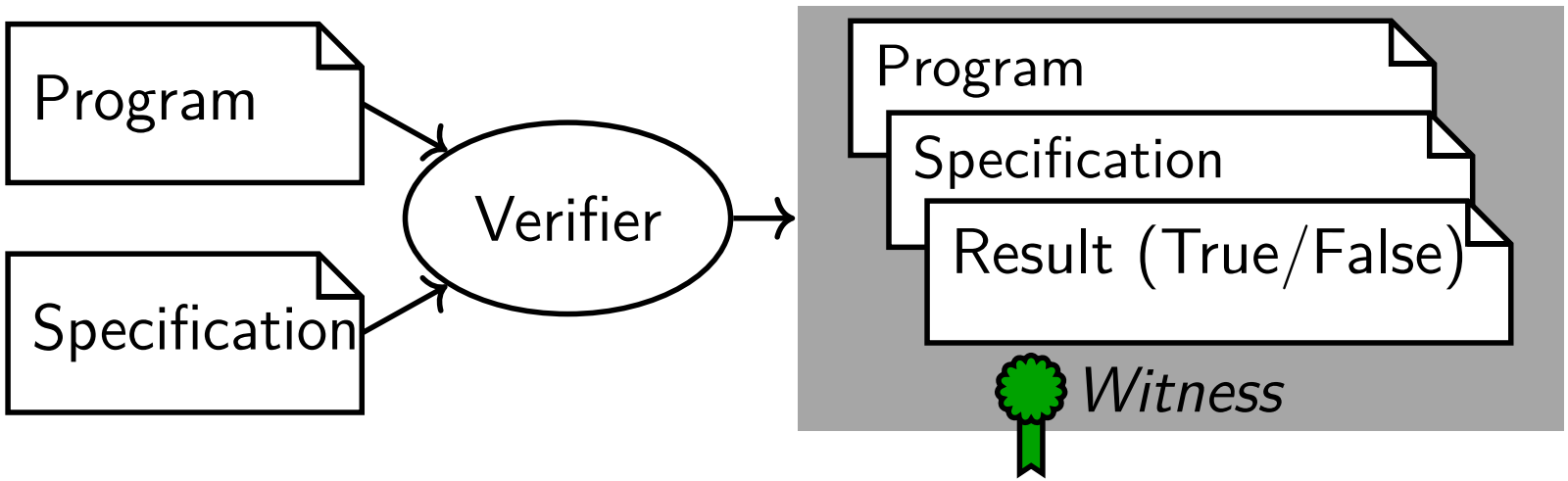
Verification Artifacts in Cooperative Verification

Survey and Unifying Component Framework

Dirk Beyer and Heike Wehrheim

LMU Munich and University of Oldenburg, Germany

Automatic Software Verification



Competitions in Software Verification and Testing

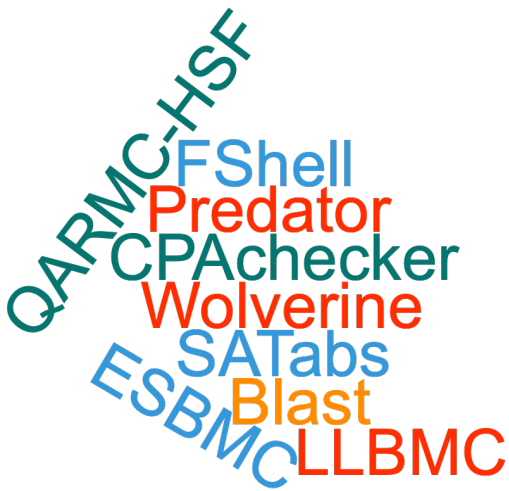
Many verification tools exist, and there are tool competitions:

- RERS: off-site, tools, free-style [16]
- SV-COMP: off-site, automatic tools, controlled [1]
- Test-Comp: off-site, automatic tools, controlled [3]
- VerifyThis: on-site, interactive, teams [17]

