Practical Issues of Software Verification

Dirk Beyer

UNIVERSITÄT PASSAU

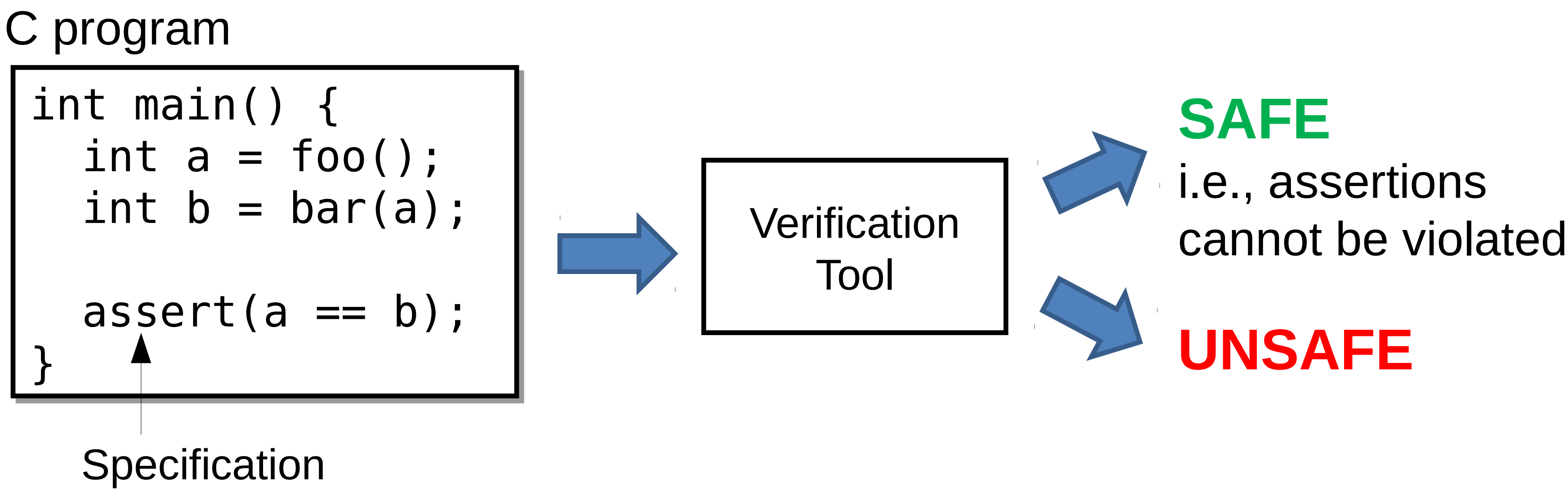
State of the Art

- Much progress in MC theory & algorithms
- Practical issues in industrial applications
- Problems:
  - Large size of individual verification tasks
  - Large number of verification tasks

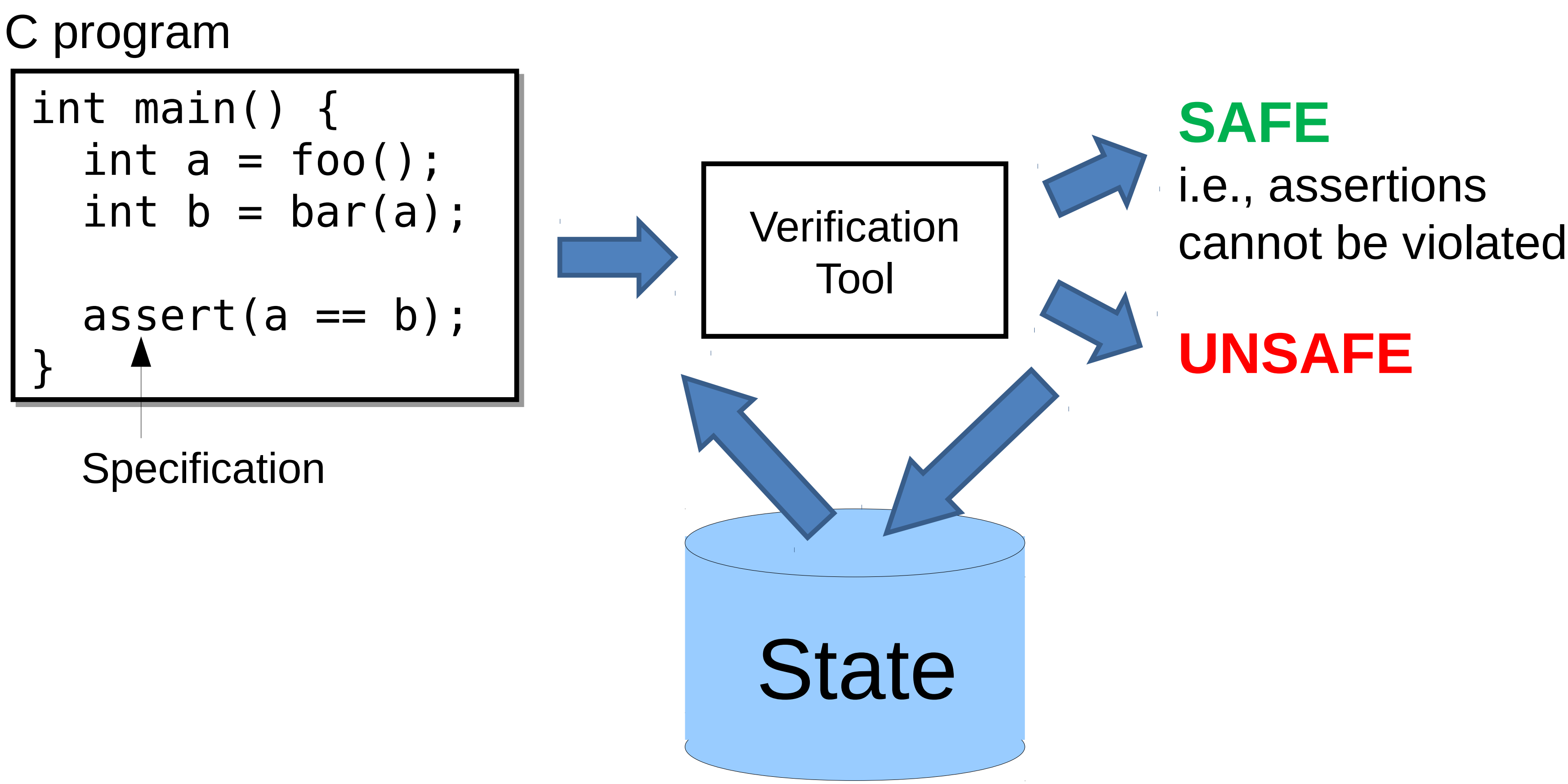
Ideas

- Combine verification tools
- Reuse partial and intermediate results
- Witnesses for results validation
- Tests from Verification

Classic Verification

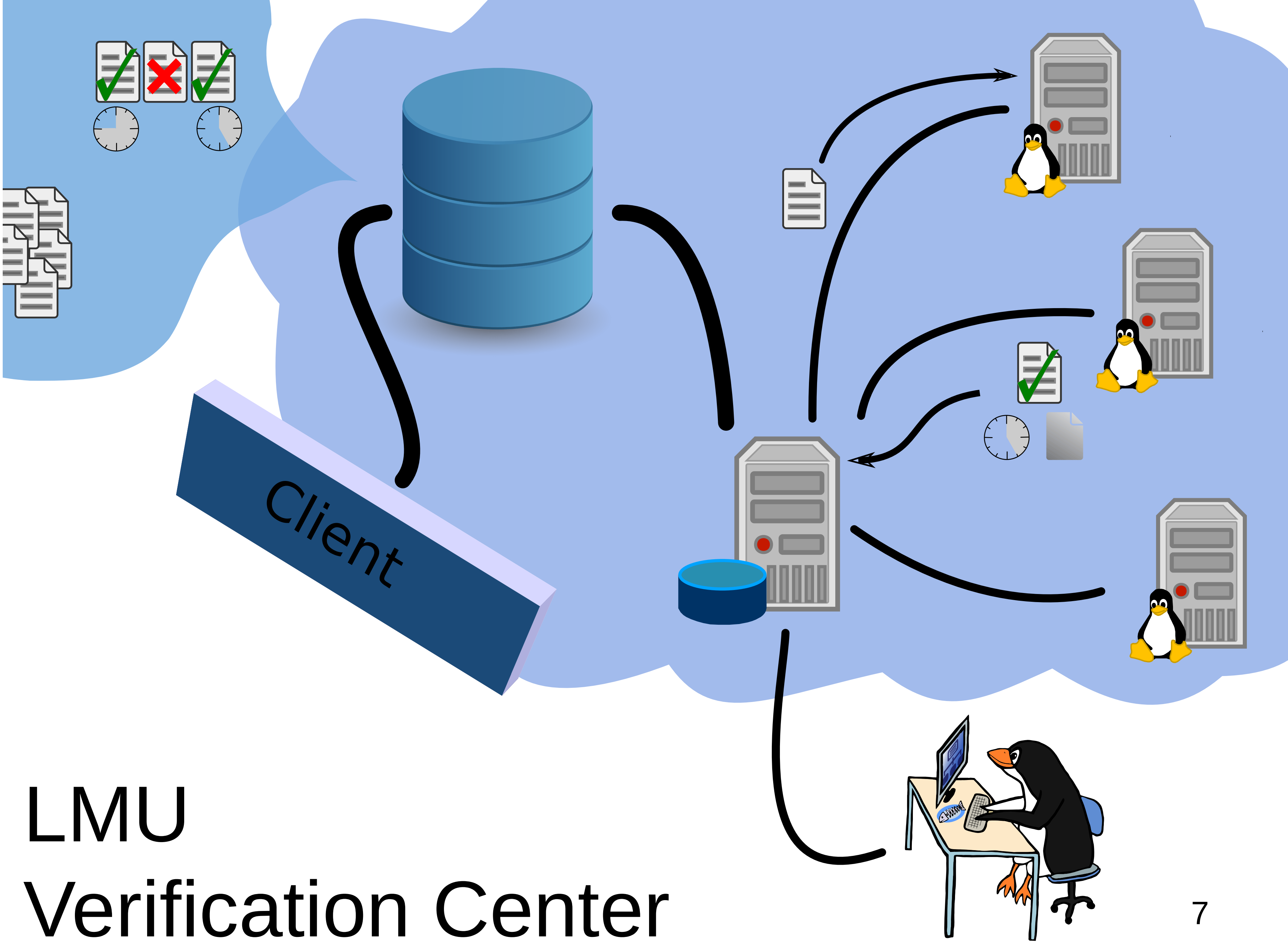


Stateful Verification



Applications of Stateful Verification

- Better performance by remembering successful (intermediate) results
- Regression Verification
- Certify results (verification witnesses)



LMU Verification Center

FSE 2012

Conditional Model Checking: A Technique to Pass Information between Verifiers *‡

Dirk Beyer
University of Passau
Germany

Thomas A. Henzinger
IST Austria
Austria

M. Erkan Keremoglu
Simon Fraser University
Canada

Philipp Wendler
University of Passau
Germany

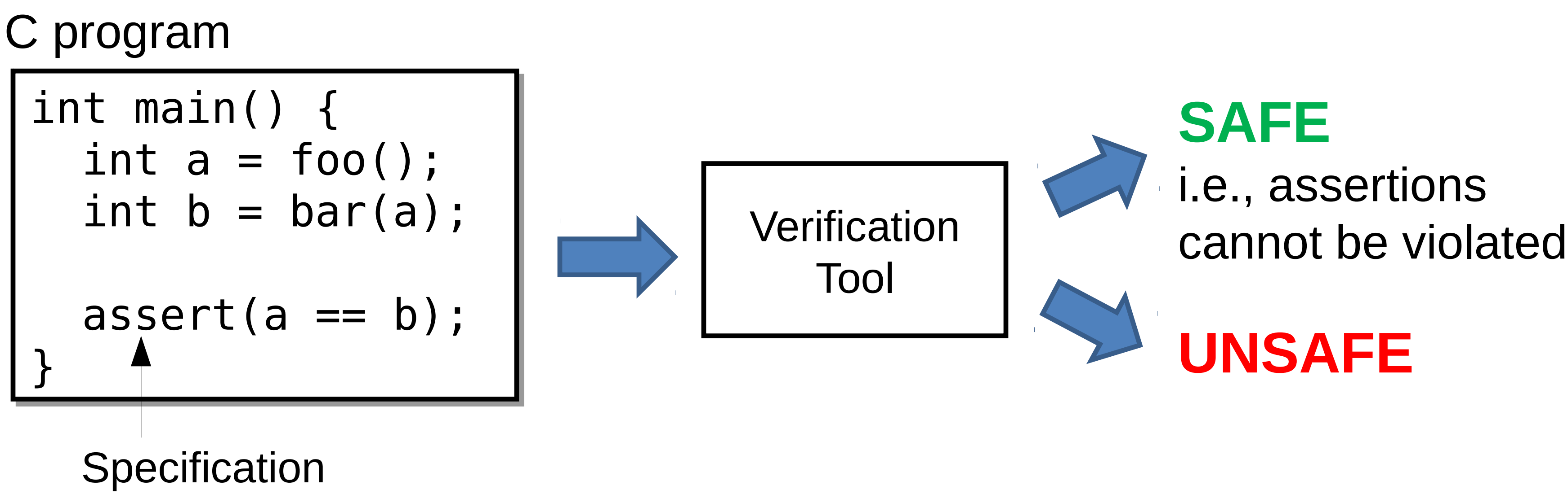
ABSTRACT

Software model checking, as an undecidable problem, has three possible outcomes: (1) the program satisfies the specification, (2) the program does not satisfy the specification, and (3) the model checker fails. The third outcome usually manifests itself in a space-out, time-out, or one component of the verification tool giving up; in all of these failing cases, significant computation is performed by the verification tool before the failure, but no result is reported. We propose to reformulate the model-checking problem as follows, in order to have the verification tool report a summary of the performed work even in case of failure: given a program and a specification, the model checker returns a condition Ψ —usually a state predicate— such that the program satisfies the specification under the condition Ψ —that is, as long as the program does not leave the states in which Ψ is satisfied. In our experiments, we investigated as one major application of conditional model checking the sequential combination of model checkers with information passing. We give the condition that one model checker produces, as input to a second

