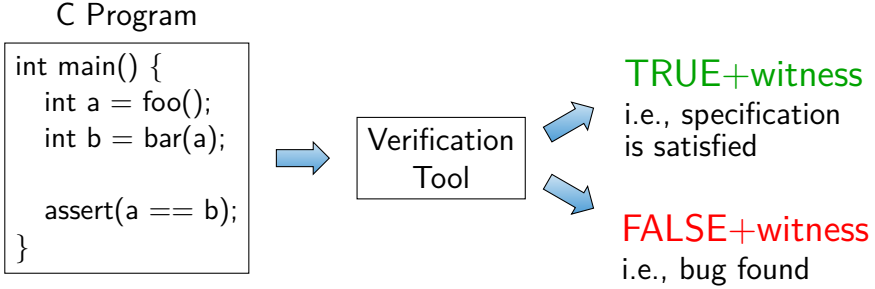
The Transformation Game: Joining Forces for Verification

Dirk Beyer
LMU Munich, Germany

SoSy-Lab @ LMU Munich

May 28, 2025, at Fuzzing Summer School in Singapore

Scope of this presentation: Automatic Software Verification



Literature

The Transformation Game: Joining Forces for Verification, Festschrift 60th Birthday Jost-Pieter Katoen, 2024, available at doi:10.1007/978-3-031-75778-5_9

Transformations in Verification

Given a model under verification (e.g., a C program in software verification), what transformations happen during the verification process?

Transformations in Verification

Given a model under verification (e.g., a C program in software verification), what transformations happen during the verification process?

- From the C language to an *intermediate representation* (IR) used by the verifier
  - E.g., CFA in CPAchecker, LLVM IR in Clang
  - With or without a human-readable exchange format

Transformations in Verification

Given a model under verification (e.g., a C program in software verification), what transformations happen during the verification process?

- From the C language to an *intermediate representation* (IR) used by the verifier
  - E.g., CFA in CPAchecker, Llvm IR in Clang
  - With or without a human-readable exchange format
- From a program path to a path formula

Transformations in Verification

Given a model under verification (e.g., a C program in software verification), what transformations happen during the verification process?

- From the C language to an *intermediate representation* (IR) used by the verifier
  - E.g., CFA in CPAchecker, LLVM IR in Clang
  - With or without a human-readable exchange format
- From a program path to a path formula
- From an error path or invariant to a verification witness

Transformations in Verification

Given a model under verification (e.g., a C program in software verification), what transformations happen during the verification process?

- From the C language to an *intermediate representation* (IR) used by the verifier
  - E.g., CFA in CPAchecker, Llvm IR in Clang
  - With or without a human-readable exchange format
- From a program path to a path formula
- From an error path or invariant to a verification witness

What characteristics do these transformations share?

Common Characteristics of Transformations

- Less time-consuming compared to logical solving (syntactic operations)

Common Characteristics of Transformations

- Less time-consuming compared to logical solving (syntactic operations)
- Often tightly coupled with individual verifiers
  - E.g., encoding the C semantics of program paths into SMT-LIB

Common Characteristics of Transformations

- Less time-consuming compared to logical solving (syntactic operations)
- Often tightly coupled with individual verifiers
  - E.g., encoding the C semantics of program paths into SMT-LIB
- Lack of exchange formats for intermediate products (often data structures)

Common Characteristics of Transformations

- Less time-consuming compared to logical solving (syntactic operations)
- Often tightly coupled with individual verifiers
  - E.g., encoding the C semantics of program paths into SMT-LIB
- Lack of exchange formats for intermediate products (often data structures)

Vision: Modular Transformation Paradigm

- Standalone and reusable transformers to construct verifiers
- Well-defined interfaces and exchange formats
- Construction recipes: easy to build new verifiers for different applications

Inputs and Outputs of Transformers: Artifacts

Type	Notation	Usage
Model	$\mathcal{M}$	Description of the system under verification
Specification	Φ	Expected behavior of the system under verification
Verdict	$\mathcal{R}$	Decision on whether a model satisfies a specification
Witness	Ω	Certificate explaining the verdict of a tool
Verification condition	$\mathcal{VC}$	Set of constraints that encode the behavior of a model

Example Transformers

Type	Signature	Functionality
Translator	$\mathcal{M} \mapsto \mathcal{M}$	Translates a model to a behaviorally equivalent one in a different language
Encoder	$\mathcal{M} \mapsto \mathcal{VC}$	Describes partial or complete behavior of a model as a verification condition
Specification transformer	$\mathcal{M} \times \Phi \mapsto \mathcal{M} \times \Phi$	Converts a verification task to an equisatisfiable one with a different specification
Witness transformer	$\mathcal{M} \times \Omega \mapsto \Omega$	Transforms a witness for a model to another witness, e.g., by making it more precise
Pruner	$\mathcal{M} \times \Omega \mapsto \mathcal{M}$	Removes irrelevant or fully-explored parts of a model based on a witness

Cooperative Verification

There is no silver bullet for verification.

- Different verifiers suitable for different (parts of) problems
- Verifiers used as basic building blocks for verification (like SMT solvers)

Status on Verifiers

- From lack of verifiers to plentitude
- 76 verification tools available [32]

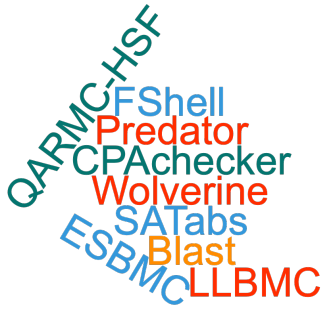
Competitions in Software Verification and Testing

Mature research area, and there are tool competitions:

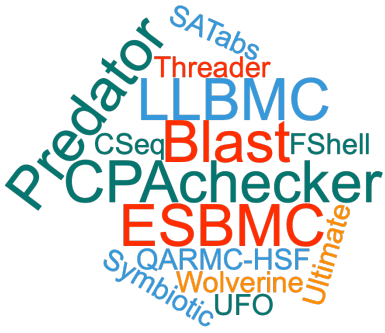
- RERS: off-site, tools, free-style [43]
- SV-COMP: off-site, automatic tools, controlled [3]
- Test-Comp: off-site, automatic tools, controlled [4]
- VerifyThis: on-site, interactive, teams [44]

(alphabetic order)

SV-COMP (Automatic Tools 2012)



