Cooperative Verification: Towards Reliable Safety-Critical Systems

Dirk Beyer
LMU Munich, Germany

Keynote at FTSCS in Auckland, December 7, 2022

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

Automatic Software Verification

C Program

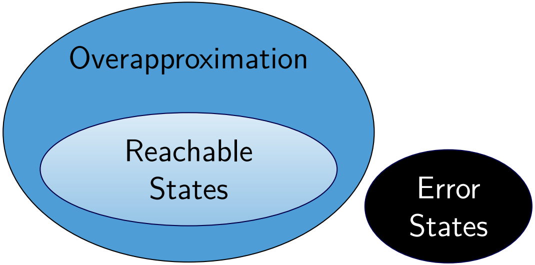
```c
int main() {
   int a = foo();
   int b = bar(a);

   assert(a == b);
}
```

Verification Tool

TRUE+witness
i.e., specification is satisfied

FALSE+witness
i.e., bug found

General method:
Create an overapproximation of the program states /
compute program invariants



Status on Verifiers

- From lack of verifiers to plentitude

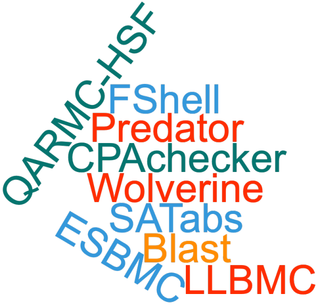
Competitions in Software Verification and Testing

Mature research area, and there are tool competitions:

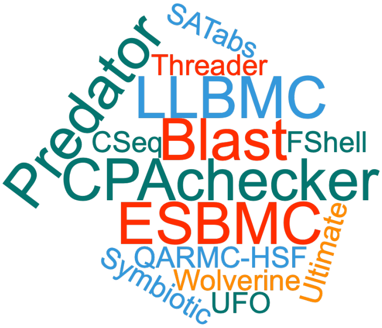
- RERS: off-site, tools, free-style [30]
- SV-COMP: off-site, automatic tools, controlled [1]
- Test-Comp: off-site, automatic tools, controlled [3]
- VerifyThis: on-site, interactive, teams [31]

(alphabetic order)

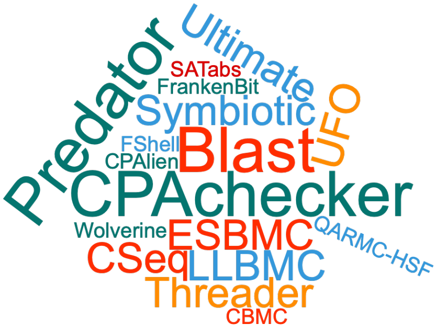
SV-COMP (Automatic Tools 2012)



SV-COMP (Automatic Tools 2013, cumulative)

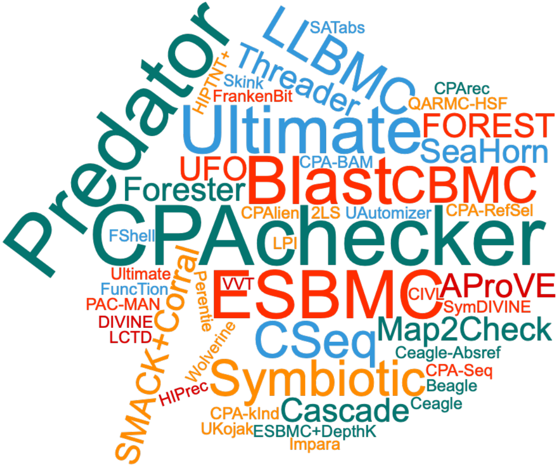


SV-COMP (Automatic Tools 2014, cumulative)