1. INTRODUCTION

Model checking is an automatic search-based procedure that exhaustively verifies whether a given model (e.g., labeled transition system) satisfies a given specification (e.g., temporal-logic formula) [12, 33]. Since model checking of software is an undecidable problem, there are three possible outcomes of the analysis process: (1) the program satisfies the specification, (2) the program does not satisfy the specification, and (3) the model checker fails. The first outcome can be obtained by the model checker if the abstract model that was computed for the program is sufficient to prove the program correct under the given specification. This outcome can be accompanied by a proof certificate [23]. The second outcome can be obtained by the model checker if an abstract counterexample path is found and can be proven feasible, i.e., a bug that can actually occur in the program. This outcome is usually accompanied by the violating program part in the form of program source code, and sometimes test input to reproduce the error at run-time [4]. The third outcome usually occurs if the model checker runs out

Software Verification

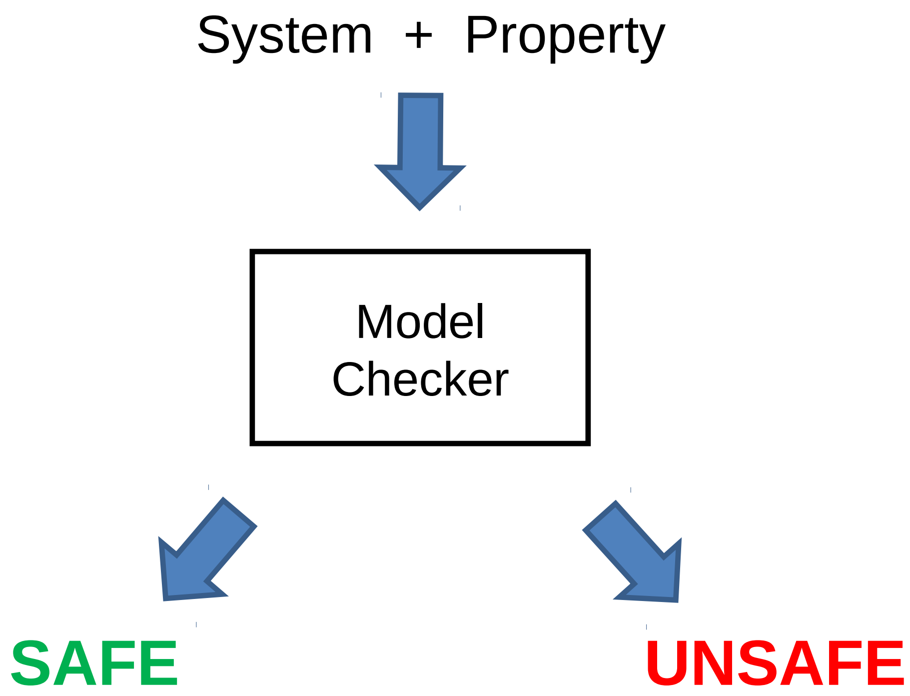


Problem:
Single Analysis not Effective

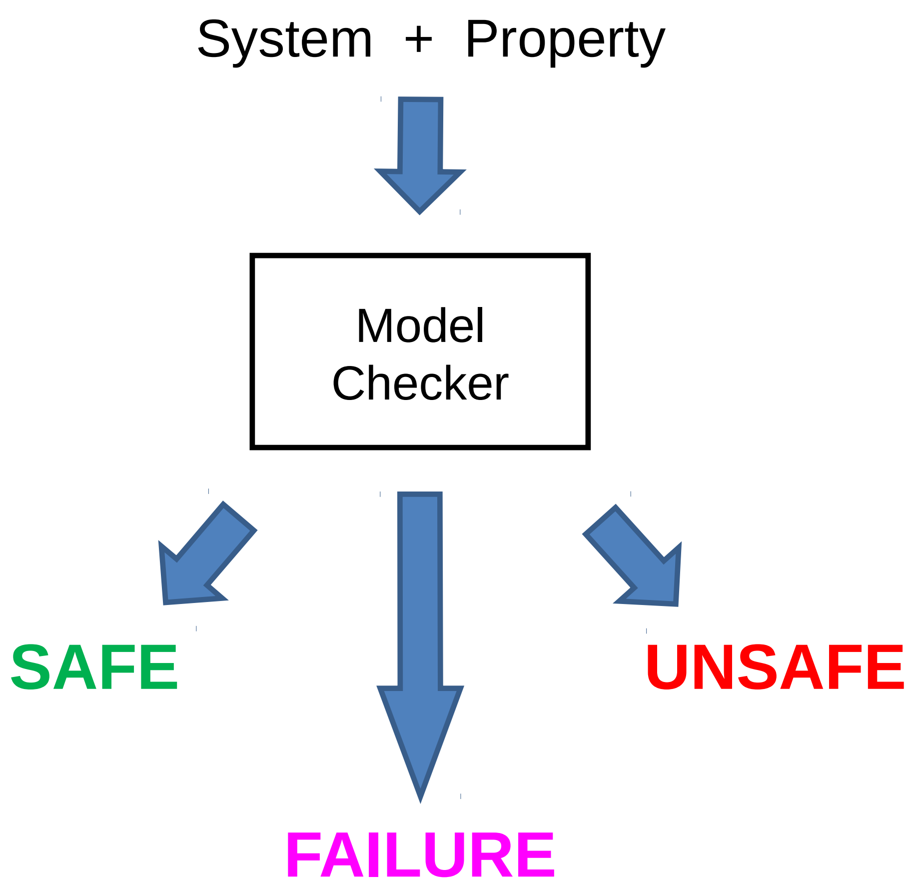
```
void main() {
  if (nondet_int()) {
    int i;
    for (i = nondet_int(); i < 1000000; i++) {
      // ...
    }
    assert(i >= 1000000);

  } else {
    int x = 5;
    int y = 6;
    int r = x * y;
    assert(r >= x);
  }
}
```

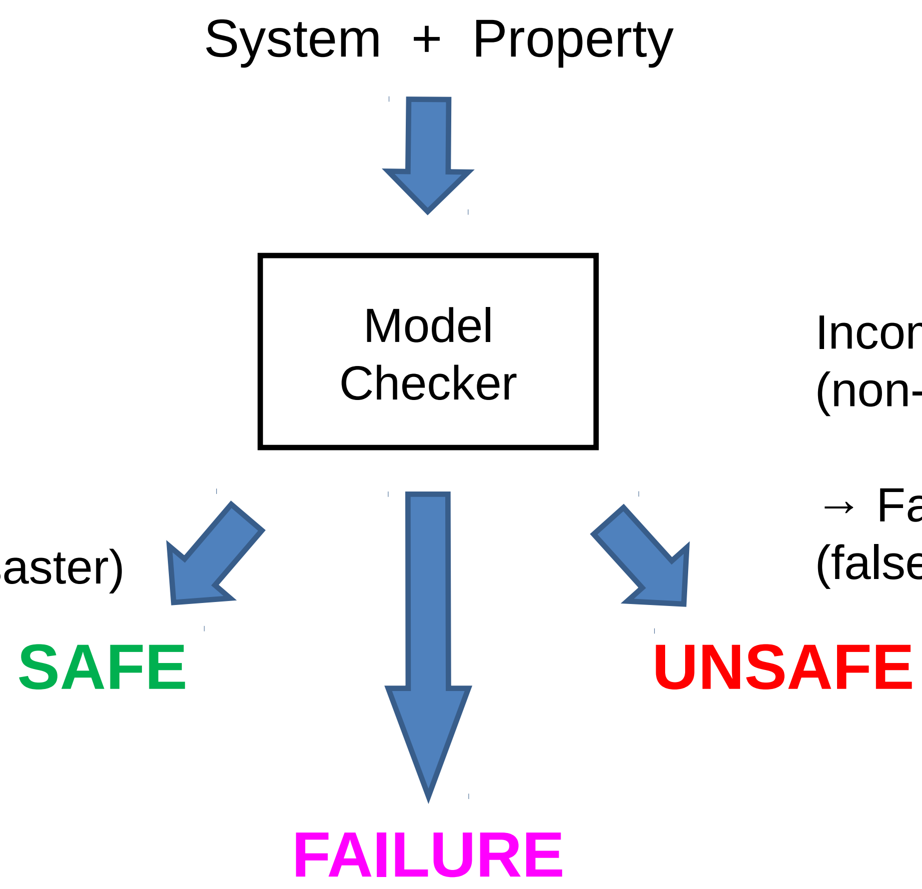
Model Checking



Classic Model Checking



Classic Model Checking



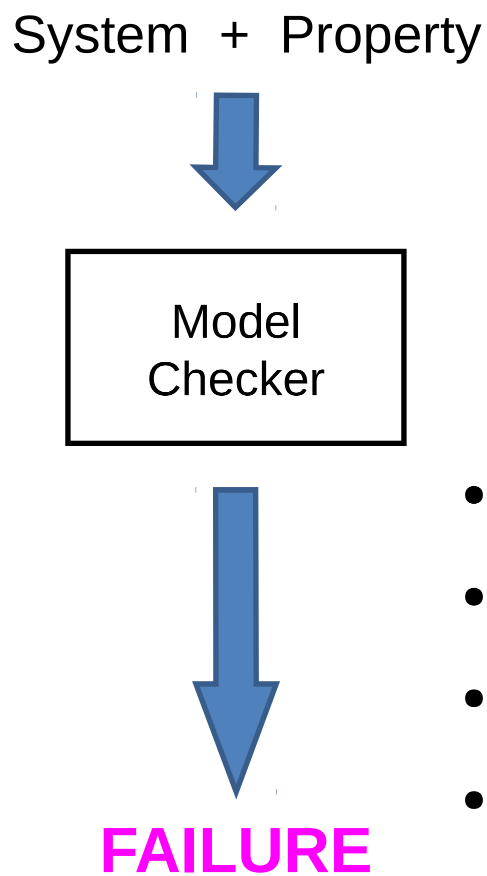
Unsound
(overflows)

→ False negatives
(missed bugs = disaster)

Incomplete
(non-linear operations)

→ False positives
(false alarms = noise)

Classic Model Checking



- Timeout
- Out of memory
- Crash of component
- Operand exception

Enormous amounts of resources **wasted**!

Conditional Model Checking

FSE 2012, joint work with Tom Henzinger,
Erkan Keremoglu, Philipp Wendler

Conditional Model Checking

System + Property

↓

Model Checker

↓

Ψ

("SAFE under Condition Ψ")

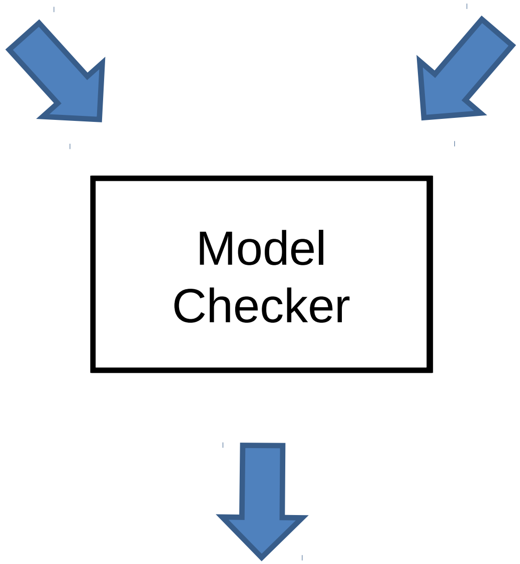
Examples:	- **Ψ =** true:	previous SAFE
	- **Ψ =** false:	previous UNSAFE
	- general:	condition for safety

Conditional Model Checking

System + Property

Condition Ψ_0

Directs the analysis
to parts to analyze



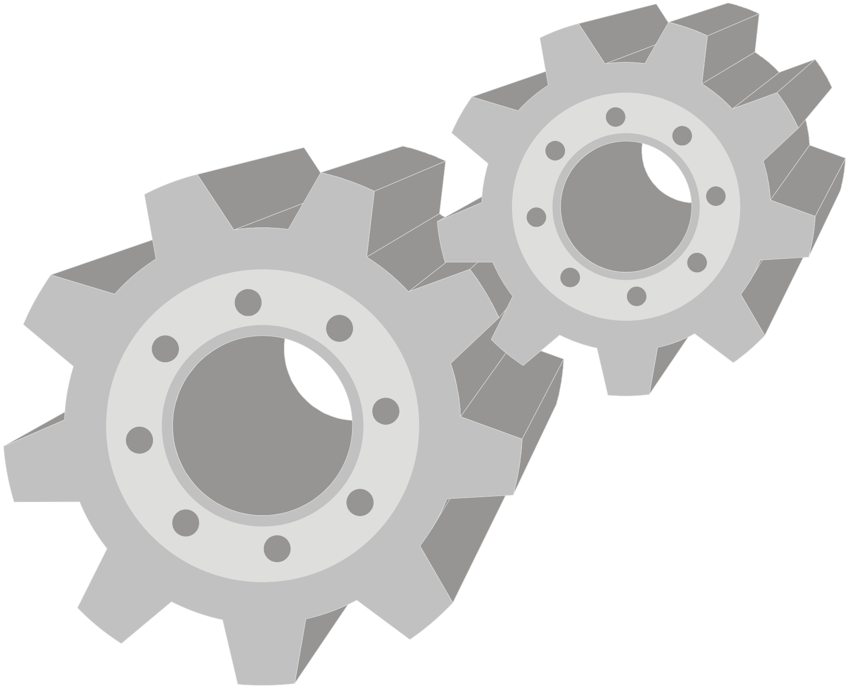
Ψ

(“SAFE under Condition Ψ”)

Examples:

- **Ψ =** true: previous SAFE
- **Ψ =** false: previous UNSAFE
- general: condition for safety

Applications of Conditional Model Checking



Back to Our Example

```
void main() {
  if (nondet_int()) {
    int i;
    for (i = nondet_int(); i < 1000000; i++) {
      // ...
    }
    assert(i >= 1000000);

  } else {
    int x = 5;
    int y = 6;
    int r = x * y;
    assert(r >= x);
  }
}
```

Back to Our Example

To show:

$$M \models \Phi$$

In this case:

$$\Phi = \Phi_1 \;\&\; \Phi_2$$

with Φ_1 = “loop is correct”

and Φ_2 = “multiplication is correct”