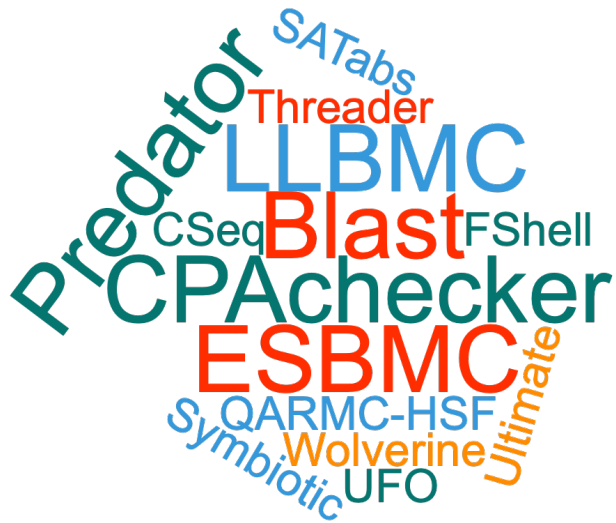
(alphabetic order)

SV-COMP (Automatic Tools 2012)

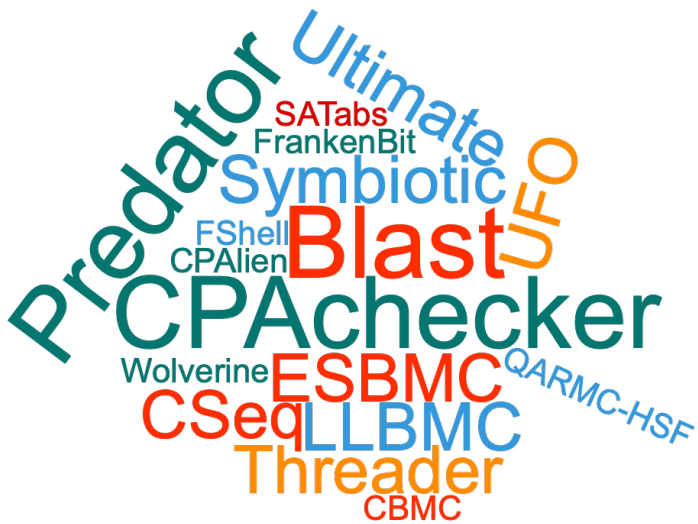
There is an increasing number of verification tools:



SV-COMP (Automatic Tools 2013, cumulative)

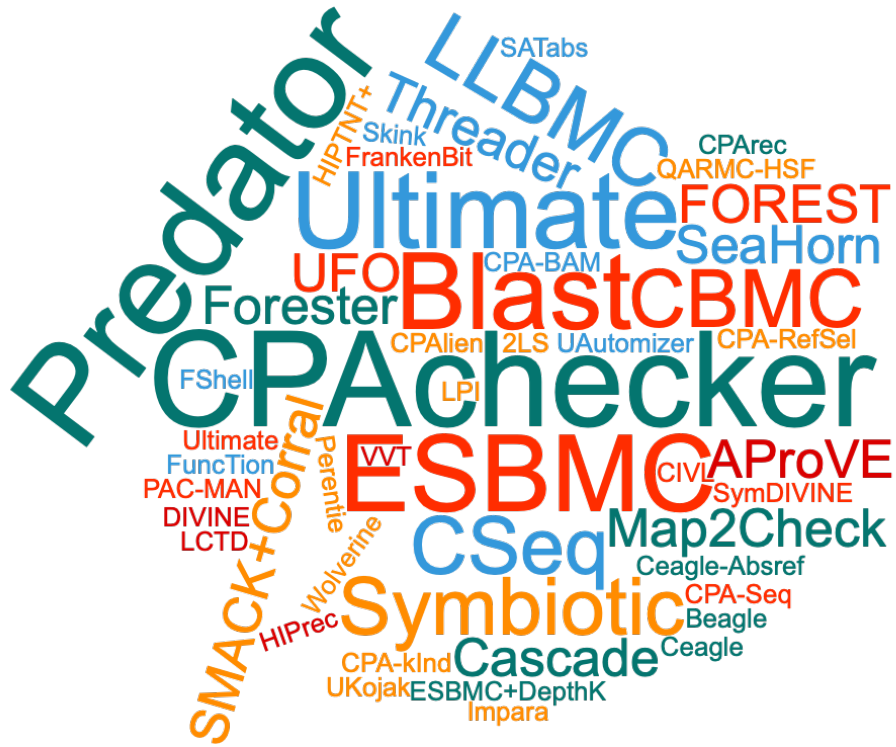


SV-COMP (Automatic Tools 2014, cumulative)