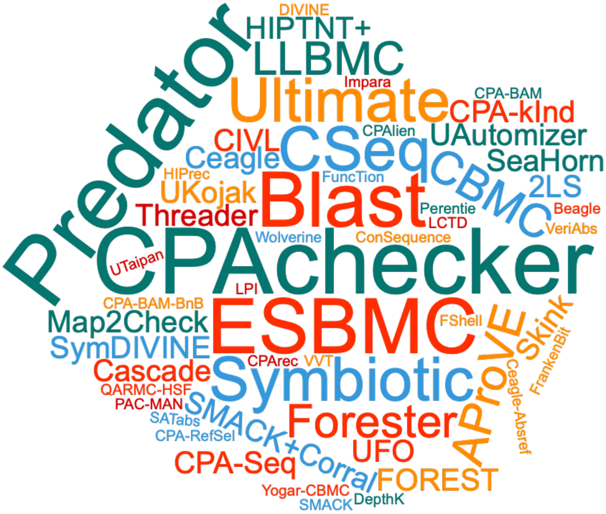


SV-COMP (Automatic Tools 2015, cumulative)

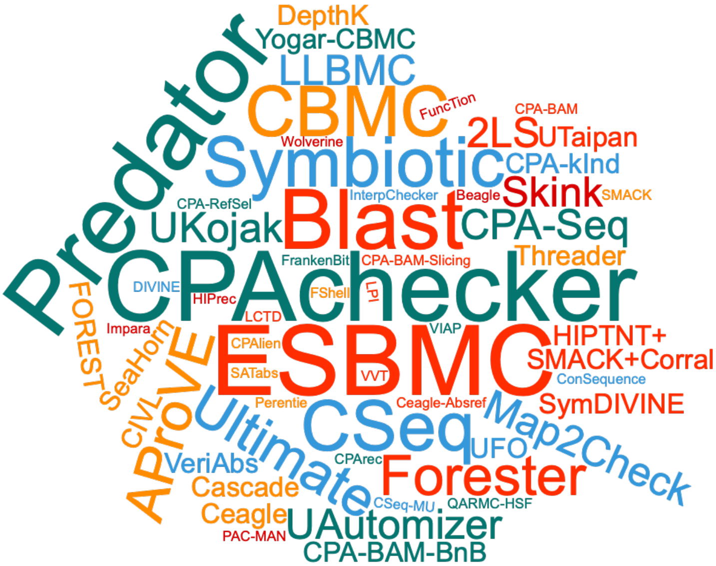
SV-COMP (Automatic Tools 2016, cumulative)



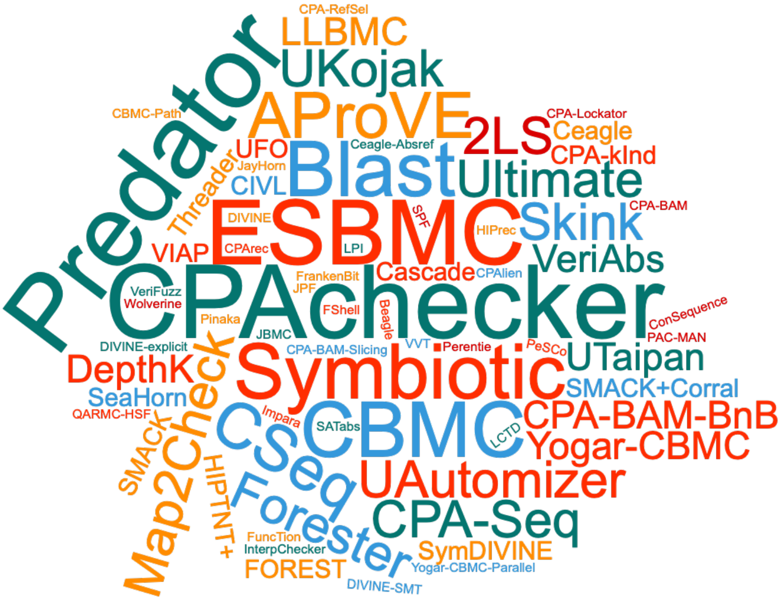
SV-COMP (Automatic Tools 2017, cumulative)



SV-COMP (Automatic Tools 2018, cumulative)

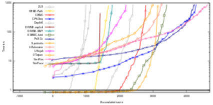


SV-COMP (Automatic Tools 2019, cumulative)