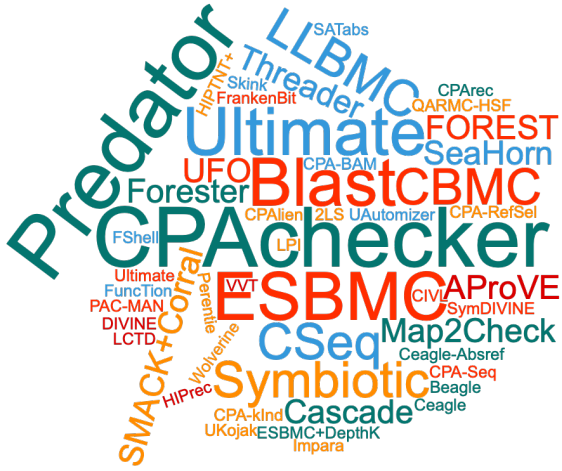
SV-COMP (Automatic Tools 2013, cumulative)



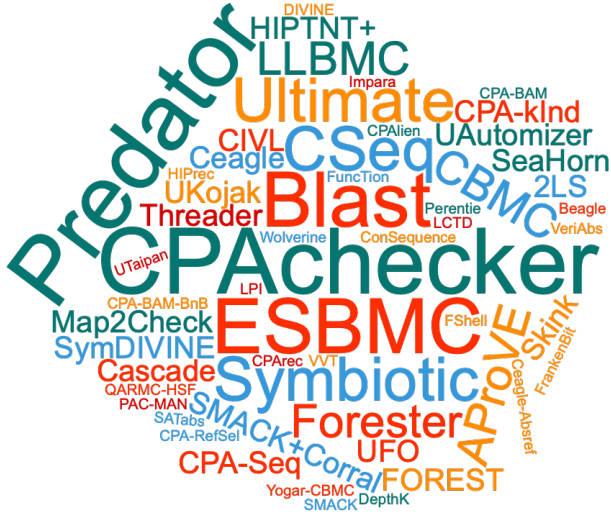
SV-COMP (Automatic Tools 2014, cumulative)

SV-COMP (Automatic Tools 2015, cumulative)

SV-COMP (Automatic Tools 2016, cumulative)

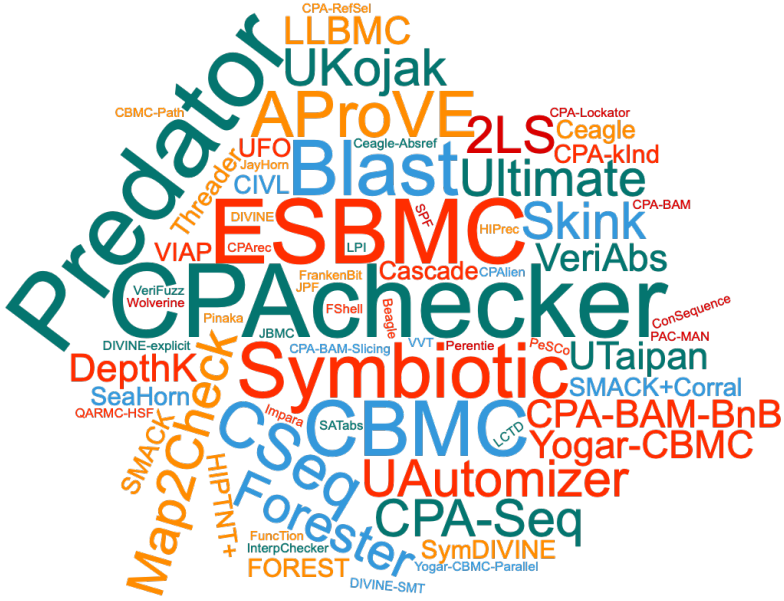


SV-COMP (Automatic Tools 2017, cumulative)

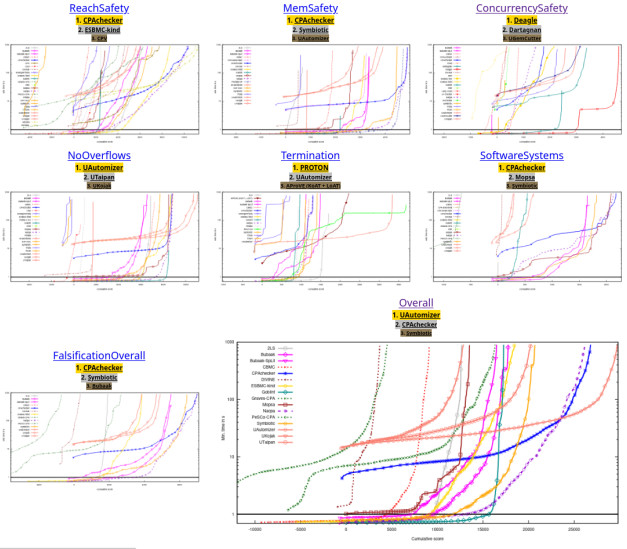


SV-COMP (Automatic Tools 2018, cumulative)

SV-COMP (Automatic Tools 2019, cumulative)



Different Strengths



https://sv-comp.sosy-lab.org/2025/results

Different Techniques (Extract from Report)

Table 8: Algorithms and techniques used by the participating tools; ⌀ for inactive, meta for meta verifiers, and new for first-time participants

Tool	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	Algorithm Selection	Portfolio	Task Translation
2LS				✓	✓			✓	✓		✓						✓				
AISE			✓																		
APROVE																	✓				
BRICK	✓		✓	✓				✓								✓					
BUBAAK			✓								✓					✓	✓			✓	
BUBAAK-SPLIT			✓		✓						✓				✓	✓	✓		✓	✓	
CBMC⌀				✓							✓					✓					
COASTAL⌀			✓																		
CONCURRENTW2T																✓					
COOPERACE meta new																✓			✓	✓	
CPACHECKER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	
CPALOCKATOR⌀	✓	✓					✓				✓	✓	✓	✓		✓					
CPA-BAM-BNB⌀	✓	✓					✓				✓	✓	✓	✓							
CPA-BAM-SMG⌀																					
CPA-W2T⌀						✓						✓			✓						
CPROVER-W2T⌀				✓																	
CPV	✓	✓		✓	✓	✓					✓			✓					✓	✓	✓
CRUX⌀			✓																		
CSEQ⌀				✓							✓					✓					