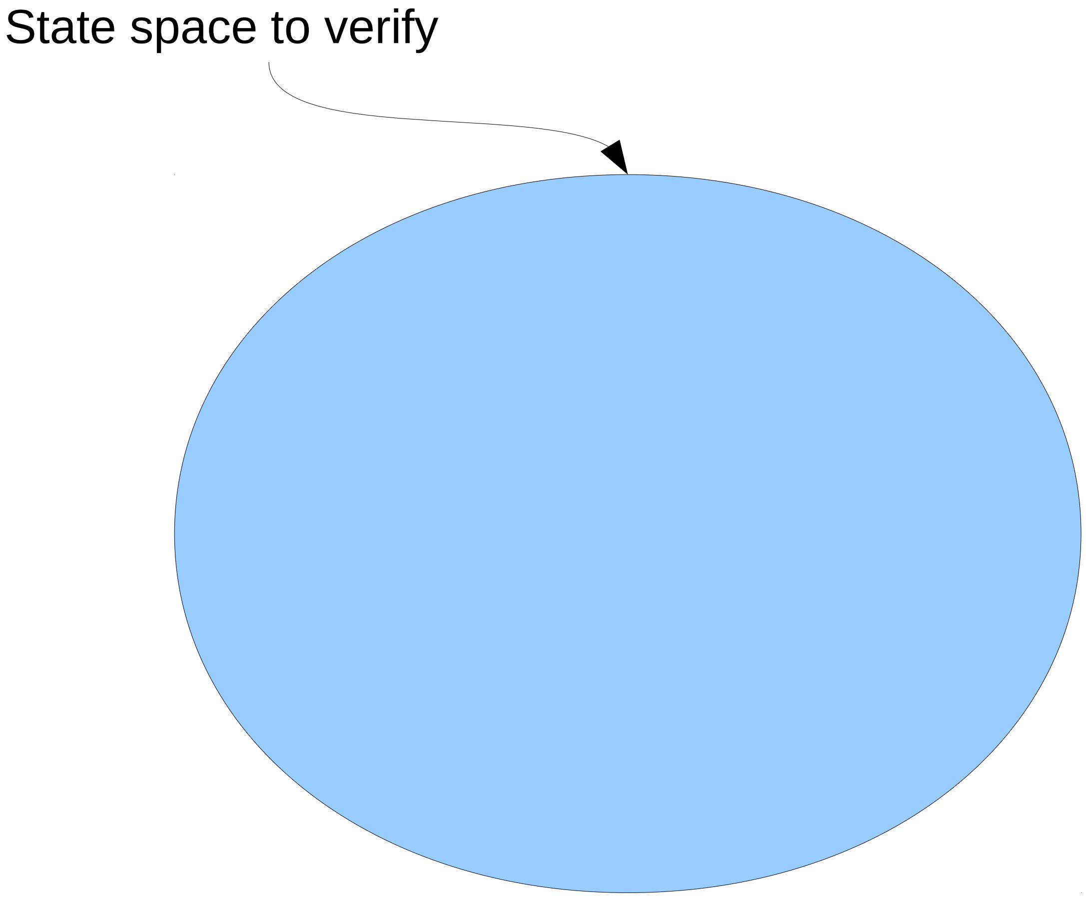
Idea: Decompose!

- Verify Φ_1 (“loop is correct”)
  → use predicate analysis
- Verify Φ_2 (“multiplication is correct”)
  → use explicit-state analysis
- Final result: Φ verified

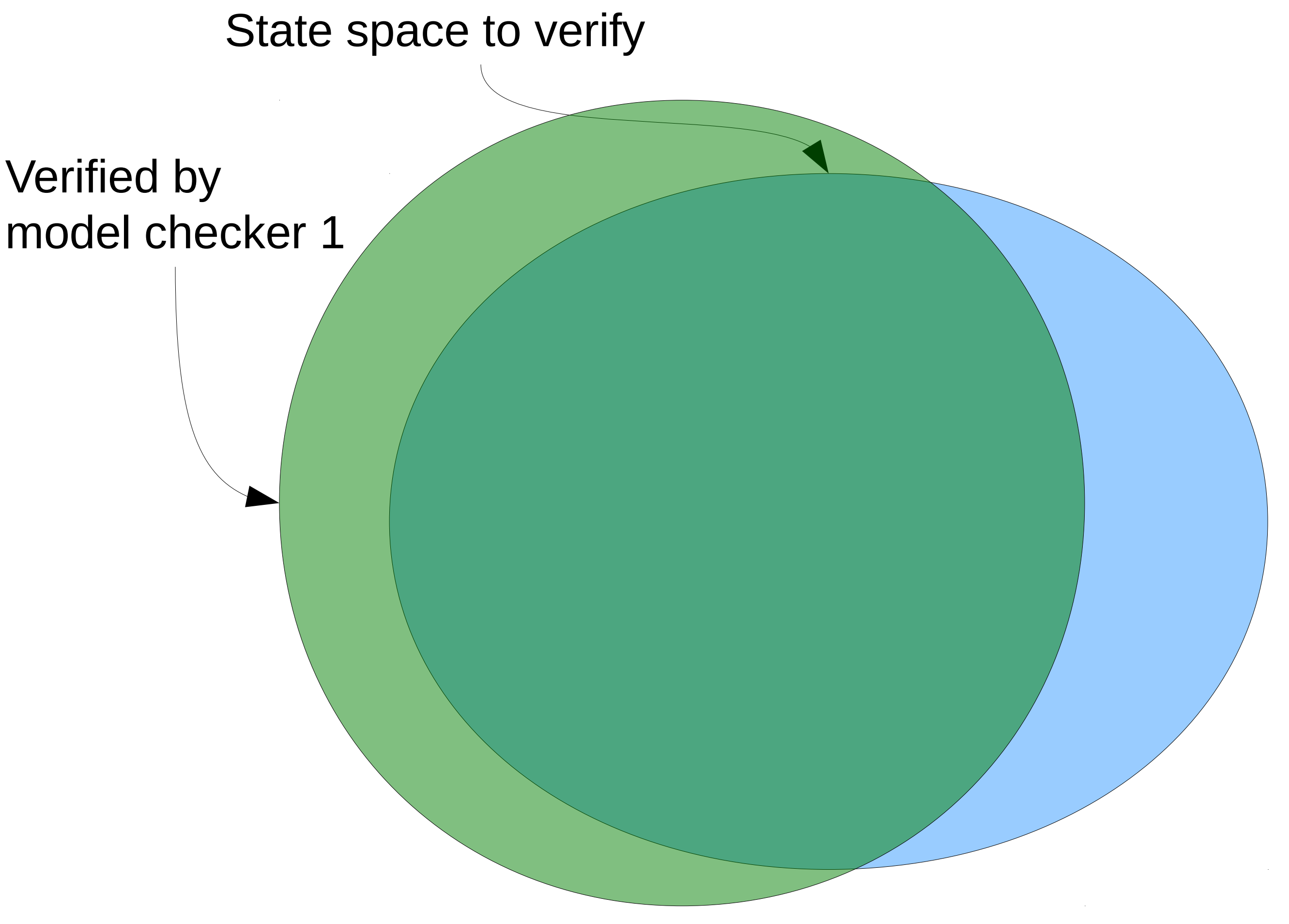
Using CMC with Input Conditions

- Tell model checker what to verify
- In our example:
  - For conditional model checker 1: verify Φ_1
  - For conditional model checker 2: verify Φ_2
  - Full verification possible

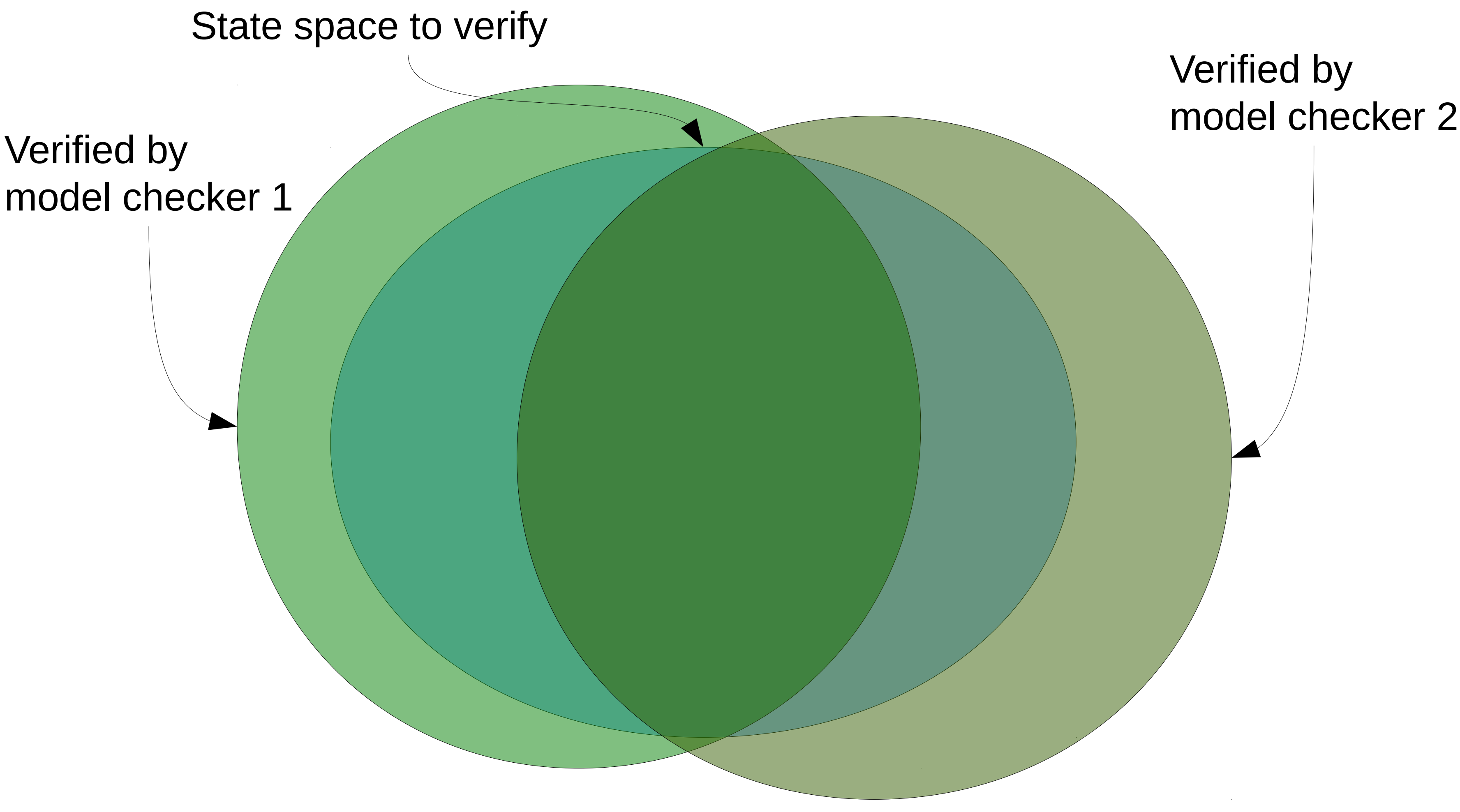
More General:



More General:



More General:



Further Input Conditions

- Limit resources
  - Time
  - Memory
  - Model Checker will not crash, but terminate itself and give useful result
- Restrict the search
  - Loop bounds (a.k.a. “bounded model checking”)
  - Path length
  - Time spent on path
  - ...

Output Conditions

- Dump partial result if analysis didn't finish
  - Output cond. summarizes what could be verified
- Explicitly state assumptions used by MC
  - Example: “variable **x** does not overflow”
- Purpose:
  - Give information to the user
  - Verify condition with other methods (testing, manual proofs, …)
  - Comparison of checkers (weaker output condition is better)

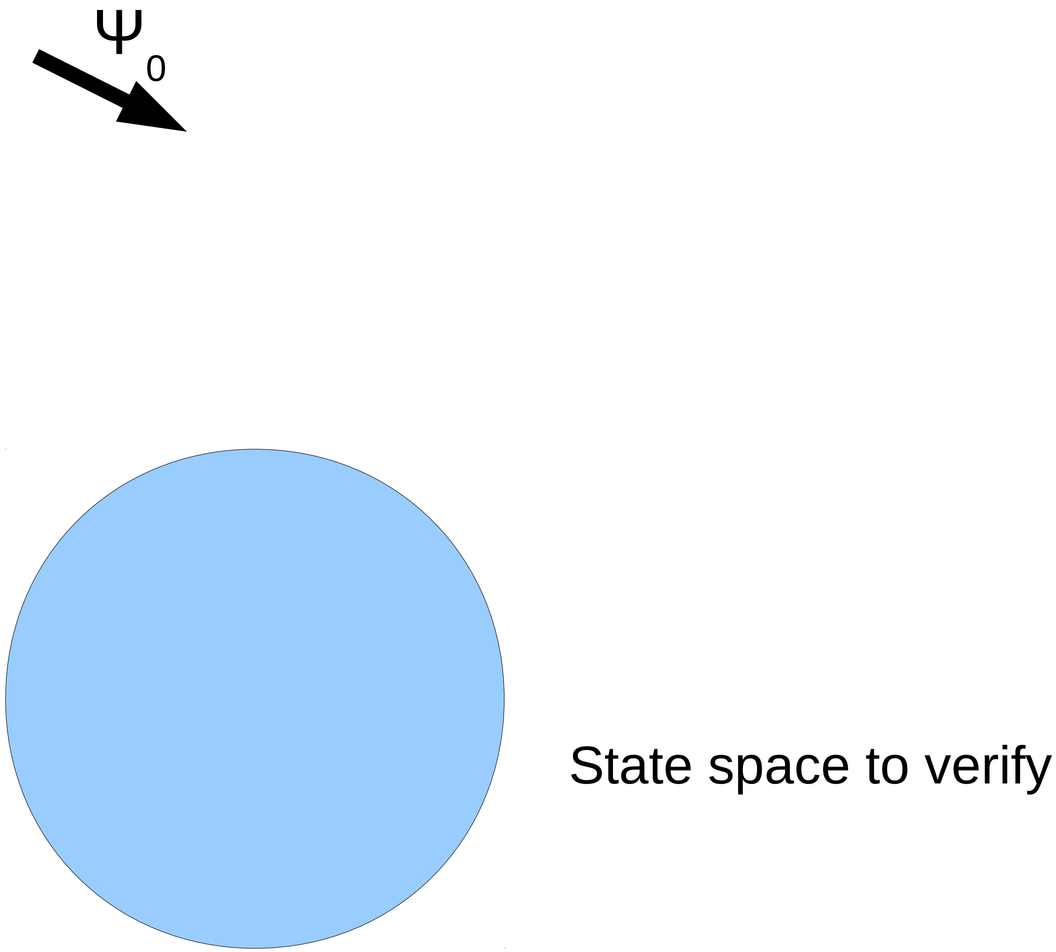
Sequential Composition

- In our example,
  we told the model checkers what to verify
- Now let them find out automatically!
- Conditional model checker 1 verifies what it can verify
- Conditional model checker 2 verifies remaining parts