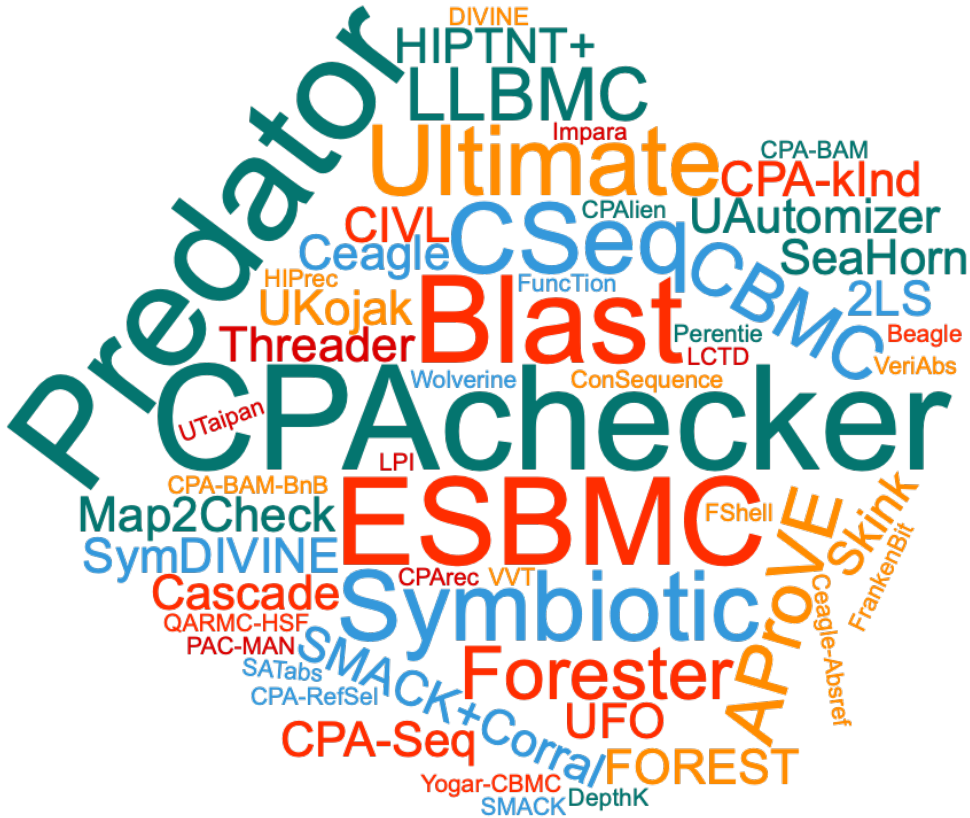


SV-COMP (Automatic Tools 2015, cumulative)

SV-COMP (Automatic Tools 2016, cumulative)