(continues on next page)

Competition Report [29]
https://doi.org/10.1007/978-3-031-90660-2_9

Example CPAchecker [20]: Many Concepts

- Included Concepts:
  - CEGAR [37] Interpolation [24, 12]
  - Configurable Program Analysis [15, 16]
  - Adjustable-block encoding [21]
  - Conditional model checking [14]
  - Verification witnesses [10, 8]
  - Various abstract domains: predicates, intervals, BDDs, octagons, explicit values
- Available analyses approaches:
  - Predicate abstraction [6, 21, 16, 25]
  - Impact algorithm [49, 31, 12]
  - Bounded model checking [38, 12]
  - k-Induction [11, 12]
  - IC3/Property-directed reachability [7]
  - Explicit-state model checking [24]
  - Interpolation-based model checking [22]

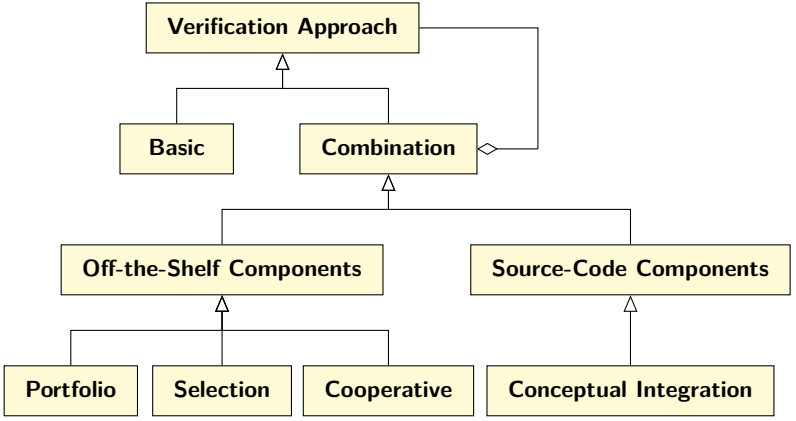
Insights from Software Model Checking

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

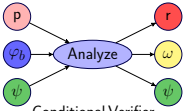
Cooperative Verification — Think big!

- Introduce a new level!
- Current tools should become "low level" components (engines)
- Construct combinations
- Clear Interfaces
  via, e.g., Conditions, Witnesses, Test Suites
- Success: SAT, SMT (common interfaces, usable as libraries)
- See also: Little Engines [52], Evidential Tool Bus [39]

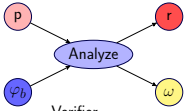
Approaches for Combinations [30]



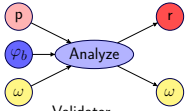
Graphical Visualization of the Coop Framework [30]



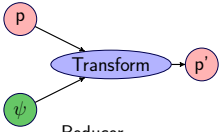
Conditional Verifier



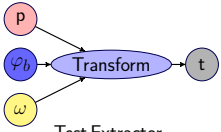
Verifier



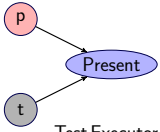
Validator



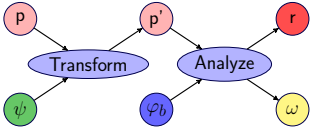
Reducer



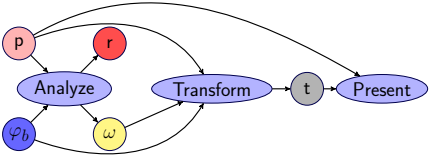
Test Extractor



Test Executor

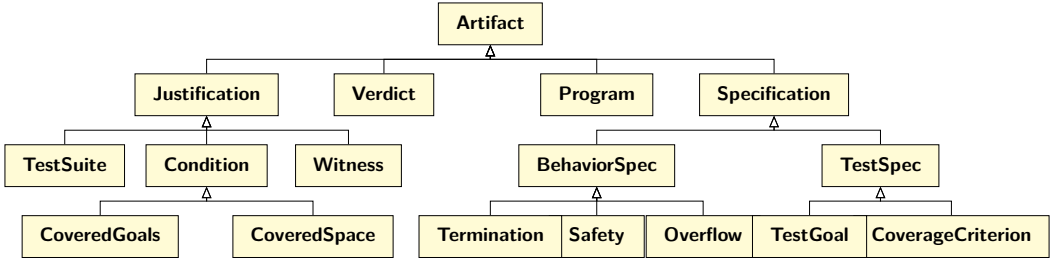


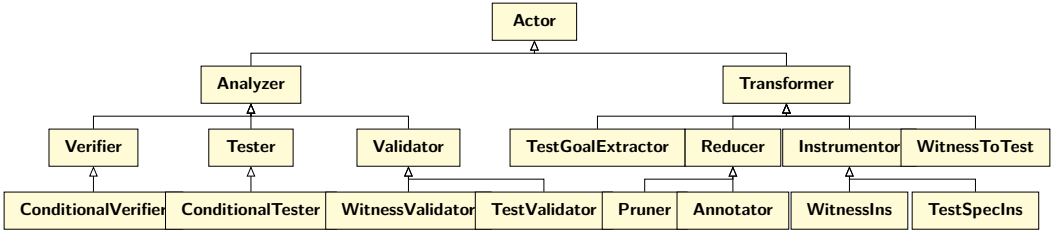
Conditional Verifier constructed from Reducer and Verifier



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

Artifacts and Actors: Classification [18]





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

Application Examples

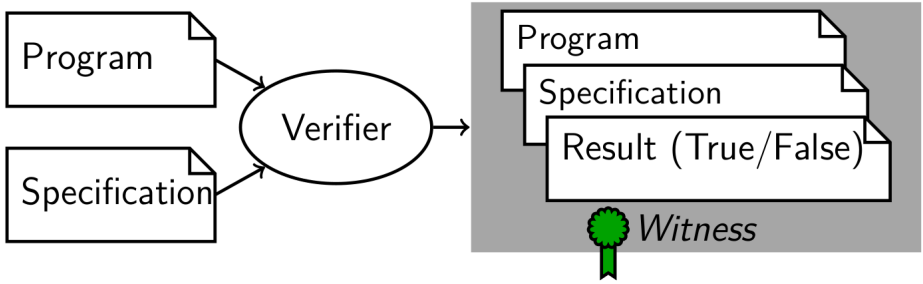
- (A1) Verification Witnesses and Validation
- (A2) LIV: Decomposing Validator
- (A3) CoVeriTeam: Language and Tool for Combination
- (A4) Simple Combinations
- (A5) Btor2C: Transforming from Hardware to Software
- (A6) Transformation-Based Verification with MoXI