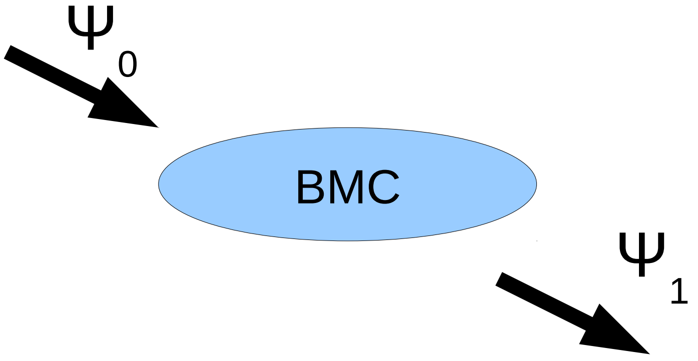
Sequential Composition

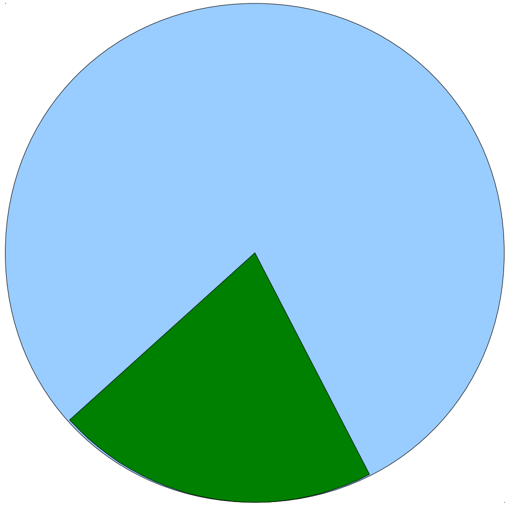
- Use input condition to limit resource usage of first analysis
- Use output condition
  as input condition for next model checker
- Iterate until finished (or run out of tools)

Sequential Composition



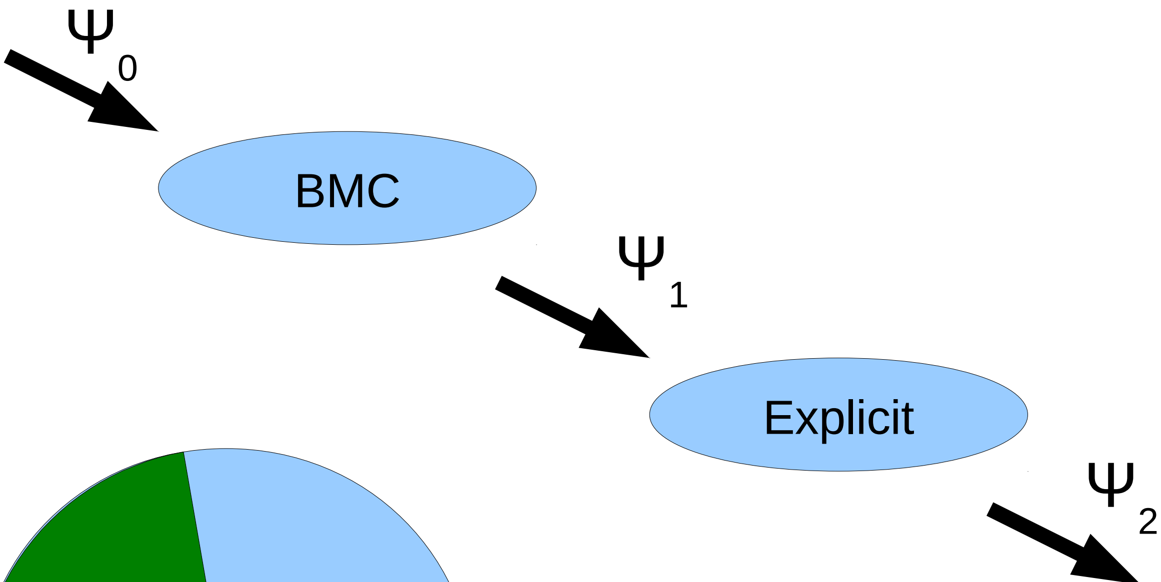
Sequential Composition

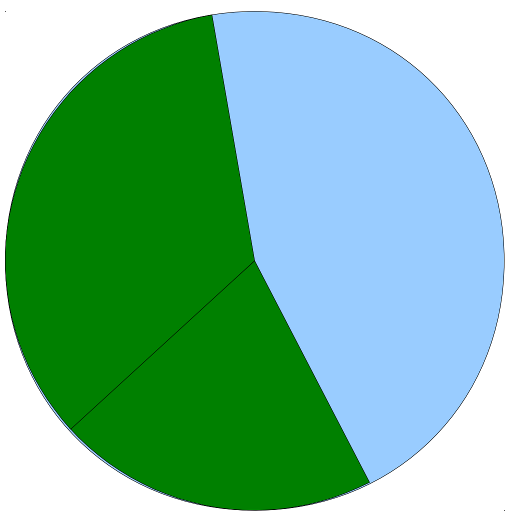




State space
partially verified

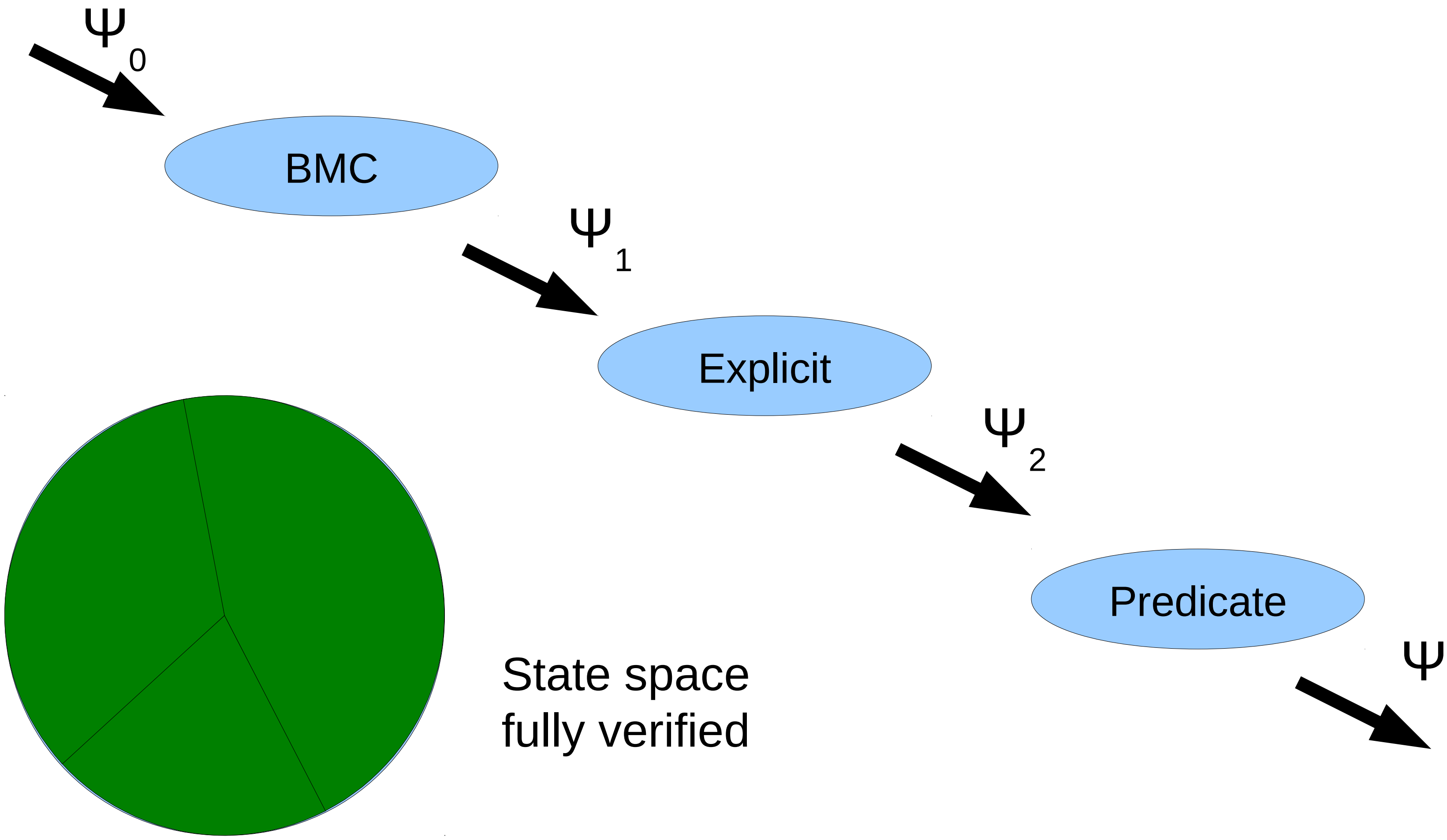
Sequential Composition





State space
partially verified

Sequential Composition



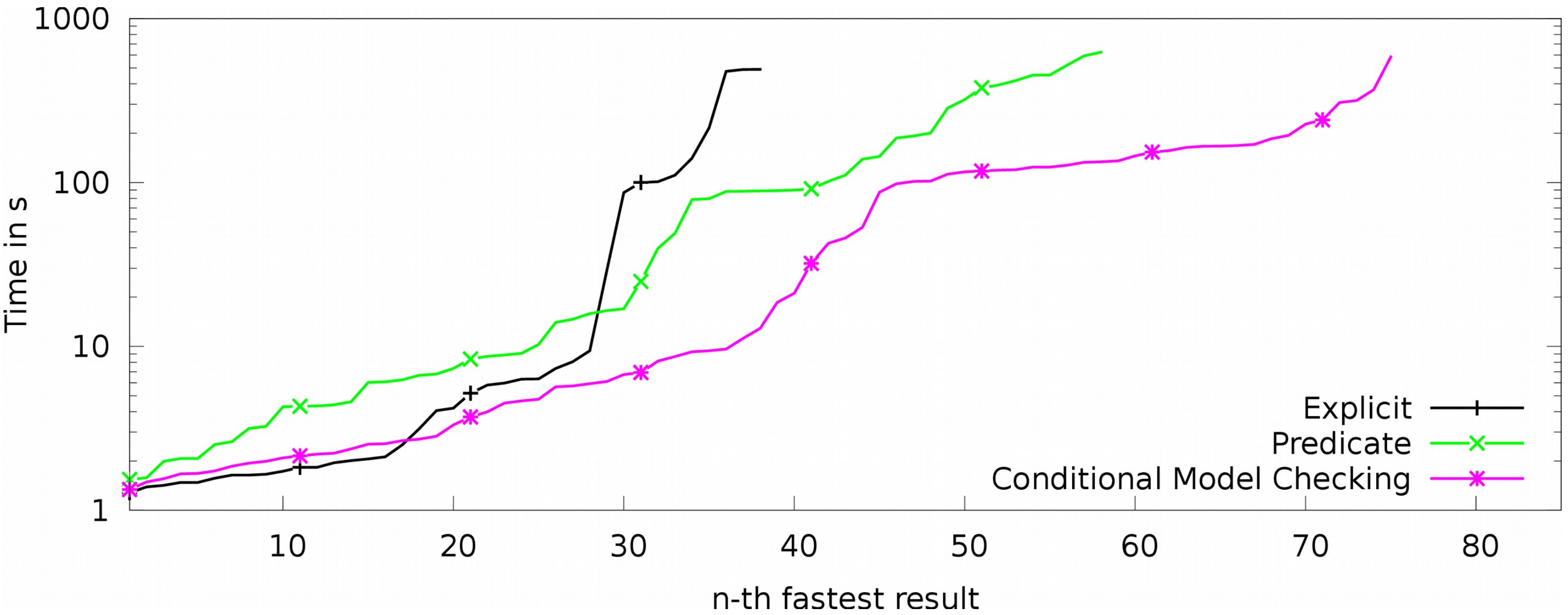
Experiment: Sequential Composition

- Implemented Conditional Model Checking in CPAchecker
- 85 C programs based on “hard” programs of Software Verification Competition 2012
- 15 min time, 15 GB RAM

Experiment: Sequential Composition

- A: Explicit-value analysis
- B: Predicate analysis
- C: Conditional model checking
  - First: explicit-value analysis
    with input condition: time limit = 100s
  - Second: predicate analysis
    with output condition of first analysis
    as input condition