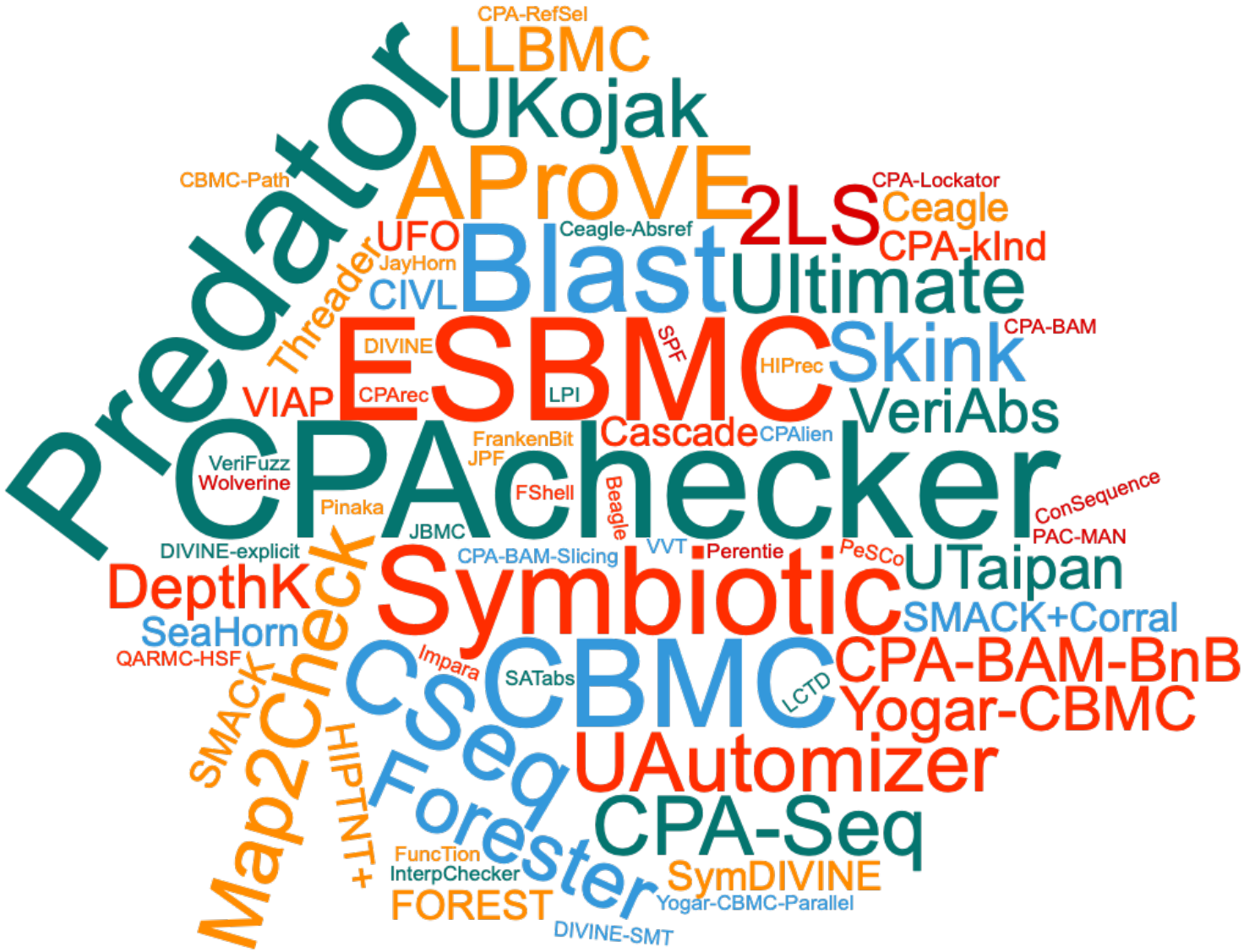


SV-COMP (Automatic Tools 2017, cumulative)



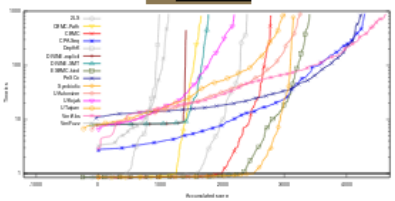
SV-COMP (Automatic Tools 2018, cumulative)

SV-COMP (Automatic Tools 2019, cumulative)

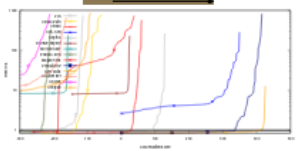


SV-COMP (Automatic Tools)

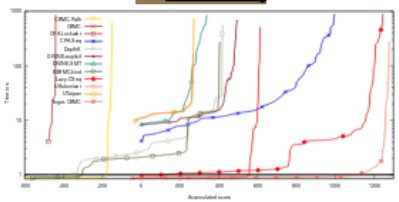
ReachSafety
1. VeriAbs
2. CPA-Seq
3. PeSCo