Application Examples

- (A7) Transformation of Specifications
- (A8) Conditional Model Checking (CMC)
- (A9) Reducer-Based CMC
- (A10) Modularization of CEGAR
- (A11) Combining Interactive and Automatic Methods
- (A12) Loop Abstraction

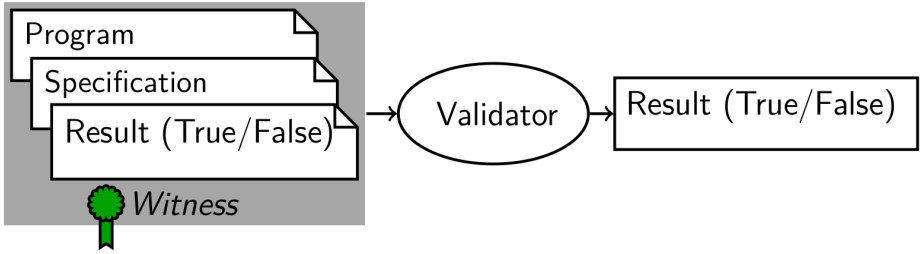
(A1) Software Verification with Witnesses

Witnesses are an important interface between tools.



[10, Proc. FSE 2015] [8, Proc. FSE 2016] [9, TOSEM 2022]

(A1) Witness-Based Result Validation



- Validate untrusted results
- Reestablish proof of correctness or violation
- Easier than full verification

(A1) Verification and Validation

Given program P and specification φ

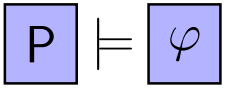
- Verification: **prove** that $P \models \varphi$
  (mainly invariant construction)
- Validation with witness w: **re-prove** that $P \models \varphi$

AI can be used to

- **write** programs
- **suggest** invariants for programs

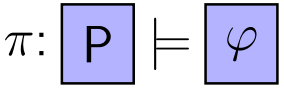
(A1) Correctness Witnesses

Program P, specification φ, proof π



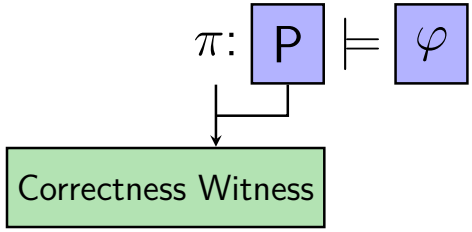
(A1) Correctness Witnesses

Program P, specification φ, proof π



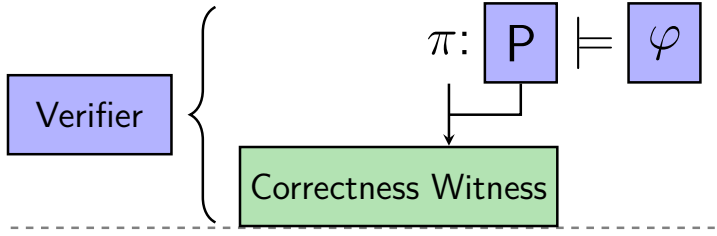
(A1) Correctness Witnesses

Program P, specification φ, proof π



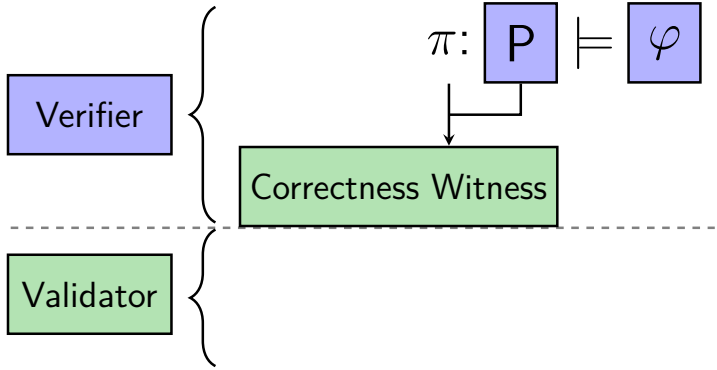
(A1) Correctness Witnesses

Program P, specification φ, proof π



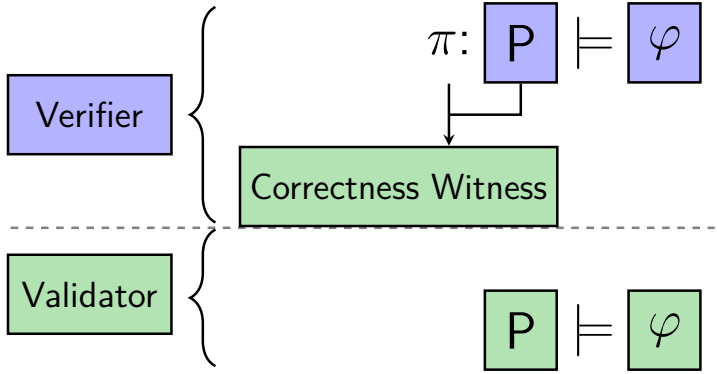
(A1) Correctness Witnesses

Program P, specification φ, proof π



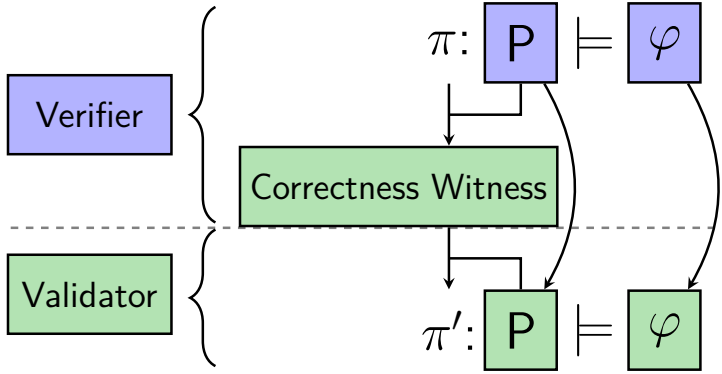
(A1) Correctness Witnesses

Program P, specification φ, proof π



(A1) Correctness Witnesses

Program P, specification φ, proof π



(A1) Example Program and Witness

Program:

```
int main() {
  unsigned char n = __nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned int  s = 0;
  unsigned int  i = 0;
  while (i < n) {
    v = __nondet_uchar();
    s += v;
    ++i;
  }
  if (s < v) {
    reach_error();
    return 1;
  }
  if (s > 65025) {
    reach_error();
    return 1;
  }
  return 0;
}
```