Experiment: Sequential Composition

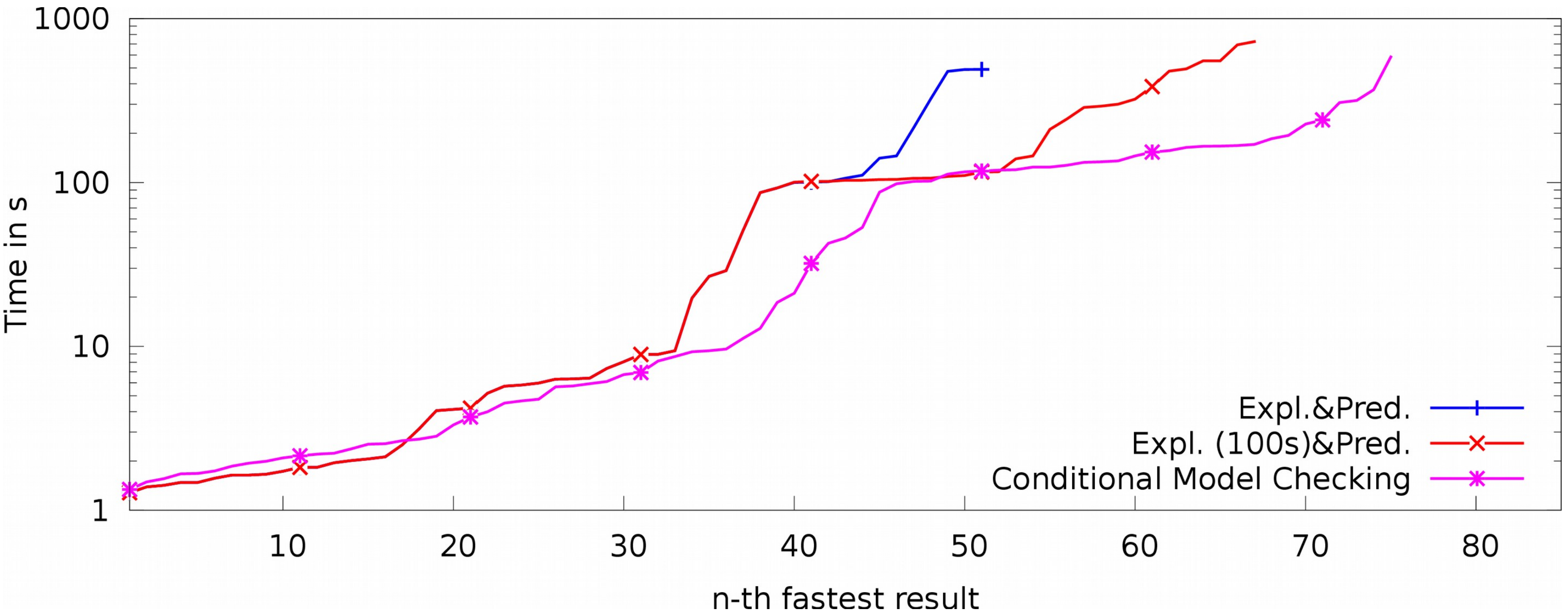


➔ Sequential composition solves more problems and is faster

Experiment: Sequential Composition

- A: Explicit-value analysis ; predicate analysis
- B: Explicit-value analysis ; predicate analysis
  - Input condition for first analysis:
    time limit = 100s
- C: Conditional model checking
  - First: explicit-value analysis
    with input condition: time limit = 100s
  - Second: predicate analysis
    with output condition of first analysis
    as input condition

Experiment: Sequential Composition



➔ Using conditional model checking for sequential composition is better

Summary Part 1

Conditional Model Checking:

- Terminates with useful results
  (no crashes)
- Enables partial / compositional verification
- Effective sequential composition
  (solve harder problems)
- Unified view on existing approaches

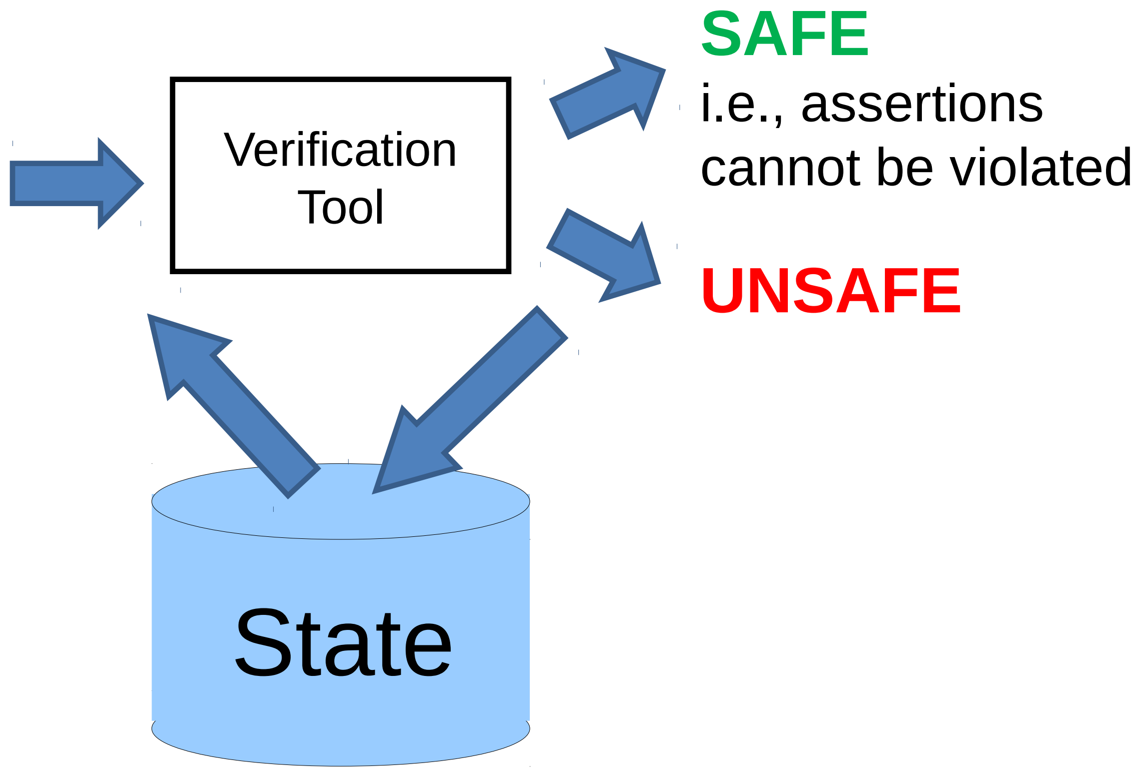
Stateful Verification

C program

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

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

Specification



Towards Reusing Information

- Context: CEGAR-based verification
- Abstract model has to be constructed every time a verification task is started
  → Refinements of Precision
  → Reconstruction and Pruning of ARG

Precision Reuse for Efficient Regression Verification

(Published in Proc. ESEC/FSE 2013, ACM.)

Dirk Beyer

Stefan Löwe

Evgeny Novikov

Andreas Stahlbauer

Philipp Wendler

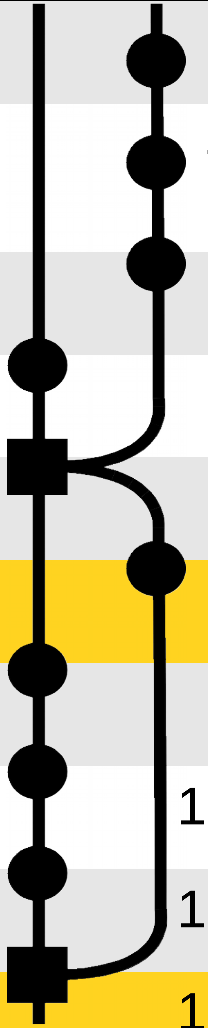
UNIVERSITÄT PASSAU

ISP RAS

UNIVERSITÄT PASSAU


Linux Driver Verification

Example



Revision	Commit Message	Safe?
3	Implement button detection support	✗
4	Free MICDET IRQ on error during probe	✗
5	fix typos in extcon-arizona	✗
6	Use bypass mode for MICVDD	✗
7	Merge tag ’driver-core-3.6’ of ...	✗
8	unlock mutex on error path in ...	✓
9	remove use of devexit	✗
10	remove use of devinit	✗
11	remove use of devexit p	✗
12	Merge tag ’pull req 20121122’ of ...	✓

High Resource Consumption!

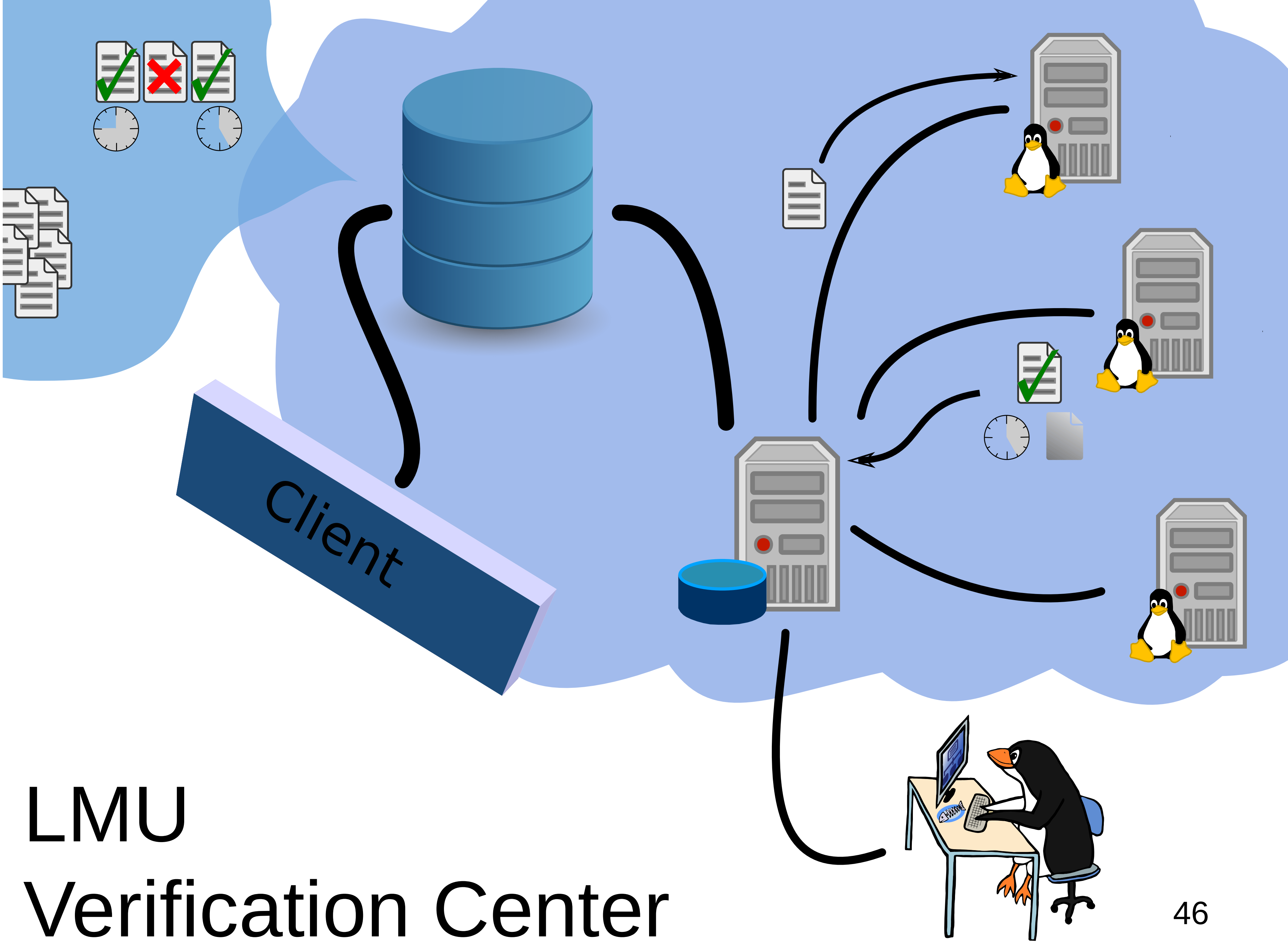
Software Verification is expensive

Verifying all **safety properties** for all **revisions** of a software ...

```
         500 drivers
        * 60 properties
        *  2 before/after
=  60 000 verification tasks
        * 10 seconds/verification task
= 600 000 seconds
```

≈ 7 days ☹

… is **really expensive**



LMU
Verification Center

Reuse of Verification Results

Drawbacks of existing approaches

- Too large: space on disk, time for loading
- Too sensitive to changes between revisions
- Too complex: modification of the verification algorithm

➡ Reuse the “precision”

Precision π

Defines the level of abstraction within an abstract domain:

Information that an abstraction-based analysis has to track to prove a property.

Examples for Precision

- Predicate Analysis

  $\pi = \{a > 0, k == 1 \wedge e == 0\}$

  Set of predicates used to compute boolean abstractions

- Explicit-Value Analysis

  $\pi = \{a, k, e\}$

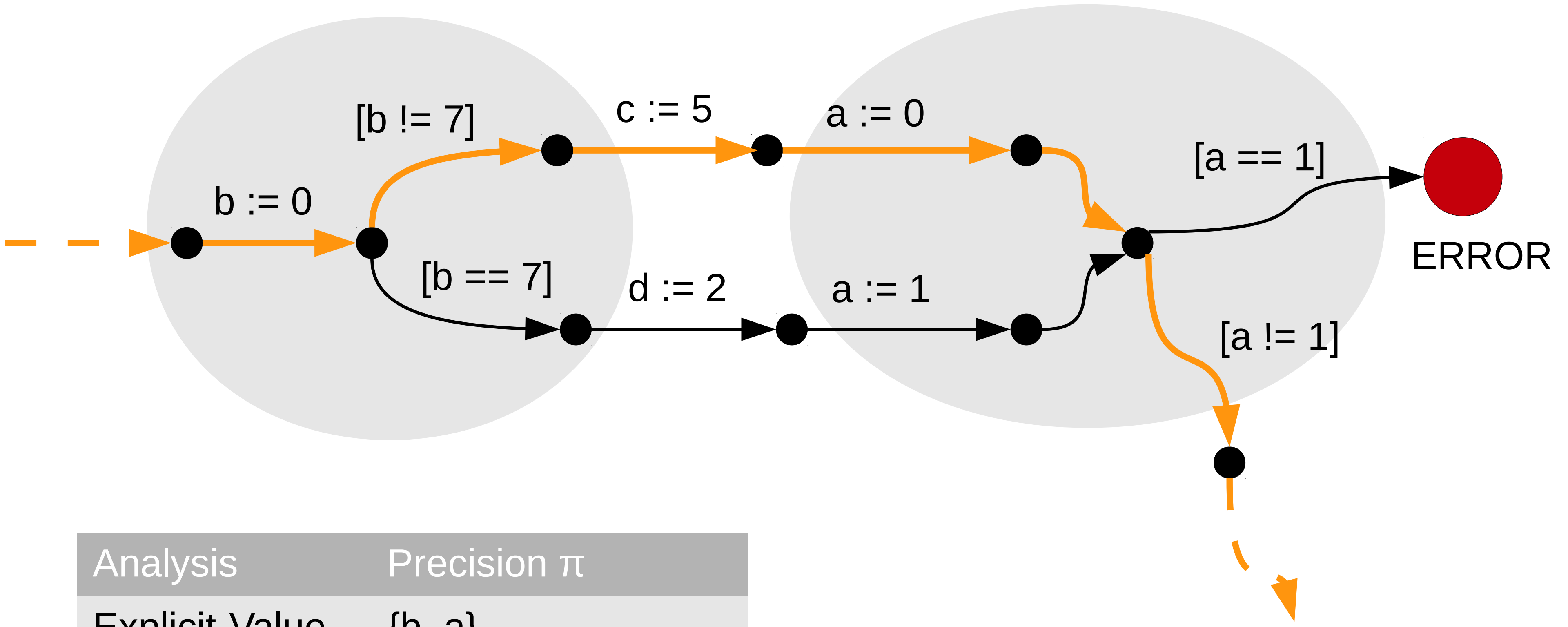
  Set of variables for which the explicit value has to be tracked