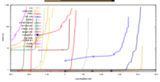


Different Strengths

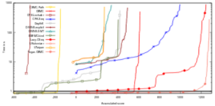
ReachSafety
1. VeriAbs
2. CPA-Seq
3. PeSCo



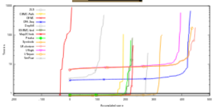
MemSafety
1. Symbiotic
2. PredatorHP
3. CPA-Seq



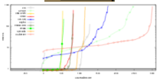
ConcurrencySafety
1. Yogar-CBMC
2. Lazy-CSeq
3. CPA-Seq



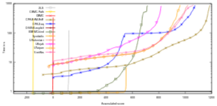
NoOverflows
1. UAutomizer
2. UTaipan
3. CPA-Seq



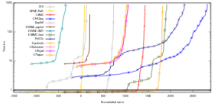
Termination
1. UAutomizer
2. AProVE
3. CPA-Seq



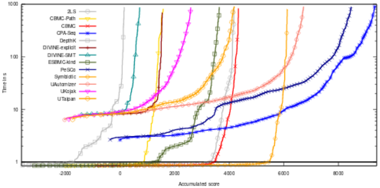
SoftwareSystems
1. CPA-BAM-BnB
2. CPA-Seq
3. VeriAbs



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

Different Techniques

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	✓	✓									✓		✓	✓	✓			
VeriAbs	✓			✓	✓		✓	✓										
VeriFuzz				✓				✓										✓
VIAP																		
Yogar-CBMC	✓			✓							✓		✓			✓		
Yogar-CBMC-Par.	✓			✓							✓		✓			✓		

Competition Report [2]
https://doi.org/10.1007/978-3-030-17502-3_9

Example CPAchecker [18]: Many Concepts

- Included Concepts:
  - CEGAR [27]   Interpolation [21, 10]
  - Configurable Program Analysis [13, 14]
  - Adjustable-block encoding [19]
  - Conditional model checking [12]
  - Verification witnesses [8, 6]
  - Various abstract domains: predicates, intervals, BDDs, octagons, explicit values
- Available analyses approaches:
  - Predicate abstraction [4, 19, 14, 22]
  - Impact algorithm [32, 26, 10]
  - Bounded model checking [28, 10]
  - k-Induction [9, 10]
  - IC3/Property-directed reachability [5]
  - Explicit-state model checking [21]
  - Interpolation-based model checking [20]

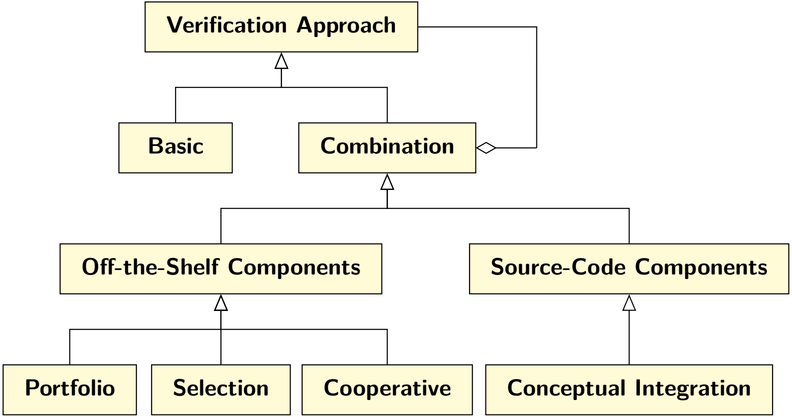
Insights from Software Model Checking

- Verifiers have different strengths
- There are plenty of tools
- $\Rightarrow$ Cooperative Verification Approaches

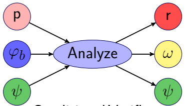
Cooperative Verification — Think big!