Witness (format v2.0):

```
content:
  - invariant:
      type: loop_invariant
      location:
        file_name: "inv-a.c"
        line: 12
        column: 1
        function: main
      value: "s <= i*255 && 0 <= i && i <= 255 && n <= 255"
      format: c_expression
```

(A1) State of the Art

- 18 validators exist for C and Java
- 4 formats for witnesses exist
  (GraphML and YAML, correctness and validation)
- Competition on Software Verification (SV-COMP) has a validation track

(A2) LIV — Decomposing Validator

[27, Proc. ASE 2023], Idea from A. Appel

Program:

```
1   int x = 0;
2   int sum = 0 ;
3   //@ loop invariant I;
4   while (x<10) {
5     x++;
6     sum+=x;
7   }
8   assert (sum<=55);
```

Proof Obligations:

- $\{P\}s_0\{Inv\}$
- $\{Inv \wedge Cond\}Body\{Inv\}$
- $Inv \Rightarrow Q$

(A2) From Proof Obligations to Straight-Line Programs

Proof Obligations:

- $\{P\}s_0\{Inv\}$ (Base Case)
- $\{Inv \wedge Cond\}Body\{Inv\}$ (Inductiveness)
- $Inv \wedge \neg Cond \Rightarrow Q$ (Safety)

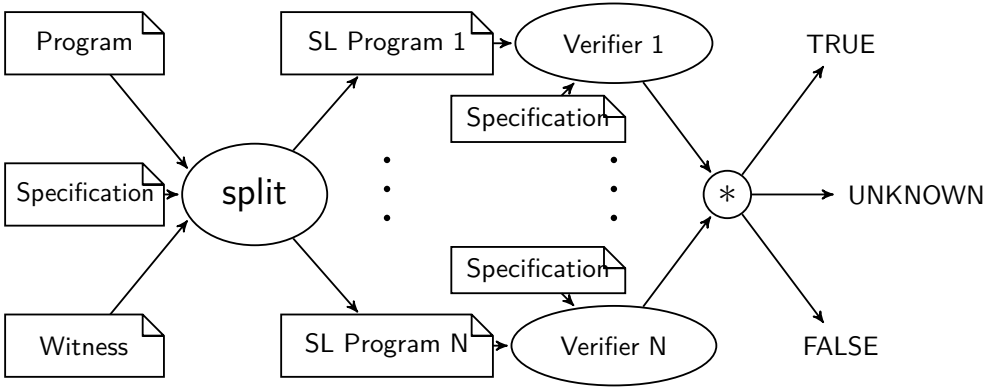
Straight-Line Programs:

```
1  int x = 0;
2  int sum = 0;
3  assert(Inv);
```

```
1  int x = nondet();
2  int sum = nondet();
3  assume(Inv && C);
4  x++;
5  sum += x;
6  assert(Inv);
```

```
1  int x = nondet();
2  int sum = nondet();
3  assume(Inv && ! C);
4  assert(Q);
```

(A2) Workflow of LIV



- Can use any off-the-shelf verifier from SV-COMP as backend
- Small frontend using pycparser for AST-based splitting

(A3) Example Combination (in DSL CoVeriTeam)

CoVeriTeam: Language and Tool [18, Proc. TACAS 2022]

Algorithm 1 Witness Validation [10, 8]

