

CoVeriTeam

Cooperative Verification
via Off-the-Shelf-Components

Dirk Beyer and Sudeep Kanav

LMU Munich, Germany



Motivation for CoVeriTeam

Problem: in software verification

- ▶ there is no silver bullet
- ▶ a lot of tools specializing in specific problem domains

Solution: cooperative verification

- ▶ employ multiple tools to solve a verification task
- ▶ which cooperate with each other through verification artifacts like witnesses

- ▶ defines a **language** for composing verification tools
- ▶ provides an **execution engine** to execute a composition
- ▶ controls the **resources** consumed by each tool during the execution
- ▶ produces a trace of **artifacts** generated throughout the execution
- ▶ contains a **library** of (some) examples of verification composition techniques from the literature
- ▶ **open-source** implementation in Python

Running Example

Algorithm 1 Witness Validation [2, 1]

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)
-

Cooperative Verification Language

- ▶ Artifact: means of information (and knowledge) exchange between the verification actors (tools)
- ▶ Actor: consumes, acts on, and produces artifacts
- ▶ Composition: of actors creating new actors

Artifacts

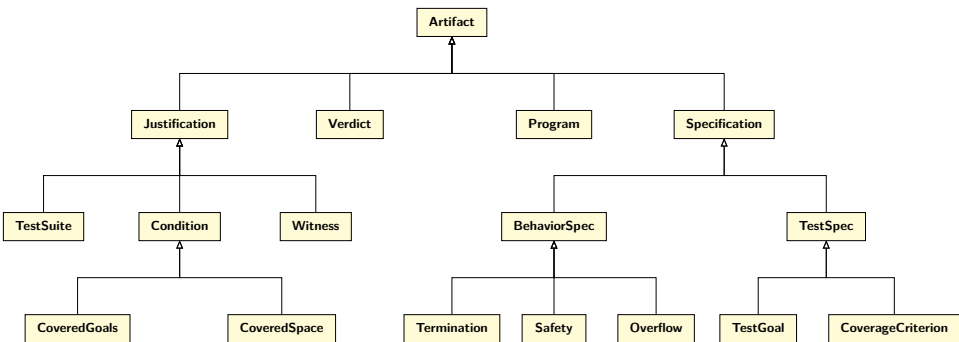


Figure: Hierachy of Artifacts (arrows indicate an is-a relation) [7]

Actors

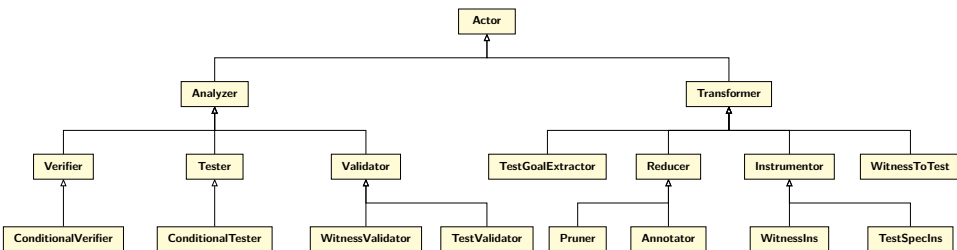


Figure: Hierachy of Actors (arrows indicate an is-a relation) [7]

Composition

- ▶ Sequential: executes actors in sequence
- ▶ Parallel: executes actors in parallel, waits for all to finish
- ▶ ITE: executes one or the other based on a condition.
- ▶ Repeat: repeats execution of an actor till a termination condition is satisfied

CoVeriTeam- Implementation and Usage

- ▶ Demo: key points
- ▶ Usage: config, input, output, command

Implementation Details

- ▶ Just-in Time Compilation
- ▶ Actor instantiation
- ▶ Use of BenchExec:
 - ▶ Tool-Info Modules
 - ▶ Containerized Execution
 - ▶ Resource Measurement

Usage and Features

```
8 actor_name: uautomizer
9 toolinfo_module: "ultimateautomizer.py"
10 archive:
11   location: "https://zenodo.org/record/3813788/files/uautomizer.zip"
12   doi: "10.5281/zenodo.3813788"
13   sdx_license_identifier: "LGPL-3.0-or-later"
14 options: ['--full-output', '--architecture', '32bit']
15 resourcelimits:
16   memlimit: "15 GB"
17   timelimit: "15 min"
18   cpuCores: "8"
19 format_version: '1.1'
```

Figure: YAML configuration for an atomic actor

Revisit the Running Example

Algorithm 2 Witness Validation [2, 1]