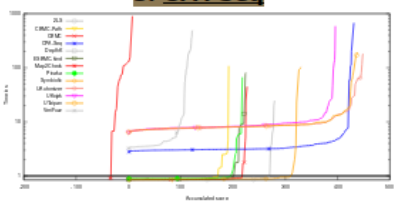
MemSafety
1. Symbiotic
2. PredatorHP
3. CPA-Seq



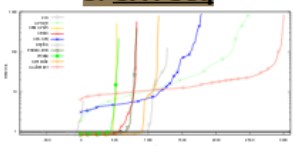
ConcurrencySafety
1. Yogar-CBMC
2. Lazy-CSeq
3. CPA-Seq



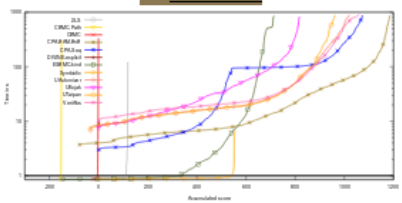
NoOverflows
1. UAutomizer
2. UTaipan
3. CPA-Seq



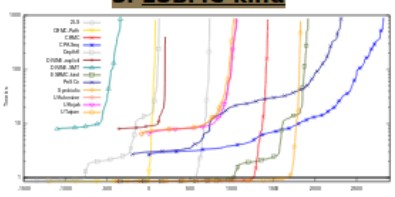
Termination
1. UAutomizer
2. AProVE
3. CPA-Seq



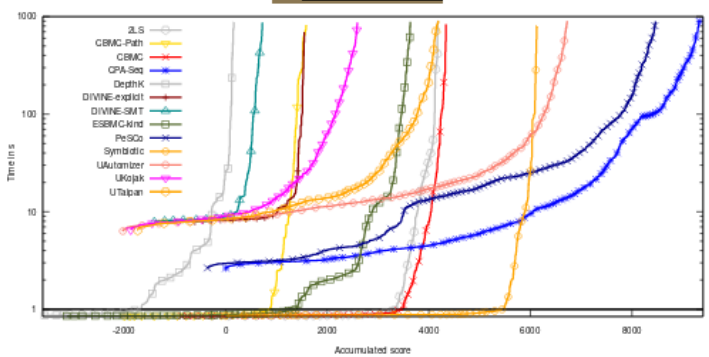
SoftwareSystems
1. CPA-BAM-BnB
2. CPA-Seq
3. VeriAbs



FalsificationOverall
1. CPA-Seq
2. PeSCo
3. ESBMC-kind



Overall
1. CPA-Seq
2. PeSCo
3. UAutomizer



https://sv-comp.sosy-lab.org/2019/results

Which techniques are used?

Participant	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✓	✓			✓			✓						✓	
APROVE			✓				✓	✓		✓	✓						✓	
CBMC				✓							✓					✓		
CBMC-PATH				✓							✓					✓		
CPA-BAM-BNB	✓	✓					✓				✓	✓	✓	✓				
CPA-LOCKATOR	✓	✓					✓				✓	✓	✓	✓		✓		
CPA-SEQ	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
DEPTHK				✓	✓						✓					✓		
DIVINE-EXPLICIT							✓				✓					✓		
DIVINE-SMT							✓				✓					✓		
ESBMC-KIND				✓	✓						✓					✓		
JAYHORN	✓	✓				✓		✓					✓	✓				
JBMC				✓							✓					✓		
JPF				✓			✓	✓			✓					✓		
LAZY-CSEQ				✓							✓					✓		
MAP2CHECK				✓							✓							
PESCO	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
PINAKA			✓	✓							✓							
PREDATORHP									✓									
SKINK	✓						✓							✓	✓			
SMACK	✓			✓		✓					✓		✓			✓		
SPF			✓						✓							✓		
SYMBIOTIC			✓					✓			✓							
UAUTOMIZER	✓	✓									✓		✓	✓	✓		✓	
UKOJAK	✓	✓									✓		✓	✓				
UTAIPAN	✓	✓									✓		✓	✓	✓			