- Introduce a new level!
- Current tools are "low level" components (engines)
- Construct combinations
- Clear Interfaces
  via, e.g., Conditions, Witnesses, Test Suites
- Success: SAT, SMT
- See also: Little Engines [33], Evidential Tool Bus [29]

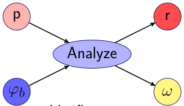
Approaches for Combinations



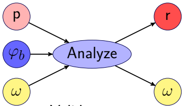
Graphical Visualization of the Coop Framework



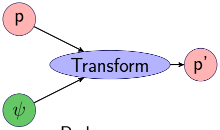
Conditional Verifier



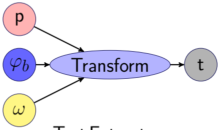
Verifier



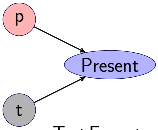
Validator



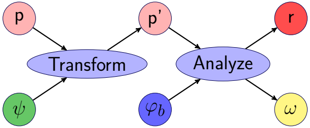
Reducer



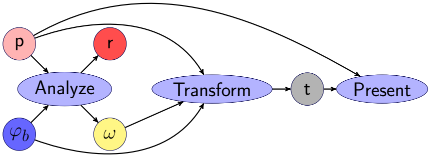
Test Extractor



Test Executor



Conditional Verifier constructed from
Reducer and Verifier



Execution-Based Validation constructed from
Verifier, Test Extractor, and Test Executor

Definition of Cooperative Verification

An approach is called **cooperative verification**, if

- identifiable *actors* pass information in form of
- identifiable *artifacts* towards the common objective of
- solving a *verification* problem,

where at least two of these actors are *analyzers*.

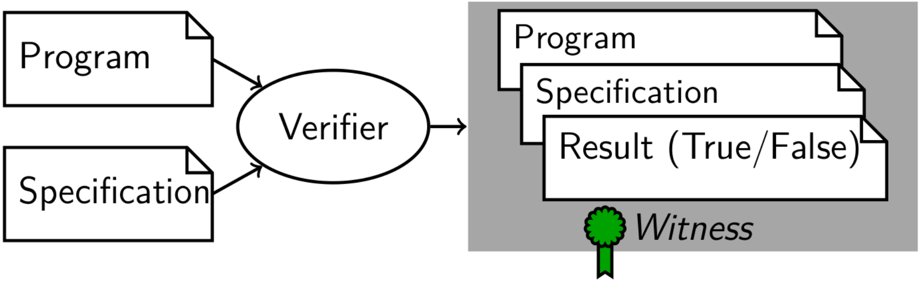
Definition of Cooperative Verification

Examples for notions:

- Identifiable actor:
  off-the-shelf components, binaries, agents, web services
- Identifiable artifacts:
  programs, witnesses, ARGs, test suites
- Verification problem:
  verification task, test task, feasibility check, refinement

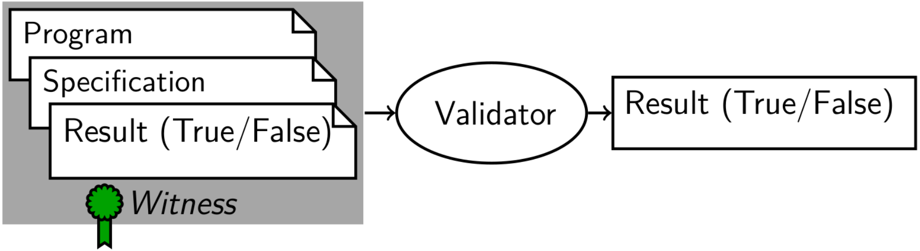
(A1) Software Verification with Witnesses

Witnesses are an important interface between tools.



[8, Proc. FSE 2015] [6, Proc. FSE 2016] [7, TOSEM 2022]

(A1) Witness Validation



- Validate untrusted results
- Easier than full verification

(A2) Example Combination (in DSL CoVeriTeam)

CoVeriTeam: Language and Tool [16, Proc. TACAS 2022]

Algorithm 1 Witness Validation [8, 6]

Input: Program p, Specification s
Output: Verdict

```
verifier := Verifier("Ultimate Automizer")
validator := Validator("CPAchecker")
result := verifier.verify(p, s)
if result.verdict ∈ {TRUE, FALSE} then
    result = validator.validate (p, s, result.witness)
return (result.verdict, result.witness)
```

(A3) Simple Combination without Cooperation

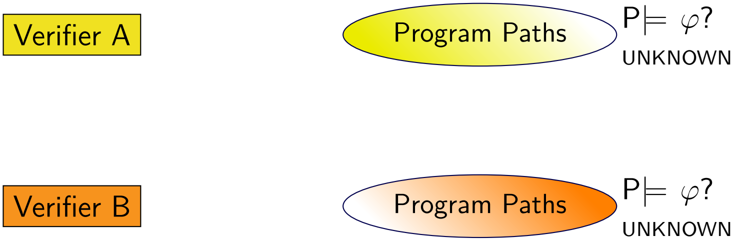
Often, even simple combinations help!

