reference	Case Study	More Info
Counterexample Checking [38]	2012	Sect. 5		
Conditional Model Checking [13]	2012	Sect. 5		
Precision Reuse [19]	2013	Sect. 5		
Witness Validation [8, 10]	2015, 2016	Figure 4	✓	Sect. 3.3
Execution-Based Validation [11]	2018	Sect. 5	✓	More info
Reducer [15]	2018	Sect. 5	✓	More info
CoVERiTEST [14]	2019	Sect. 5		
CONDTEST [18]	2019	Figures 7 and 8	✓	More info
METAVAL [21]	2020	Figure 9	✓	More info

already covered test goals. A conditional tester outputs a condition, in addition to the generated test suite, representing the work already done. Then this condition is passed as an input to another conditional tester, in addition to the program and test specification. This tester can then focus on only the uncovered goals.

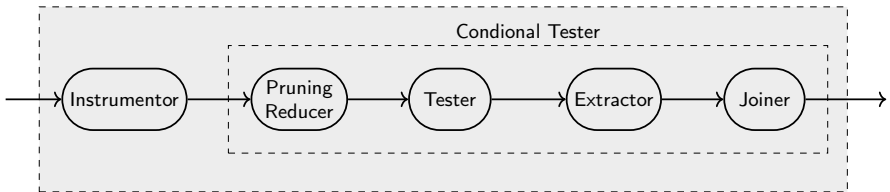


Fig. 7: Design of a conditional tester in CoVERiTEAM

Conditional testers can be constructed from off-the-shelf testers [18] with the help of three tools: a reducer, an extractor, and a joiner. A reducer used in conditional testing ($\text{Program} \times \text{Specification} \times \text{Condition} \rightarrow \text{Program}$) produces a residual program with the same behavior as the input program with respect to the remaining test goals. A set of test goals represents the condition. An extractor ($\text{Program} \times \text{Specification} \times \text{TestSuite} \rightarrow \text{Condition}$) extracts the condition—a set of test goals—covered by the provided test suite.

Figure 7 shows the composition of a conditional tester. First, the reducer produces the reduced program. The composition here uses a *pruning* reducer, which prunes the program according to the covered goals. Second, the tester generates the test cases. Third, the extractor extracts the goals covered in these test cases. Forth, the joiner merges the previously and newly covered goals. The reducer that we used expects the input program to be in a format containing certain labels for the purpose of tracking test goals. So, we put an instrumentor that *instruments* the test specification into the program, by adding these labels.

The conditional-testing concept can also be used iteratively to generate a test suite using a tester based on a verifier [18]. Such a composition uses a verifier as a backend and transforms a counterexample generated by the verifier to a test case.

Figure 8 shows the construction of a cyclic conditional tester. In this case, the *tester* itself is a composition of a verifier and a tool, *Witness2Test*, which generates test cases based on a witness produced by a verifier. This tester, in composition

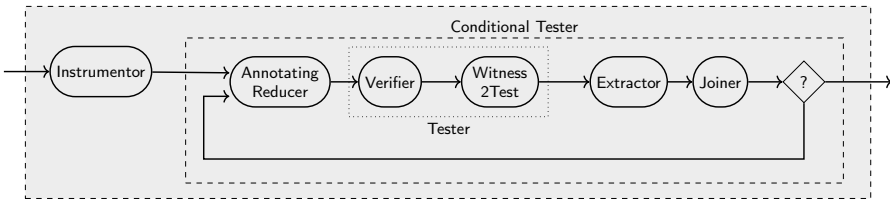


Fig. 8: Design of a cyclic conditional tester in CoVeriTeam

with a reducer, extractor, and a joiner is our conditional tester. This construction uses an *annotating* reducer, which (i) annotates the program with *error* labels for the verifier to find the path to and (ii) filters out the already covered goals, i.e., the condition, from the list of goals to be annotated. We put the conditional tester in the REPEAT composition to execute iteratively. The composition tracks the set ‘covered_goals’ to detect the fixed point to decide termination of the iteration. This composition will keep on accumulating the test suite generated in each iteration and finally output the union of all the generated test suites (see Sect. 3.1). As above, an instrumentor is placed before the conditional tester.

Verification-Based Validation à la METAVAL. METAVAL [21] uses off-the-shelf verifiers to perform validation tasks. A validator ($\text{Program} \times \text{Specification} \times \text{Verdict} \times \text{Witness} \rightarrow \text{Verdict} \times \text{Witness}$) validates the result produced by a verifier. METAVAL employs a three-stage process for validation. In the first stage, METAVAL instruments the input program with the input witness. The instrumented program—a product of the witness and the original program—is equivalent to the original program modulo the provided witness. This means that the instrumented program can be given to an off-the-shelf verifier for verification; and this verification functions as validation. In the second stage, METAVAL selects the verifier to use based on the specification. It chooses CPACHECKER for reachability, UAUTOMIZER for integer overflow and termination, and SYMBIOTIC for memory safety.⁴ In the third stage, the instrumented program is fed to a verifier along with the specification for verification. If the verification produces the expected result, then the result is confirmed and the witness valid, otherwise not.