Competition Report [2]

https://doi.org/10.1007/978-3-030-17502-3_9

Various Algorithms

17	Bounded Model Checking
13	CEGAR
8	Predicate Abstraction
5	k-Induction
4	Symbolic Execution
3	Automata-Based Analysis
2	Property-Directed Reachability (IC3)

Various Abstract Domains

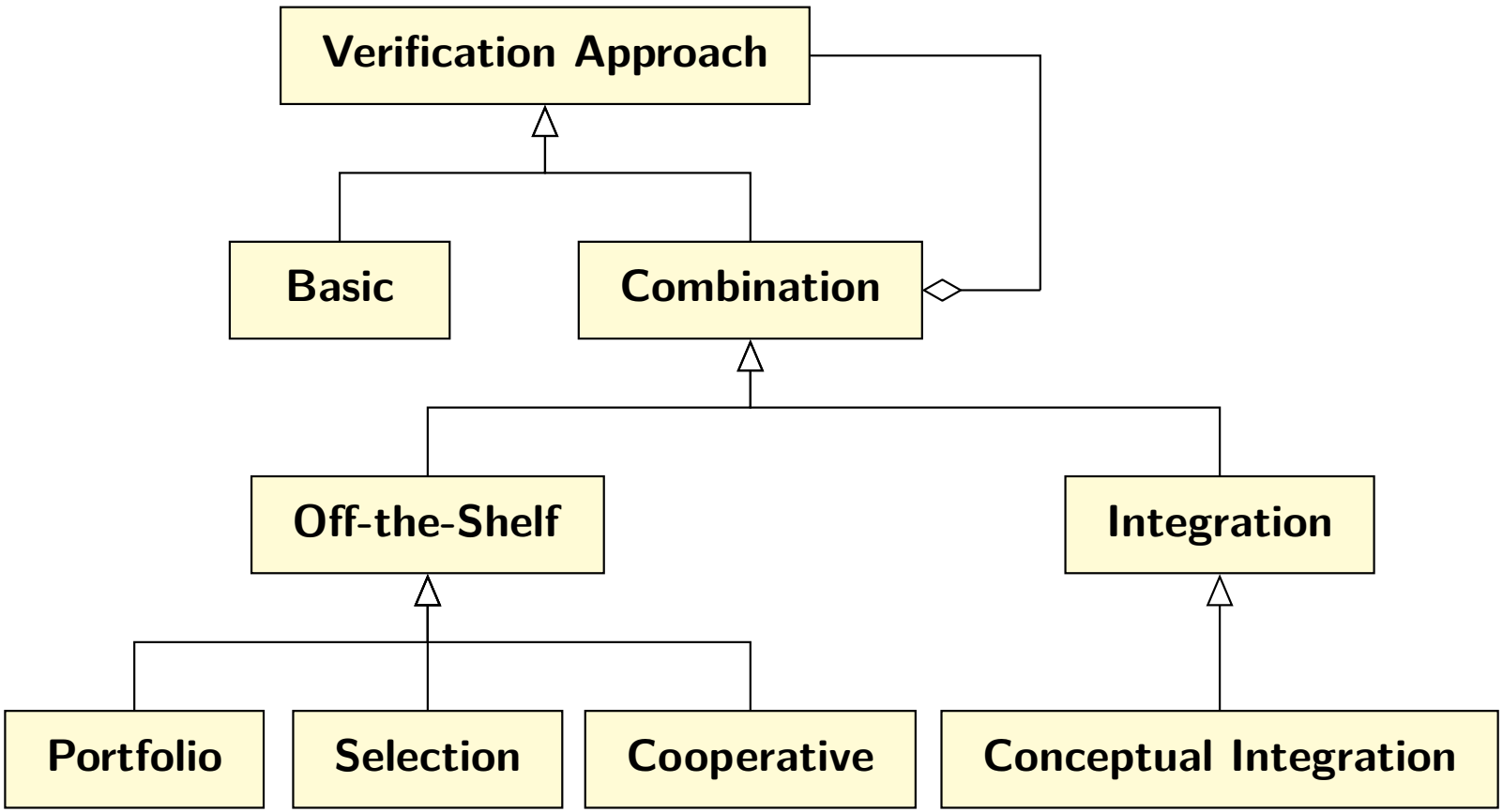
24	Bit-Precise Analysis
10	Explicit-Value Analysis
9	Numerical Interval Analysis
4	Shape Analysis
1	Separation Logic

So what is the conclusion?