Portfolio construction using off-the-shelf
verification tools [17, Proc. FASE 2022]

Consider AWS category (177 tasks) in SV-COMP 2022:
CBMC: 69 (8 wrong)
CoVeriTeam-Parallel-Portfolio: 147 (3 wrong)
(improvement did not require any change in a verification tool)

(A4) Facing Hard Verification Tasks

Given: Program P$\models \varphi$?



(A4) Facing Hard Verification Tasks

Given: Program $P \models \varphi$?

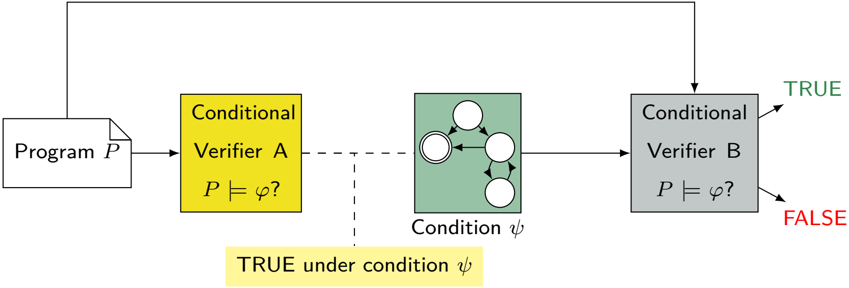
Verifier A — Program Paths — $P \models \varphi$? UNKNOWN

Verifier B — Program Paths — $P \models \varphi$? UNKNOWN

Verifier A + Verifier B — Program Paths — $P \models \varphi$ ✓

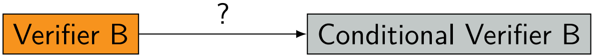
e.g., conditional model checking

(A4) Conditional Model Checking



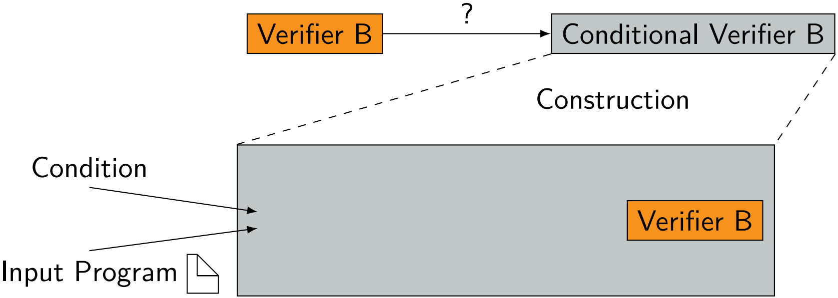
Proc. FSE 2012 [12]

(A4) Reducer-Based Construction



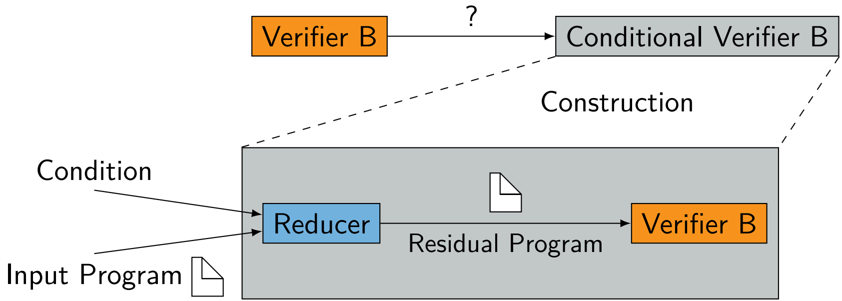
Proc. ICSE 2018 [15]