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)
```

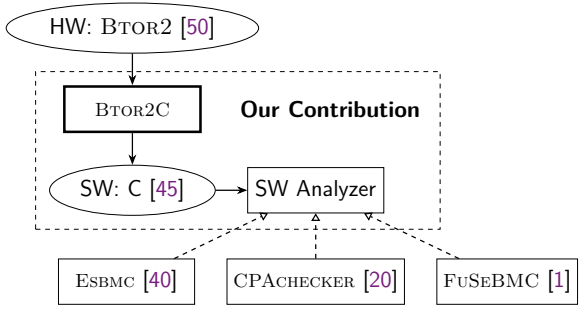
(A4) Simple Combination without Cooperation

Often, even simple combinations help!

Portfolio construction using off-the-shelf
verification tools [19, 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)

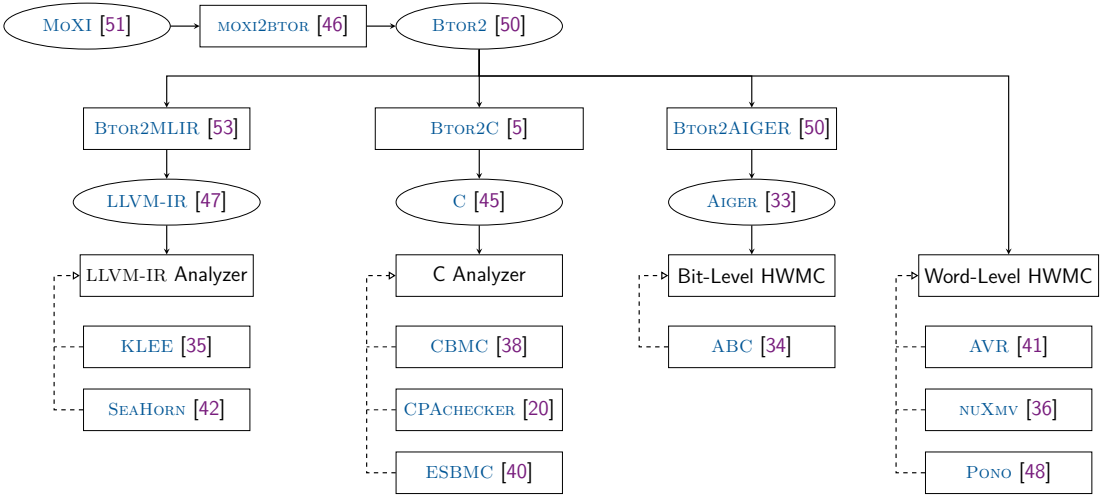
(A5) Btor2C: Transforming from Hardware to Software



- **43** HW-verification tasks uniquely solved by SW analyzers in our evaluation
  $\rightarrow$ enhance HW quality assurance using SW analyzers

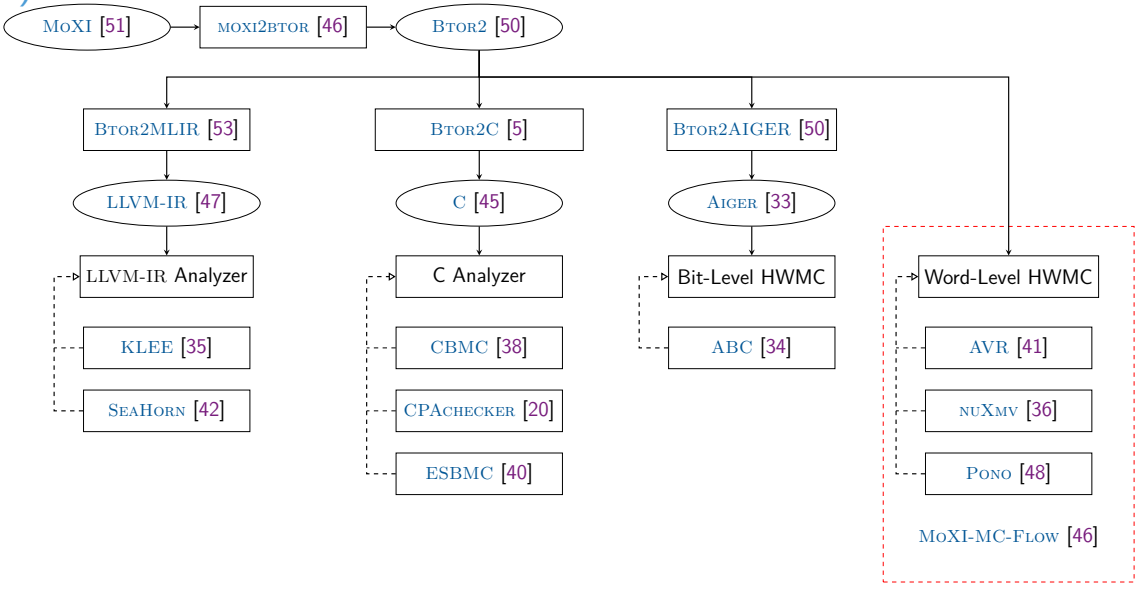
[5, Proc. TACAS '23]

(A6) Transformation-Based Verification with MoXI



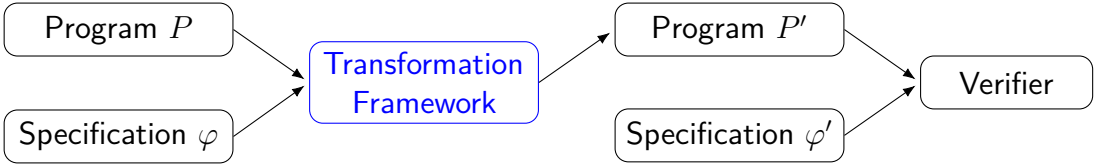
[2, Proc. VSTTE '24]

(A6) Transformation-Based Verification with MoXI



[2, Proc. VSTTE '24]

(A7) Transformation of Specifications



Our framework:

- *Easy to adopt* → Used by three tools in SV-COMP 25
- *Moudular* → Can be used by any verifier supporting SV-COMP syntax
- *Configurable* → The transformations given by *Instrumentation Automata (IA)*

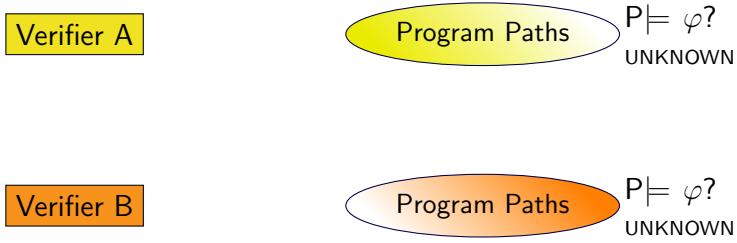
Proc. SPIN 2025

(A7) Results on Termination Reduction

Results (#Tasks)	UAutomizer	2Ls	UAutomizer-R	CPAchecker-R
Correct 377	312	259	**333**	121
Proofs 264	250	189	**264**	55
Alarms 69	62	**70**	69	66

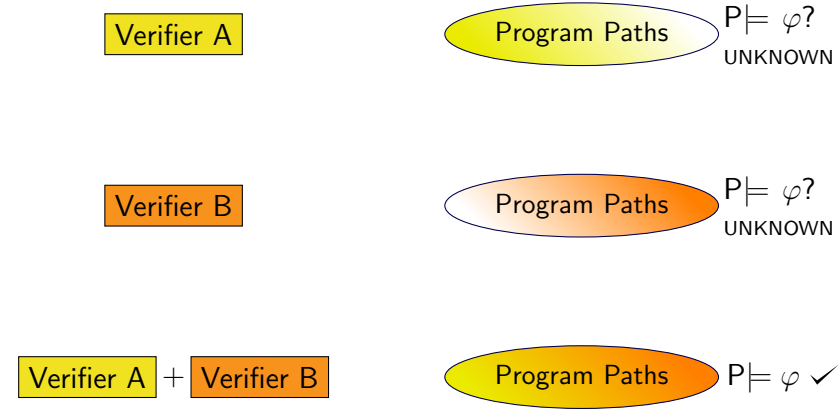
(A8) Facing Hard Verification Tasks

Given: Program P$\models \varphi$?

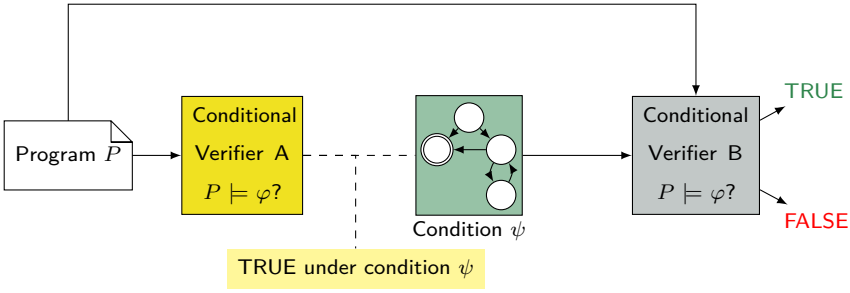