- Many different kinds of programs seem to require many different good tools with different strengths.
- We do not necessarily need **more** verification tools.
- But we need **stronger** verification tools.
- And we need **combined** verification tools.
  ($\rightarrow$ this presentation)

Cooperative Verification

Approaches for Combinations



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

CoVeriTeam

- 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 [5, 4]

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 ∈ {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

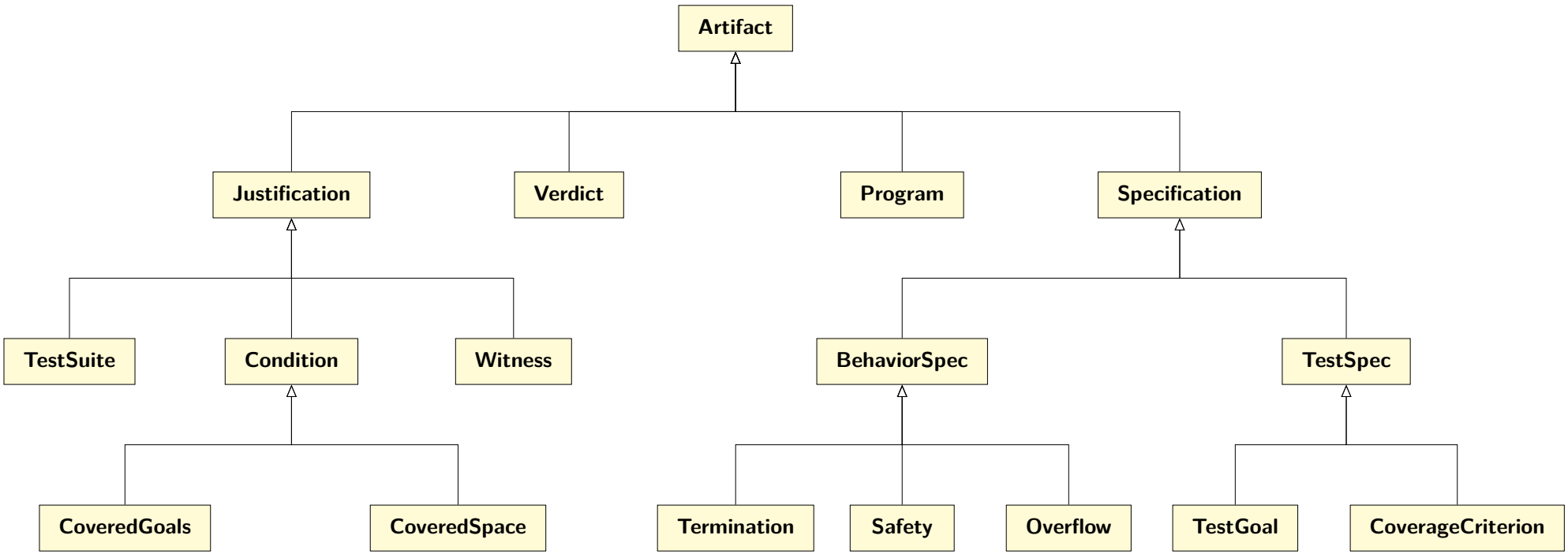


Figure: Hierachy of Artifacts (arrows indicate an is-a relation) [10]