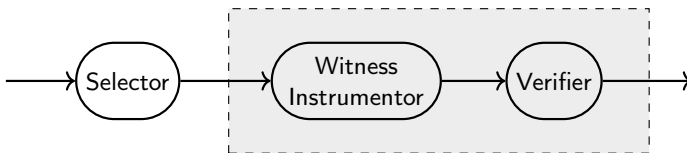


Fig. 9: Design of METAVAL in CoVeriTeam

Figure 9 shows the construction of METAVAL. First, the selector is executed that selects the backend verifier to execute. After this step, the program is

⁴ These were the best performing tools for a property according to SV-COMP results.

instrumented with the witness, and then the instrumented program is given to the selected verifier for checking the specification.

4.2 Performance

CoVeriTeam is a lightweight tool. Its container mode causes an overhead of around 0.8s for each actor execution in the composition, and the tool needs about 44MB memory. This means that if we run a tool 10 times in a sequence in a shell script unprotected and compare this to using the sequence composition in CoVeriTeam in protected container mode on the same input, the execution using CoVeriTeam will take 8s longer and requires 44MB more memory. In our experience, this overhead is not an issue for verification as, in general, the time taken for verification dominates the total execution time. For short-running, high-performance needs, the container mode can be switched off. We have conducted extensive experiments for performance evaluation of CoVeriTeam and point the reader to the [supplementary webpage](#) for this article for more details.

5 Related Work

We divide our literature overview into two parts: approaches for tool combinations, and cooperative verification approaches.

Approaches for Tool Combinations. *Evidential Tool Bus* (ETB) [29, 30, 39] is a distributed framework for integration of tools based on a variant of Data-log [1, 24]. It stores the established claims along with the corresponding files and their versions. This allows the reuse of partial results in regression verification. ETB orchestrates tool interaction through scripts, queries, and claims.

Our work seems close to ETB on a quick glance, but on a closer look there are profound differences. Conceptually, ETB is a query engine that uses claims, facts, and rules to define and execute a workflow. Whereas, CoVeriTeam has been designed to create and execute actors based on tools and their compositions. We give some semantic meaning, arguably simplistic, to the tools using (i) wrapper types of artifacts for the files produced and consumed by a tool and (ii) the notion of *verification actors* that allows us to see a tool as a *function*. This allows us to type-check tool compositions and allow only well-defined compositions. On the implementation side, we support more tools. This task was simplified by our design choice to use the integration mechanisms provided by BENCHEXEC (as used in SV-COMP and Test-Comp). Most well known automated verification tools already have been integrated in CoVeriTeam.

Electronic Tools Integration platform (ETI) [40] was envisioned as a “one stop shop” for the experimentation and evaluation of tools from the formal-methods community. It was intended to serve as a tool presentation, tool evaluation, and benchmarking site. The idea was to allow users to access tools through the internet without the need to install them. An ETI user is expected to provide an LTL based specification, based on which an execution scheme is synthesized.

The key focus of ETI and its incarnations has been remote tool execution, and their integration over internet. The tools are viewed agnostic to their function. We, in contrast, (i) have tackled local execution concerns and (ii) see a tool in its function as an actor that consumes and produces certain kinds of artifacts. The semantic meaning of a tool is given by this role.

Cooperative Verification Approaches. Our work targets developing a framework to express and execute cooperative verification approaches. In this section we describe some of these approaches from literature. We have implemented some of these combinations in COVERITEAM, some of which are described in Sect. 4.

A reduction of the input program using the counterexample produced by a verifier was discussed [38], where the key idea is to use the counterexample to provide the variable assignments to the program.

Conditional model checking (CMC) [13] outputs a condition—a summary of the knowledge gained—if the model checker fails to produce a verdict. The condition allows another model checker to save the effort of looking into already explored state space. Reducers [15] can turn any off-the-shelf model checker into a conditional model checker. Reducers take a source program and a condition and produce a *residual program* whose paths cover the unverified state space (negation of the condition). Conditional testing [18] applies the principle of conditional model checking to testing. A conditional tester outputs, in addition to the generated test cases, the goals for which test cases have been generated.

The idea of reusing the knowledge about already done work to reduce the workload of another tool was also applied to combine program analysis and testing [25, 31, 35]. One of these approaches [31] is based on conditional model checking [13]. In this case, the condition is used to construct a residual program, which is then fed to a test-case generator. Another approach [25] instruments the program with assumptions and assertions describing the already completed verification work. Then a testing tool is used to test the assumptions. Program partitioning [35] first performs the testing and then removes the satisfactorily tested paths and verifies the rest. COVERITEST [14], cooperative verifier-based testing, is a tester based on cooperation between different verification-based test-generation techniques. COVERITEST uses conditional model checkers [13] as verifier backends.

Precision reuse [19] is based on the use of abstraction precisions. The precision of an abstract domain is a good candidate for cooperation because it is small in size, and represents important information, i.e., the level of abstraction at which the analysis works. A model checker in addition to producing a verdict also produces a file containing information specifying precision, e.g., predicates.