(A4) Reducer-Based Construction



Proc. ICSE 2018 [15]

(A4) Reducer-Based Construction

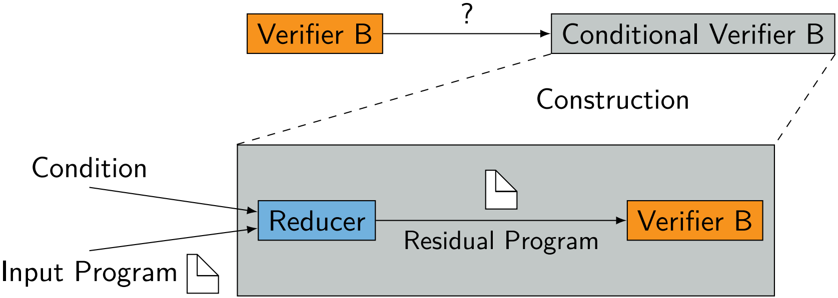


Reducer (preprocessor)

- Builds standard input (C program)
- Representing a subset of paths
- Contains at least all non-verified paths

Proc. ICSE 2018 [15]

(A4) Reducer-Based Construction



Reducer (preprocessor)

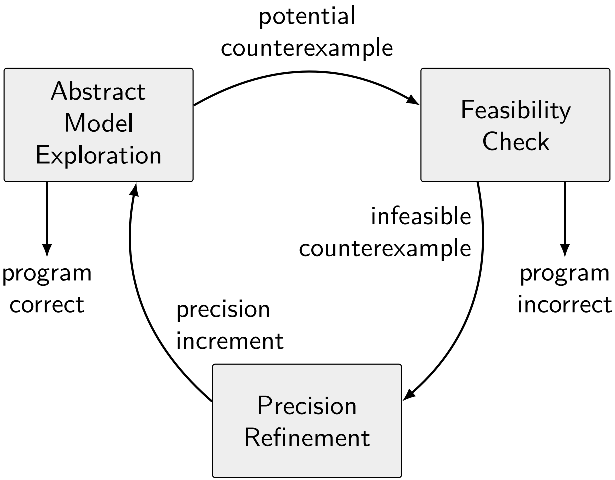
- Builds standard input (C program)
- Representing a subset of paths
- Contains at least all non-verified paths

\+ Verifier-unspecific approach

\+ Many conditional verifiers possible

Proc. ICSE 2018 [15]

(A5) CEGAR

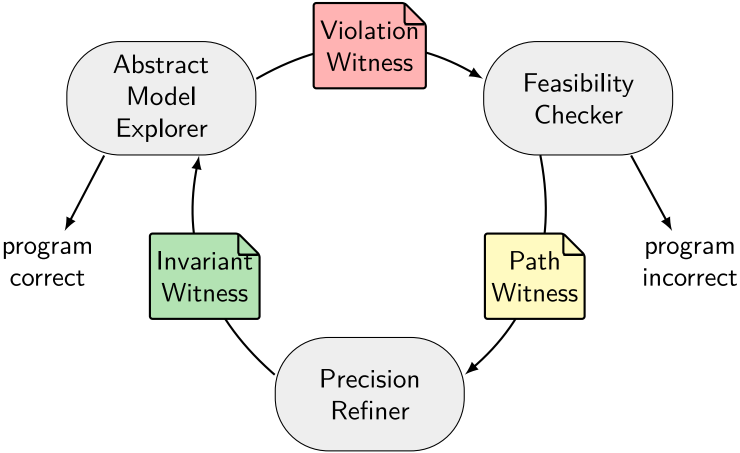

potential
counterexample
Abstract
Model
Exploration
Feasibility
Check
infeasible
counterexample
program
correct
program
incorrect
precision
increment
Precision
Refinement

(A5) Modularization of CEGAR

- CEGARdefines I/O interfaces
- But instances not exchangeable
- Aim: generalize CEGAR, allow exchange of components