Actors

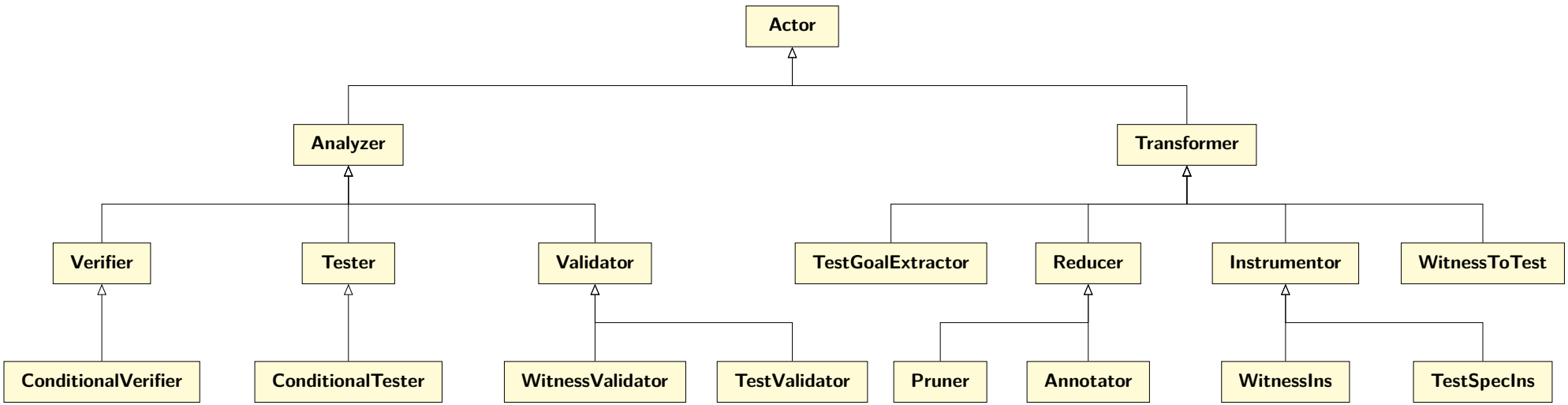


Figure: Hierachy of Actors (arrows indicate an is-a relation) [10]

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

```yaml
actor_name: uautomizer
toolinfo_module: "ultimateautomizer.py"
archive:
  location: "https://zenodo.org/record/3813788/files/uautomizer.zip"
  doi: "10.5281/zenodo.3813788"
  spdx_license_identifier: "LGPL-3.0-or-later"
options: ['--full-output', '--architecture', '32bit']
resourcelimits:
  memlimit: "15 GB"
  timelimit: "15 min"
  cpuCores: "8"
format_version: '1.1'
```

Figure: YAML configuration for an atomic actor

Revisit the Running Example

Algorithm 2 Witness Validation [5, 4]

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 ∈ {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 CoVeriTeam 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 CoVeriTeam

Evaluation Through Case Studies

Table: Examples of cooperative techniques in literature

Technique	Year
Counterexample Checking [18]	2012
Conditional Model Checking [7]	2012
Slicing and Symbolic Execution [19]	2013
Precision Reuse [12]	2013
Algorithm Selection [20, 14]	2014, 2017
Witness Validation [5, 4]	2015, 2016
Execution-Based Validation [6]	2018
Reducer [9]	2018
Executable Counterexamples [15]	2018
CoVeriTest [8]	2019
CondTest [11]	2019
MetaVal [13]	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.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). doi:10.1007/978-3-642-28756-5_38

[2] Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). doi:10.1007/978-3-030-17502-3_9

[3] Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS (3). pp. 167–175. LNCS 11429, Springer (2019). doi:10.1007/978-3-030-17502-3_11

[4] 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

References II

[5] 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

[6] 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

[7] 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

[8] 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

References III

[9] 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

[10] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems (2021)

[11] 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

[12] 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

[13] 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

References IV

[14] 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

[15] Gennari, J., Gurfinkel, A., Kahsai, 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

[16] Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Proc. ISoLA. pp. 608–614. LNCS 7609, Springer (2012). doi:10.1007/978-3-642-34026-0_45

[17] Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012: A program verification competition. STTT **17**(6), 647–657 (2015). doi:10.1007/s10009-015-0396-8

References V

[18] 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

[19] 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)

[20] 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