(A8) Facing Hard Verification Tasks

Given: Program P$\models \varphi$?

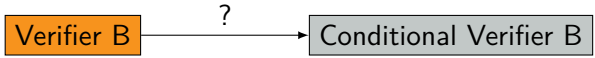


(A8) Conditional Model Checking



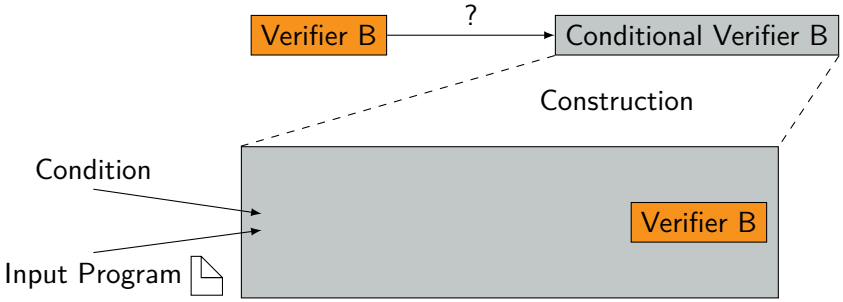
Proc. FSE 2012 [14]

(A9) Reducer-Based Construction



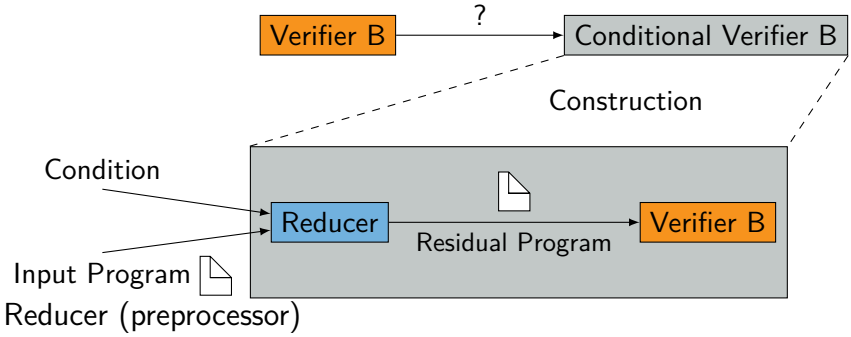
Proc. ICSE 2018 [17]

(A9) Reducer-Based Construction



Proc. ICSE 2018 [17]

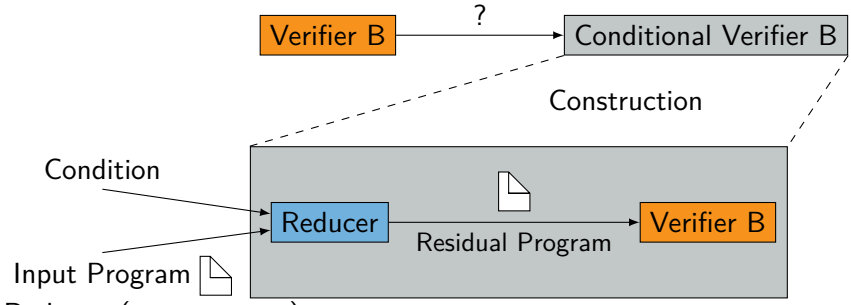
(A9) Reducer-Based Construction



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

Proc. ICSE 2018 [17]

(A9) Reducer-Based Construction



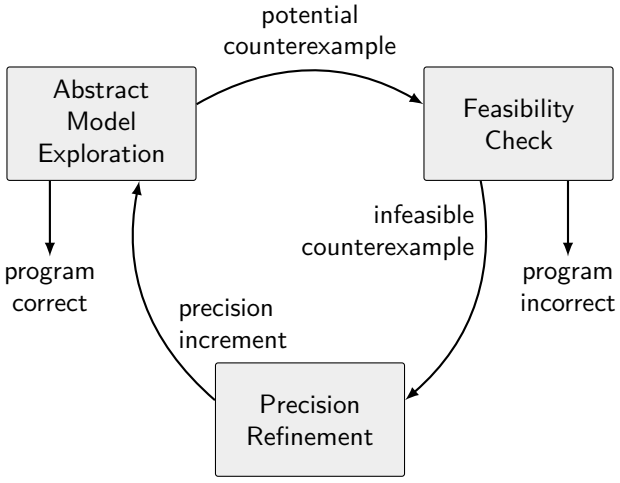
- Builds standard input (C program)
- Representing a subset of paths
- Contains at least all non-verified paths

\+ Verifier-unspecific approach

\+ Many conditional verifiers possible

Proc. ICSE 2018 [17]

(A10) CEGAR

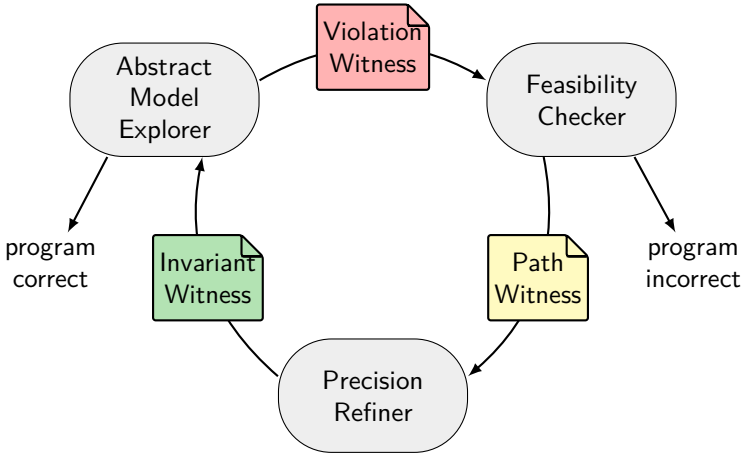

potential
counterexample
Abstract
Model
Exploration
Feasibility
Check
infeasible
counterexample
program
correct
program
incorrect
precision
increment
Precision
Refinement

(A10) Modularization of CEGAR

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

$\Rightarrow$ Modular reformulation

(A10) Workflow of Modular CEGAR



Proc. ICSE 2022 [13]

(A11) Interactive and Automatic Methods

- How to achieve cooperation between automatic and interactive verifiers?
- Idea: Try to use existing interfaces for information exchange
- [28, 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;
}
```