$\Rightarrow$ Modular reformulation

(A5) Workflow of modular CEGAR



Proc. ICSE 2022 [11]

(A6) Interactive and Automatic Methods

- How to achieve cooperation between automatic and interactive verifiers?
- Idea: Try to use existing interfaces for information exchange
- [25, Proc. SEFM '22]

```
//@ensures \return==0;
int main() {
  unsigned int x = 0;
  unsigned int y = 0;
  //@loop invariant x==y;
  while (nondet_int()) {
    x++;
    //@assert x==y+1;
    y++;
  }
  assert(x==y);
  return 0;
}
```

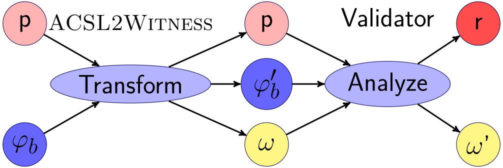
ACSL-annotated program, as used by Frama-C

```
...
<node id="q1">
<data key="invariant">( y == x )</data>
<data key="invariant.scope">main</data>
</node>
<edge source="q0" target="q1">
<data key="enterLoopHead">true</data>
<data key="startline">6</data>
<data key="endline">6</data>
<data key="startoffset">157</data>
<data key="endoffset">165</data>
</edge>
...
```

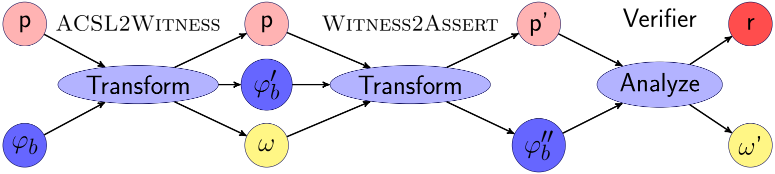
GraphML-based witness automaton generated by automatic verifiers

(A6) From Components: Construct Interactive Verifiers

- Turn a witness validator into an interactive verifier:



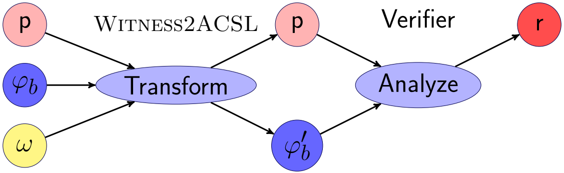
- Turn an automatic verifier into an interactive verifier:



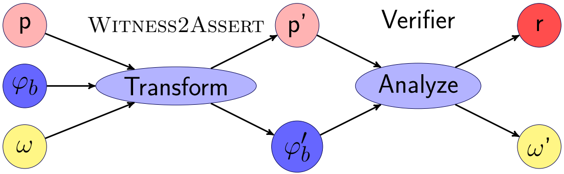
- Annotating in ACSL is more human-readable than witness automata
- Works for a wide range of automatic verifiers/validators

(A6) Component Framework: Constructing Validators

- Turn an interactive verifier (Frama-C) into a validator:

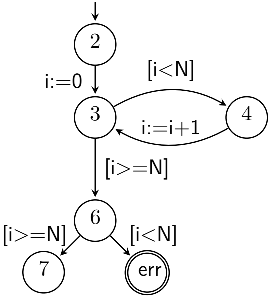


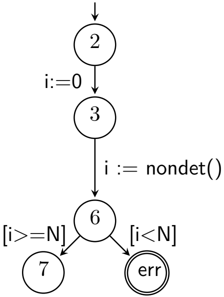
- Turn an automatic verifier into a validator [24, CAV '20]:



(A7) Loop Abstraction

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i>=N);
}
```





- Instead of a precise acceleration, we can also apply an overapproximating *abstraction*
- Here we just havoc all variables that are modified in the loop, but more elaborate abstraction strategies exist

(A7) Example: Havoc Abstraction

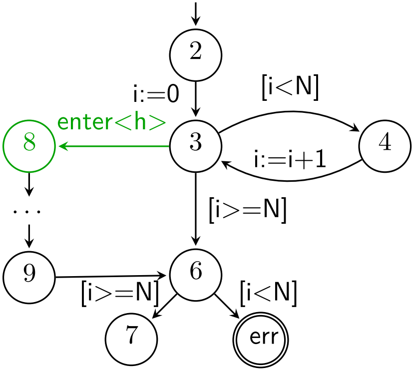
```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i>=N);
}
```