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)
-

Usage and Features

```
1 // Create verifier and validator from yml files
2 ver = ActorFactory.create(Verifier, "config/uautomizer.yml");
3 val = ActorFactory.create(Validator, "config/cpa-val.yml");
4
5 // Use validator if verdict is true or false,
6 // otherwise use an identity mapping of inputs to outputs
7 condition = ELEMENTOF(verdict, {TRUE, FALSE});
8 second_component = ITE(condition, val, Identity(val));
9
10 // Verifier and second component to be executed in sequence
11 validating_verifier = SEQUENCE(ver, second_component);
12
13 // Print type information about the composition (for illustration)
14 print(validating_verifier);
15
16 // Prepare example inputs
17 prog = ArtifactFactory.create(CProgram, prog_path);
18 spec = ArtifactFactory.create(ReachSafety, spec_path);
19 inputs = {'program':prog, 'spec':spec};
20
21 // Execute the new component on the inputs
22 execute(validating_verifier, inputs);
```

Figure: Running Example in COVERTEAM Language

Usage and Features

Command to execute the example:

```
bin/coveriteam examples/validated-verifier.cvt \  
--input prog_path=c/Problem02_label16.c \  
--input spec_path=properties/unreach-call.prp
```

Usage and Features

```
5 <actor name="Sequence">
6   <inputs>
7     <input name="program" type="CProgram">.../c/Problem02_label16.c</input>
8     <input name="spec" type="Specification">.../properties/unreach-call.prp</input>
9   </inputs>
10  <outputs>
11    <output name="verdict" type="Verdict">>false</output>
12    <output name="witness" type="Witness">.../witness.graphml</output>
13  </outputs>
14  <first>
15    <actor name="uautomizer">
16      <inputs>
17        <input name="program" type="CProgram">.../c/Problem02_label16.c</input>
18        <input name="spec" type="Specification">.../properties/unreach-call.prp</input>
19      </inputs>
20      <outputs>
21        <output name="verdict" type="Verdict">>false</output>
22        <output name="witness" type="Witness">.../witness.graphml</output>
23      </outputs>
24      <measurements cputime="25.170865524" memory="659492864" walltime="7.451572395977564"/>
25    </actor>
26  </first>
27  <second>
28    <actor name="ITE">
```

Figure: Snippet of the output xml produced by COVERTTEAM

Evaluation Through Case Studies

Table: Examples of cooperative techniques in literature

Technique	Year
Counterexample Checking [13]	2012
Conditional Model Checking [4]	2012
Slicing and Symbolic Execution [14]	2013
Precision Reuse [9]	2013
Algorithm Selection [15, 11]	2014, 2017
Witness Validation [2, 1]	2015, 2016
Execution-Based Validation [3]	2018
Reducer [6]	2018
Executable Counterexamples [12]	2018
COVERITEST [5]	2019
CONDTEST [8]	2019
METAVAL [10]	2020

Conclusion: CoVeriTeam

- ▶ defines a language for composing verification tools
- ▶ provides an execution engine to execute a composition
- ▶ predefined actors
- ▶ open-source implementation in Python

References I

- [1] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). doi:[10.1145/2950290.2950351](https://doi.org/10.1145/2950290.2950351)
- [2] 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). doi:[10.1145/2786805.2786867](https://doi.org/10.1145/2786805.2786867)
- [3] 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). doi:[10.1007/978-3-319-92994-1_1](https://doi.org/10.1007/978-3-319-92994-1_1)
- [4] 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). doi:[10.1145/2393596.2393664](https://doi.org/10.1145/2393596.2393664)

References II

- [5] Beyer, D., Jakobs, M.C.: COVERITEST: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). doi:10.1007/978-3-030-16722-6_23
- [6] Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). doi:10.1145/3180155.3180259
- [7] Beyer, D., Kanav, S.: COVERITEAM: On-demand composition of cooperative verification systems (2021)
- [8] Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). doi:10.1007/978-3-030-31784-3_11
- [9] Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). doi:10.1145/2491411.2491429

References III

- [10] Beyer, D., Spiessl, M.: *METAVAL: Witness validation via verification*. In: *Proc. CAV*. pp. 165–177. LNCS 12225, Springer (2020). doi:10.1007/978-3-030-53291-8_10
- [11] Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: *Predicting rankings of software verification tools*. In: *Proc. SWAN*. pp. 23–26. ACM (2017). doi:10.1145/3121257.3121262
- [12] Gennari, J., Gurfinkel, A., Kahsay, T., Navas, J.A., Schwartz, E.J.: *Executable counterexamples in software model checking*. In: *Proc. VSTTE*. pp. 17–37. LNCS 11294, Springer (2018). doi:10.1007/978-3-030-03592-1_2
- [13] 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). doi:10.1007/978-3-642-30729-4_10

References IV

- [14] Slaby, J., Strejček, J., Trtík, M.: Symbiotic: Synergy of instrumentation, slicing, and symbolic execution (competition contribution). In: Proc. TACAS. pp. 630–632. LNCS 7795, Springer (2013)
- [15] Tulsian, V., Kanade, A., Kumar, R., Lal, A., Nori, A.V.: MUX: Algorithm selection for software model checkers. In: Proc. MSR. ACM (2014). doi:[10.1145/2597073.2597080](https://doi.org/10.1145/2597073.2597080)