Model checkers can also produce a witness, in addition to the verdict, as a justification of the verdict. These witnesses could be counterexamples for violations of a safety property, invariants as a proof of a safety property, a lasso for non-termination, a ranking function for termination, etc. These witnesses can be used later to help validate the result produced by a verifier [8, 9, 10].

Execution-based result validation [11] uses violation witnesses to generate test cases. A violation witness of a safety specification is refined to a test case. The test case is then used to validate the result of the verification.

6 Conclusion

Due to the free availability of many excellent verifiers, the time is ripe to view verification tools as components. It is necessary to have standardized interfaces, in order to define the inputs and outputs of verification components. We have identified a set of verification artifacts and verification actors, and a programming language for on-demand construction of new, combined verification systems.

So far, the architectural hierarchy ends mostly at the verifiers: verifiers are based on SMT solvers, which are based on SAT solvers, which are based on data-structure libraries. CoVeriTeam wants to change this and use verification artifacts as first-class objects in specifying new verifiers. We show on a few selected examples how easy it is to construct some verification systems that were so far hard-coded using glue code and wrapper scripts. We hope that many researchers and practitioners in the verification community find it interesting and stimulating to experiment on a high level with verification technology.

Future Work. The approach of CoVeriTeam opens up a whole new area of possibilities that yet needs to be explored. We have identified three key areas for the further work: (i) remote execution of tools, (ii) policy specification and enforcement, and (iii) more compositions and combinations. CoVeriTeam provides an interface for a verification tool based on its behavior. A web service wrapped around CoVeriTeam can be used to delegate execution of an actor, hence verification work, to the host of the service. The client for such a service can be transparently integrated in CoVeriTeam. In fact, we already provide client integration for a restricted and experimental version of such a service. Also, a user executing a combination of tools might want to have some restrictions on which tools should be allowed to execute. For example, a user might want to execute only those tools that comply with a certain license, or only those tools that are downloaded from a trusted source. A cooperative verification tool should support the specification and enforcement of such user *policies*. Further, we plan to support more compositions for cooperative verification in CoVeriTeam as we come across them. Recently, we were working on a *parallel-portfolio* composition [17].

Declarations

Data Availability Statement. CoVeriTeam is publicly available under the Apache 2 license.⁵ The data from our performance evaluation is available at the supplementary webpage of the paper.⁶ A replication package including all evaluated external tools is available at Zenodo [16].

Funding Statement. This work was funded in part by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

Acknowledgement. We thank Thomas Lemberger and Philipp Wendler for their valuable feedback on this article, and the SV-COMP community for their valuable feedback on experimenting with CoVeriTeam.

⁵ <https://gitlab.com/sosy-lab/software/coveriteam/>

⁶ <https://www.sosy-lab.org/research/coveriteam/>

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011). <https://doi.org/10.1145/1965724.1965743>
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., University of Iowa (2015), available at www.smt-lib.org
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
6. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Proc. FASE. pp. 505–519. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_25
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). <https://doi.org/10.1145/2950290.2950351>
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022)
10. 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>
11. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
12. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. ACM (2022)
13. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). <https://doi.org/10.1145/2393596.2393664>
14. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
15. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). <https://doi.org/10.1145/3180155.3180259>
16. Beyer, D., Kanav, S.: Reproduction package for article ‘CoVeriTeam: On-demand composition of cooperative verification systems’. Zenodo (2021). <https://doi.org/10.5281/zenodo.5644953>
17. Beyer, D., Kanav, S., Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: Proc. FASE. Springer (2022)
18. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11

19. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). <https://doi.org/10.1145/2491411.2491429>
20. 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>
21. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
22. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
23. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
24. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Eng.* **1**(1), 146–166 (1989). <https://doi.org/10.1109/69.43410>
25. Christakis, M., Müller, P., Wüstholtz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
26. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15
27. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
28. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL. pp. 269–282. ACM (1979). <https://doi.org/10.1145/567752.567778>
29. 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
30. Cruanes, S., Heymans, S., Mason, I., Owre, S., Shankar, N.: The semantics of Datalog for the Evidential Tool Bus. In: Specification, Algebra, and Software. pp. 256–275. Springer (2014). https://doi.org/10.1007/978-3-642-54624-2_13
31. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
32. 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
33. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. *Science* **275**(7), 51–54 (1997). <https://doi.org/10.1126/science.275.5296.51>
34. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. *Discrete Applied Mathematics* **154**(16), 2291–2306 (2006). <https://doi.org/10.1016/j.dam.2006.04.015>
35. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA. pp. 11–16. ACM (2006). <https://doi.org/10.1145/1138912.1138916>

36. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
37. Rice, J.R.: The algorithm selection problem. *Advances in Computers* **15**, 65–118 (1976). [https://doi.org/10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3)
38. Rocha, H.O., Barreto, R.S., Cordeiro, L.C., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Proc. IFM. pp. 128–142. LNCS 7321, Springer (2012). https://doi.org/10.1007/978-3-642-30729-4_10
39. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM. pp. 36–36. LNCS 3785, Springer (2005). https://doi.org/10.1007/11576280_3
40. 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>

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.