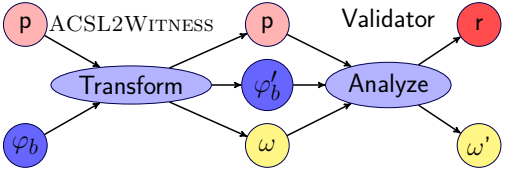
ACSL-annotated program, as used by FRAMA-C

```
...
<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>
...
```

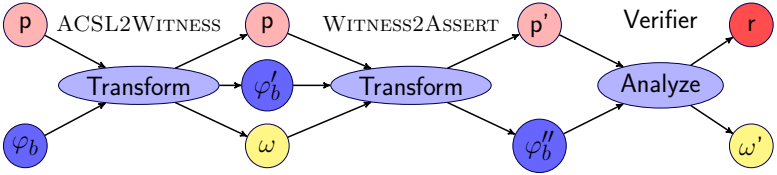
GraphML-based witness automaton generated by automatic verifiers

(A11) From Components: Construct Interactive Verifiers

- Turn a witness validator into an interactive verifier:



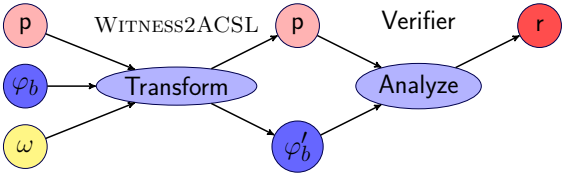
- Turn an automatic verifier into an interactive verifier:



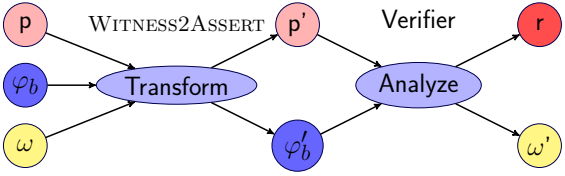
- Annotating in ACSL is more human-readable than witness automata
- Works for a wide range of automatic verifiers/validators

(A11) Component Framework: Constructing Validators

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

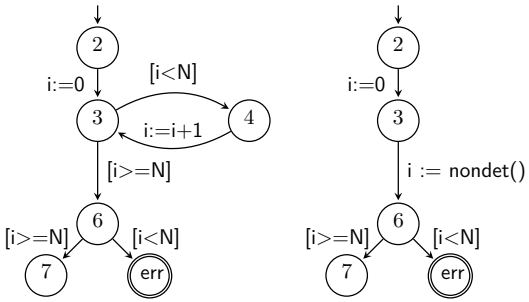


- Turn an automatic verifier into a validator [26, CAV '20]:



(A12) Loop Abstraction

```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i>=N);
7  }
```



- 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

(A12) Example: Havoc Abstraction

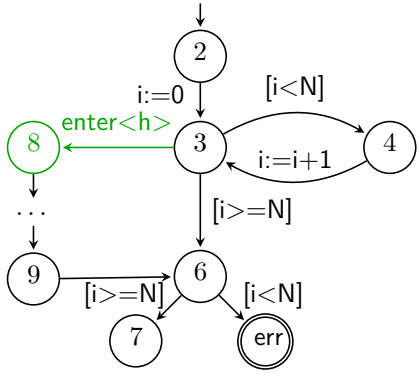
```
1  void main() {
2    int i = 0;
3    while (i<N) {
4      i=i+1;
5    }
6    assert (i>=N);
7  }
```

```
1  void main() {
2    int i = 0;
3    if (i<N) {
4      i = nondet();
5      assume(!(i<N));
6    }
7    assert (i>=N);
8  }
```

- 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)

(A12) 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]



(A12) 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?
  $\Rightarrow$ 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

- Many verification tools and techniques
- External combinations are important
- Interfaces (artifacts, actors)
- Combinations and Cooperation
- Leverage Cooperation between Tools

References I

[1] Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6

[2] Ates, S., Beyer, D., Chien, P.C., Lee, N.Z.: MoXIchecker: An extensible model checker for MoXI. In: Proc. VSTTE 2024. pp. 1–14. LNCS 15525, Springer (2025). https://doi.org/10.1007/978-3-031-86695-1_1

[3] 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

[4] 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

[5] Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS (2). pp. 152–172. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12

[6] 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

[7] 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

[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

References II

[9] 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

[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., 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

References III

[17] 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

[18] 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

[19] 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

[20] 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

[21] Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)

[22] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning **69**(1), 5 (2025). https://doi.org/10.1007/s10817-024-09702-9

[23] Beyer, D., Lingsch-Rosenfeld, M., 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., 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

References IV

[25] 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

[26] 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

[27] Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214

[28] 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

[29] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9

[30] 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

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

[32] Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27

References V

[33] Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University (2007). https://doi.org/10.35011/fmvtr.2007-1

[34] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV. pp. 24–40. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5

[35] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)

[36] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The NUXMV symbolic model checker. In: Proc. CAV. pp. 334–342. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22

[37] 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

[38] 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

[39] 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

[40] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). https://doi.org/10.1145/3238147.3240481

References VI

[41] Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23

[42] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20

[43] 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

[44] 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

[45] ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), https://www.iso.org/standard/74528.html

[46] Johannsen, C., Nukala, K., Dureja, R., Irfan, A., Shankar, N., Tinelli, C., Vardi, M.Y., Rozier, K.Y.: The MoXI model exchange tool suite. In: Proc. CAV. pp. 203–218. LNCS 14681, Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_10

[47] Lattner, C., Adve, V.S.: Llvm: A compilation framework for lifelong program analysis and transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665

[48] Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.W.: Pono: A flexible and extensible SMT-based model checker. In: Proc. CAV. pp. 461–474. LNCS 12760, Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_22

References VII

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

[50] Niemetz, A., Preiner, M., Wolf, C., Biere, A.: BTOR2, BTORMC, and BOOLECTOR 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32

[51] Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: MoXI: An intermediate language for symbolic model checking. In: Proc. SPIN. pp. 26–46. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_2

[52] 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

[53] Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: BTOR2MLIR: A format and toolchain for hardware verification. In: Proc. FMCAD. pp. 55–63. TU Wien Academic Press (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13