- Shape Analysis

  $\pi = \{p1, p2\}$

  Set of pointer variables to track

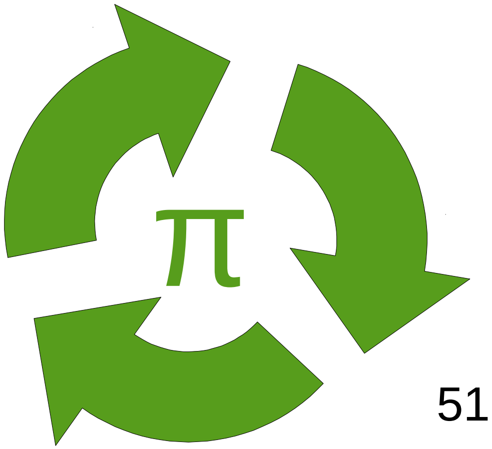
Example



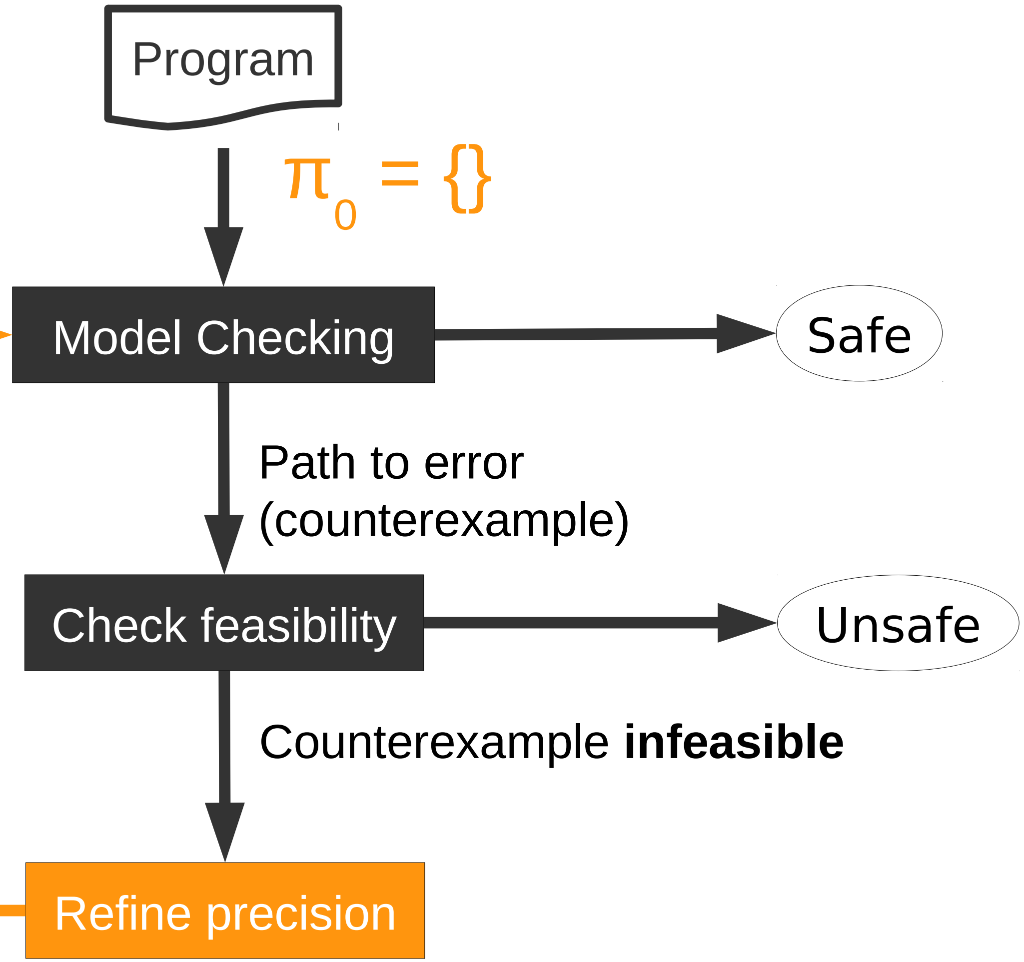
Analysis	Precision π
Explicit-Value	{b, a}
Predicate	{b == 7, a == 1}

Advantages of Reusing Precisions

- No modification of the verification algorithm
- Easy to extract from model checkers
- Small memory footprint
- Low sensitivity to changes in the input programs

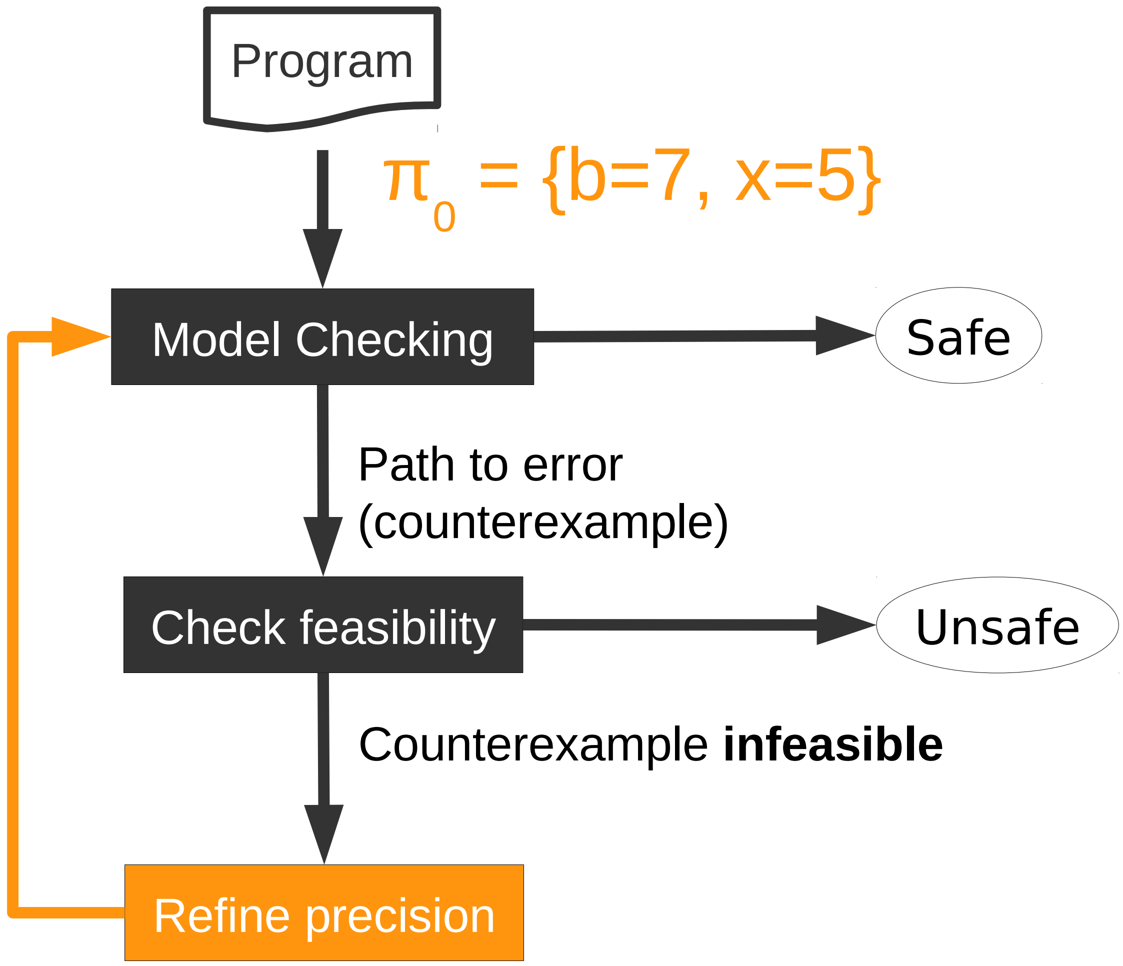


CEGAR



$\pi_{i+1} = \pi_i \cup \text{Interpolants}_{i+1}$

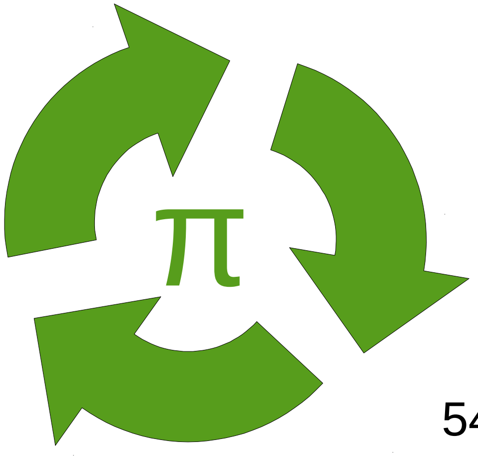
CEGAR + Reuse



$\pi_{i+1} = \pi_i \cup \text{Interpolants}_{i+1}$

Advantages of Reusing Precisions

- **No modification** of the verification algorithm
- **Easy to extract** from model checkers
- **Small** memory footprint
- **Low sensitivity** to changes in the input programs