```
void main() {
  int i = 0;
  if (i<N) {
    i = nondet();
    assume(!(i<N));
  }
  assert (i>=N);
}
```

- Havoc Abstraction: if loop is entered, havoc all input variables of the loop and perform one loop iteration, then assume the loop is left
- Only sound if the loop body does not contain assertions
- Overapproximation, but sometimes enough (as in this example)

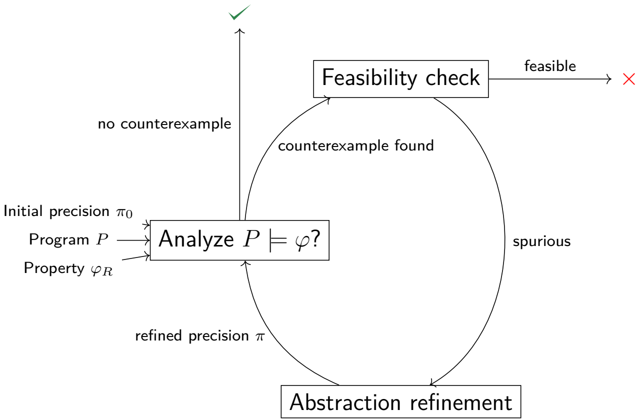
(A7) Configurable Solution a la CPAChecker

- Use the CFA as interface
- Add our loop abstractions next to the original loop
- Mark the entry nodes of each added alternative with an identifier for the applied strategy: $\sigma : L \to S$
- In the example:
  $S = \{b, h\}$
  $\sigma(8) = h$
  $\sigma(l) = b$ for all l except 8
- Select allowed strategies during state-space exploration using σ
- [23, Proc. SEFM '22]



(A7) CEGAR: Feasibility of Counterexamples

- In general, CEGAR works as shown on the right
- For our approach, we need to rethink what it means if a counterexample is feasible: Even if the path formula is satisfiable, the counterexample is only feasible if there are no over-approximating strategies used along the path!



(A7) Accessibility of Loop Abstractions via Patches

- We provide loop abstractions as patches
- We also output a the abstracted version of the program in case we found a proof
- Can be used independently by other tools
- Does this work in practice?
  ⇒ Experiments

```
--- havoc.c
+++ havoc.c
-14,13 +14,16
    return;
 }

 int main(void) {
   unsigned int x = 1000000;
- while (x > 0) {
- x -= 4;
+ // START HAVOCSTRATEGY
+ if (x > 0) {
+ x = __VERIFIER_nondet_uint();
   }
+ if (x > 0) abort();
+ // END HAVOCSTRATEGY
   __VERIFIER_assert(!(x % 4));
```

Conclusion

- External combinations
- Interfaces
- Combinations and Cooperation
- Leverage Cooperation between Tools

References I

[1] Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/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). https://doi.org/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). https://doi.org/10.1007/978-3-030-17502-3_11

[4] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147

[5] Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Proc. TACAS (1). pp. 3–21. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_1

[6] 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

[7] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579

[8] 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

References II

[9] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42

[10] Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6

[11] Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064

[12] 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

[13] Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51

[14] Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13

[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.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31

References III

[17] Beyer, D., Kanav, S., Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3

[18] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

[19] Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010), https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[20] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). https://doi.org/10.48550/arXiv.2208.05046

[21] Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

[22] 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

[23] Beyer, D., Rosenfeld, M.L., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1

[24] 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

References IV

[25] Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7

[26] Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. IMPACT. In: Proc. FMCAD. pp. 106–113. FMCAD (2012)

[27] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643

[28] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15

[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] 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). https://doi.org/10.1007/978-3-642-34026-0_45

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

[32] McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14

References V

[33] Shankar, N.: Little engines of proof. In: Proc. FME. pp. 1–20. LNCS 2391, Springer (2002). https://doi.org/10.1007/3-540-45614-7_1