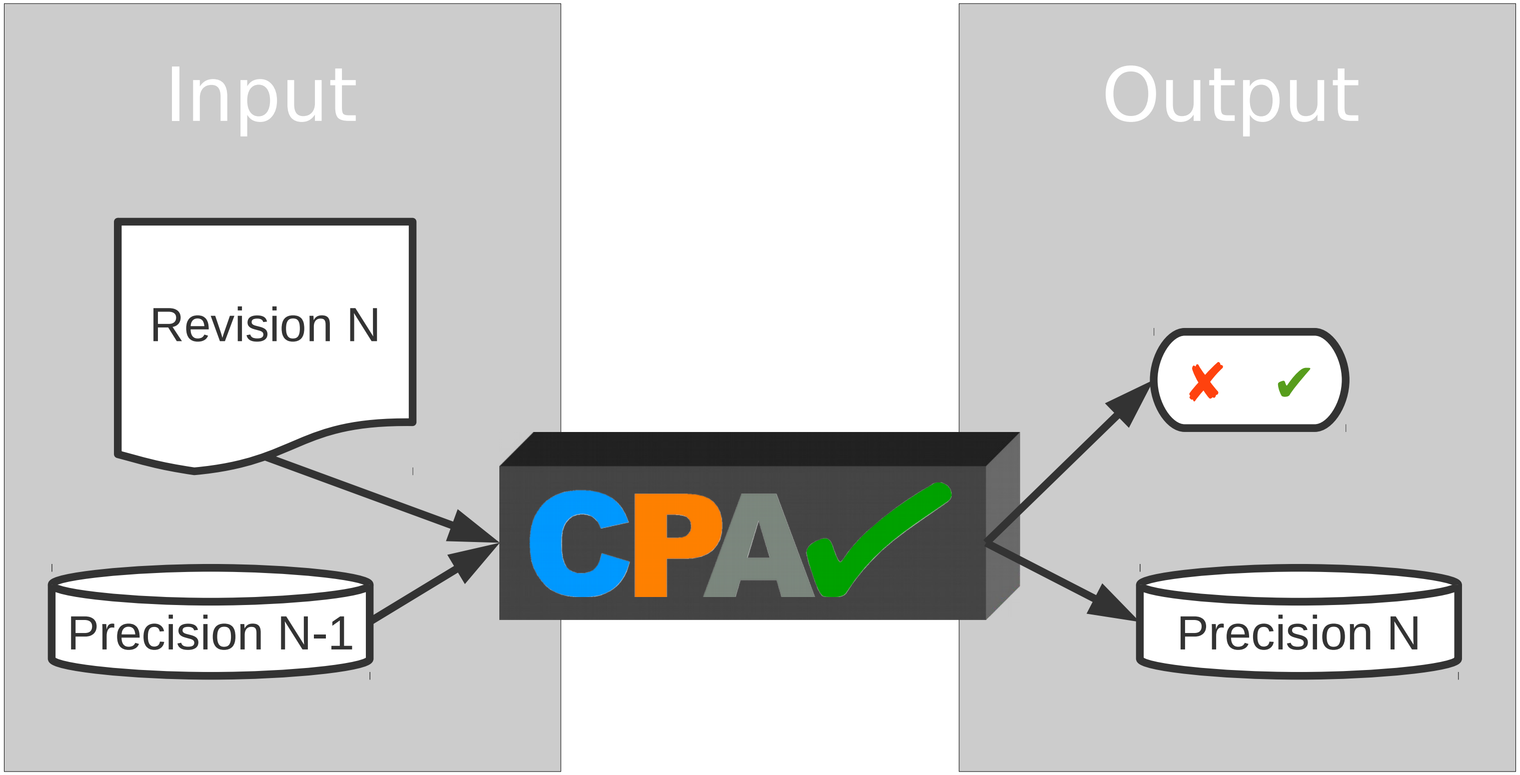


Implementation

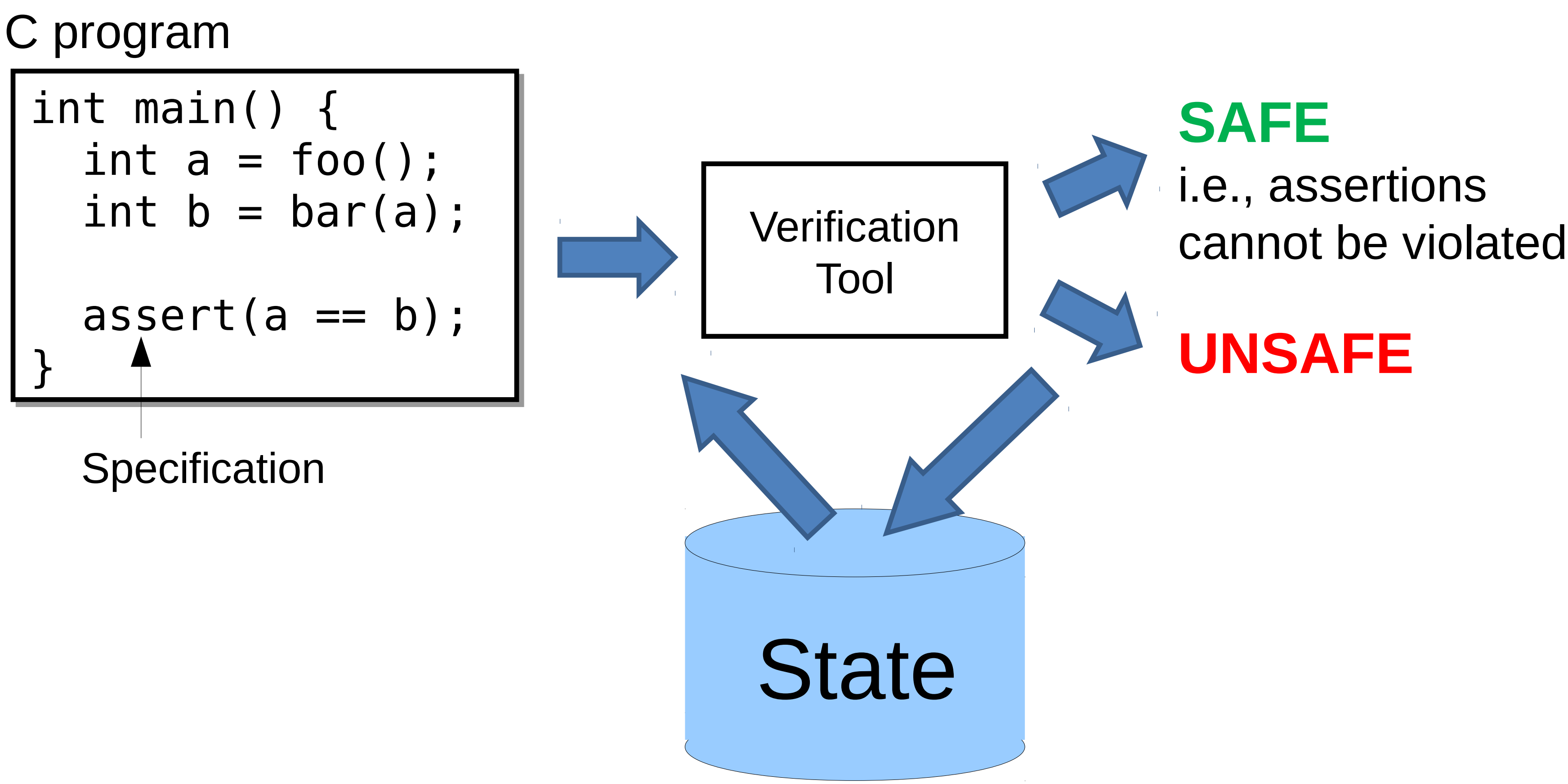
http://cpachecker.sosy-lab.org

- Implemented in CPAchecker
  - Predicate Analysis
  - Explicit-State Analysis
- Common to both analyses:
  - Lazy abstraction
  - CEGAR
  - Construct an abstract reachability graph

Workflow



Stateful Verification



Storing Precisions

Explicit-State Analysis

```
*:
lock

main f:
x
```

Predicate Analysis

```
(declare-fun |lock|() Real)
(declare-fun |x|() Real)
(define-fun t1() Bool (= |lock| 0))
(define-fun t2() Bool (<= |x| 1))

*:
(assert t1)

main f:
(assert t2)
```

Really simple! Dump the precision

Benchmark Suite

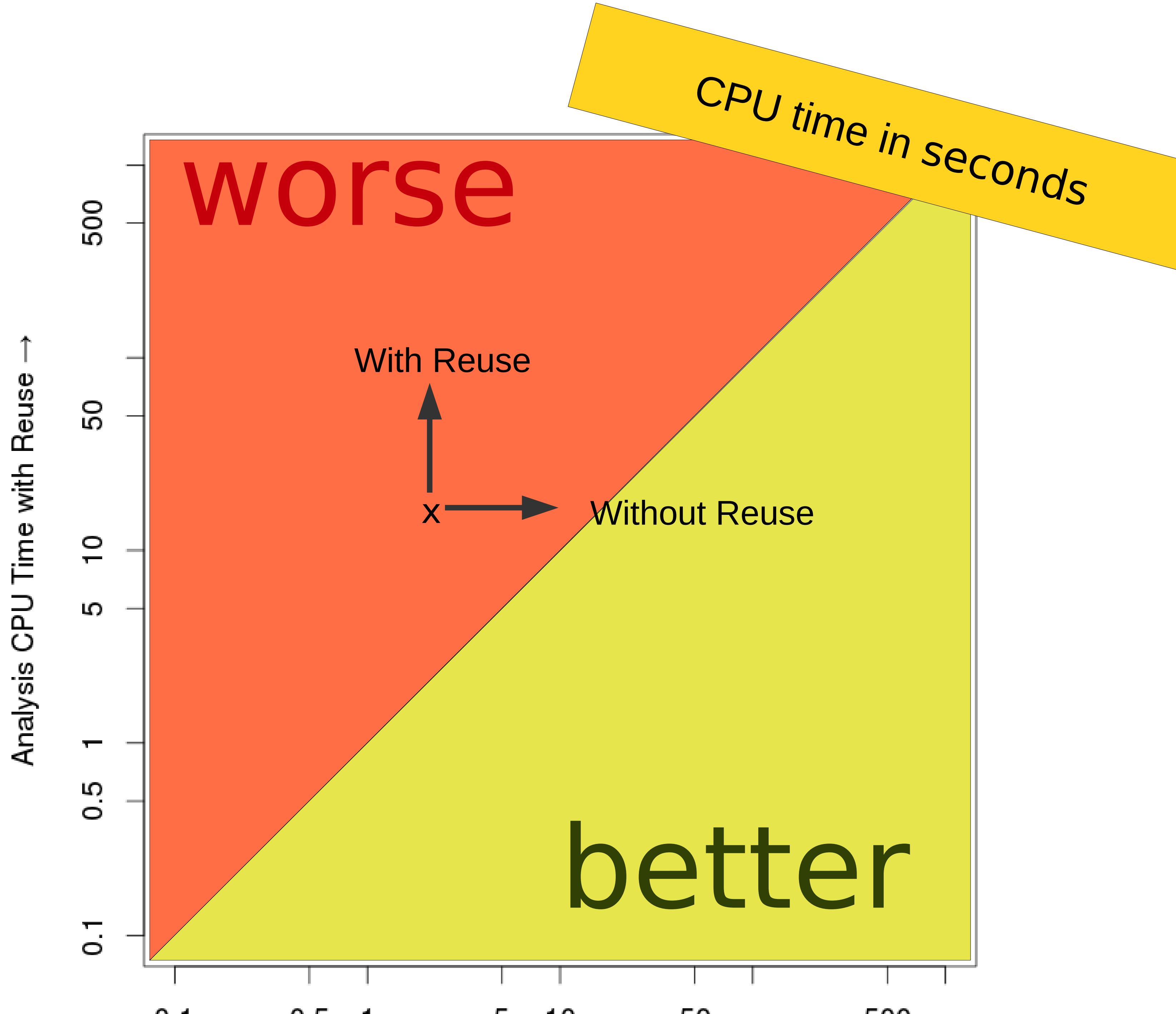
- Derived from industrial code (Linux kernel)
  - 4193 verification problems
  - 62 Linux device drivers
  - 1119 revisions
    spanning more than 5 years of development
- Publicly available

http://sosy-lab.org/~dbeyer/cpa-reuse/

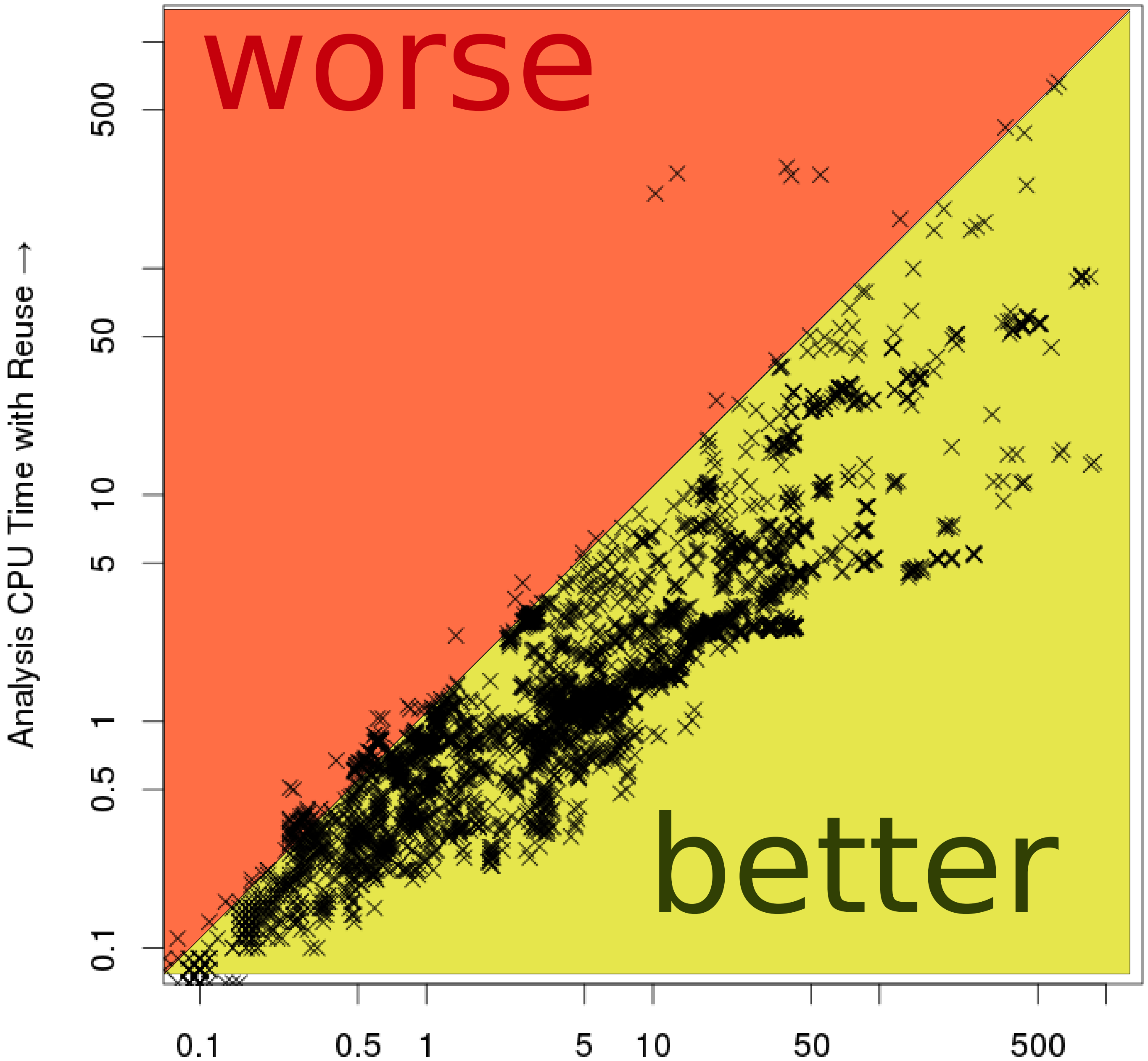
Benchmark Setup

- Processor: Intel i7 3.4 GHz Quad Core
- Time limit: 15 minutes
- Memory limit: 15 GB

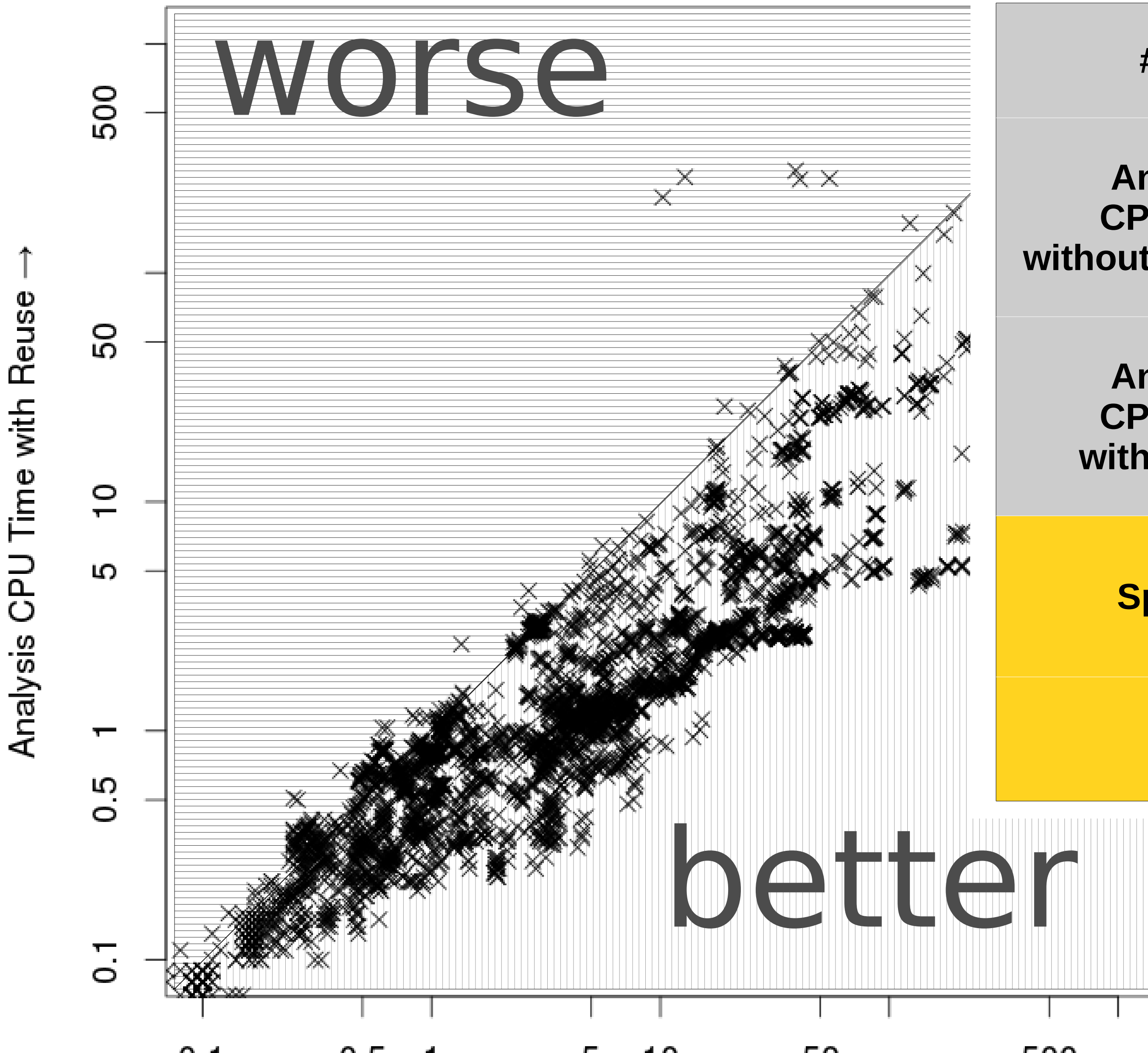
= Setup of the Intl. Competition on Software Verification


CPU time in seconds
worse
better
With Reuse
Without Reuse
x
Analysis CPU Time with Reuse →
500
50
10
5
1
0.5
0.1

Results for Predicate Analysis



Results for Predicate Analysis



# Tasks	4 193
Analysis CPU Time without Reuse	83 000
Analysis CPU Time with Reuse	23 000
Speedup	**4.3**
Solved	4 048 **+ 30**

Summary – Part 2

Precision reuse has a significant positive effect!

- Drastically improves performance
  → Reduces the number of refinements
- More problems can be solved
- Low sensitivity to changes in the program code

Reusing Witnesses

- Learn from previous proofs
- If you know a previous error path,

  → check this first, try to “re-play”
- If you know a previous proof,

  → try to “re-validate”, watch for changes

Correctness Witnesses: Exchanging Verification Results between Verifiers

Dirk Beyer, Matthias Dangl,
Daniel Dietsch, and Matthias Heizmann

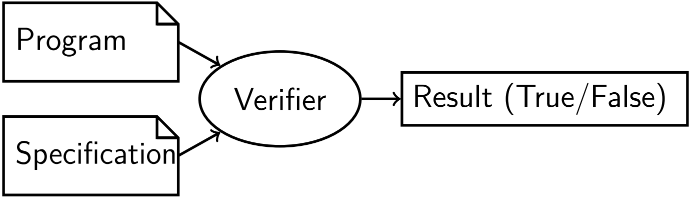



UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

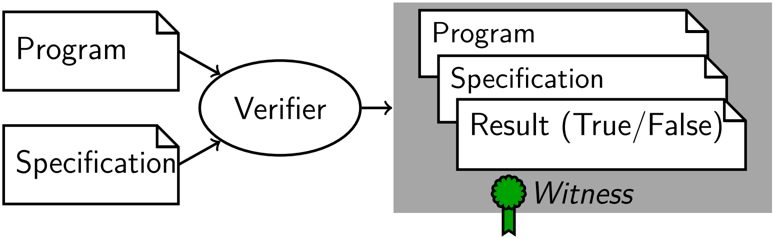
LMU
LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

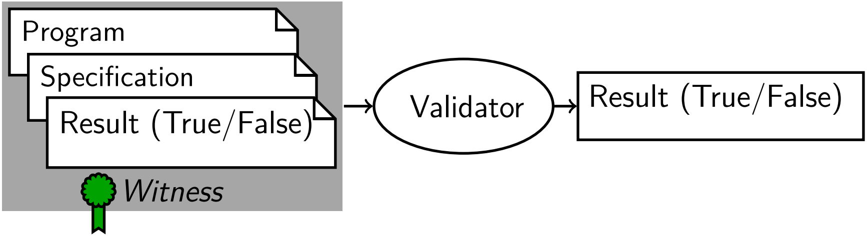
Software Verification



Software Verification with Witnesses

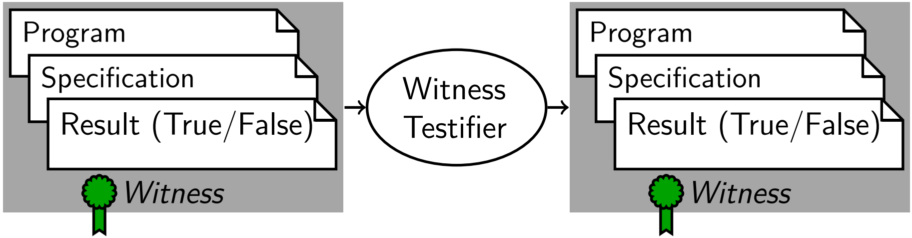


Witness Validation



- Validate untrusted results
- Easier than full verification

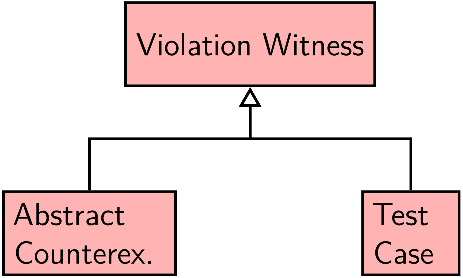
Stepwise Testification



Violation Witnesses

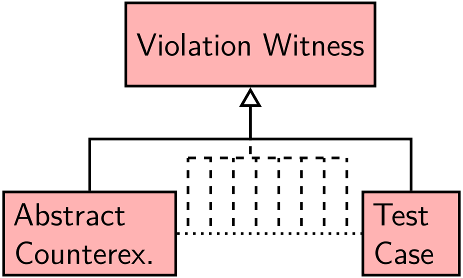
Violation Witness

Violation Witnesses

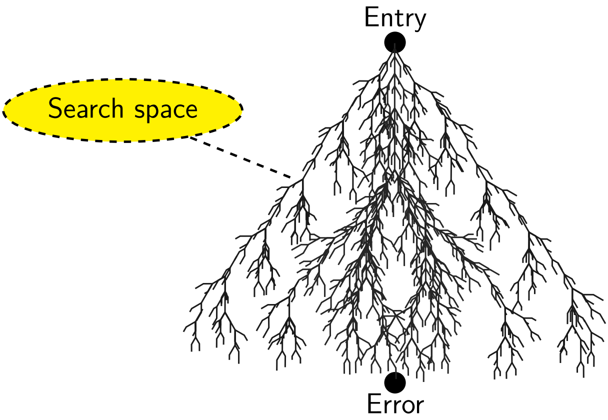


Violation Witnesses

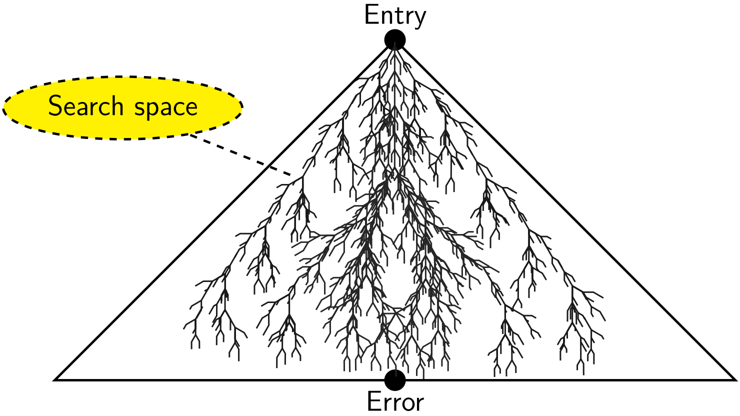
FSE'15



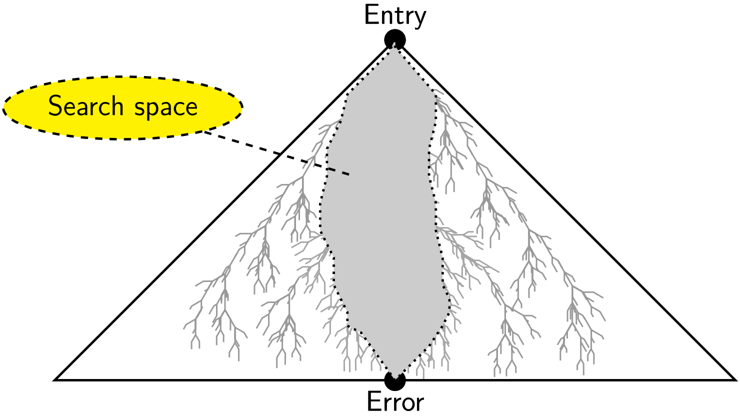
Search-Space Reduction for Stepwise Testification



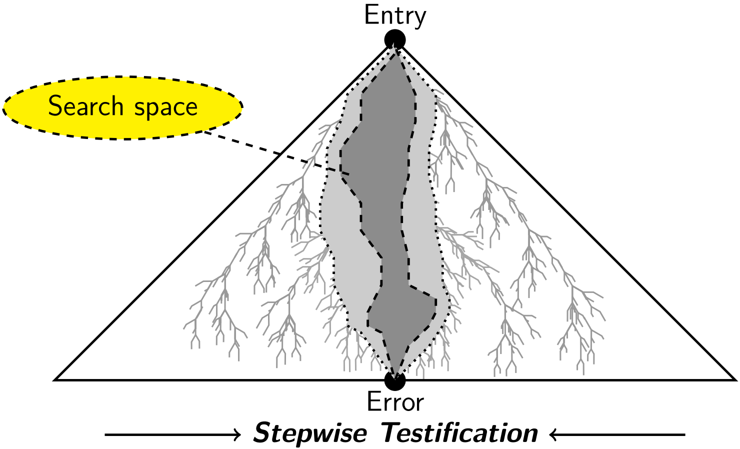
Search-Space Reduction for Stepwise Testification



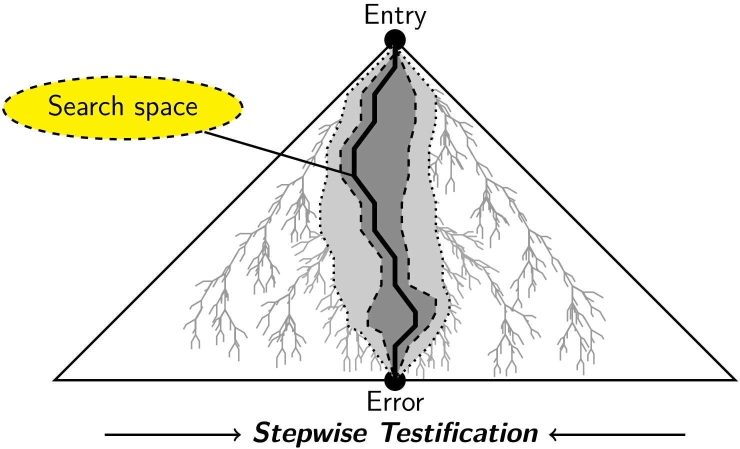
Search-Space Reduction for Stepwise Testification



Search-Space Reduction for Stepwise Testification



Search-Space Reduction for Stepwise Testification



Correctness: State of the Art

1. **Rarely any** additional information

Correctness: State of the Art

1. **Rarely any** additional information
2. **Not** human **readable**

Correctness: State of the Art

1. **Rarely any** additional information
2. **Not** human **readable**
3. **Not easily exchangeable** across tools

Open Problems

1. **Standardized way** to document verification results to enhance engineering processes **required**

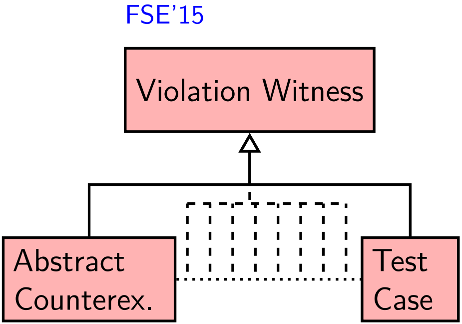
Open Problems

1. **Standardized way** to document verification results to enhance engineering processes **required**
2. **Difficult to establish trust** in results from an untrusted verifier

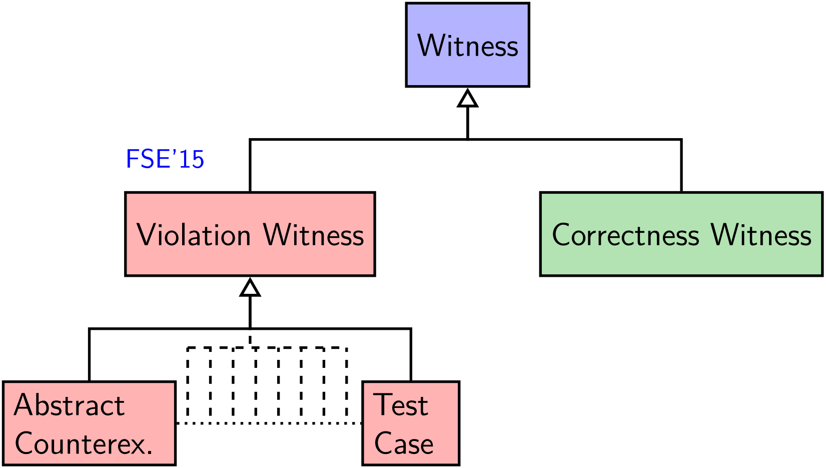
Open Problems

1. **Standardized way** to document verification results to enhance engineering processes **required**
2. **Difficult to establish trust** in results from an untrusted verifier
3. Potential for synergies between tools and techniques is **left unused**

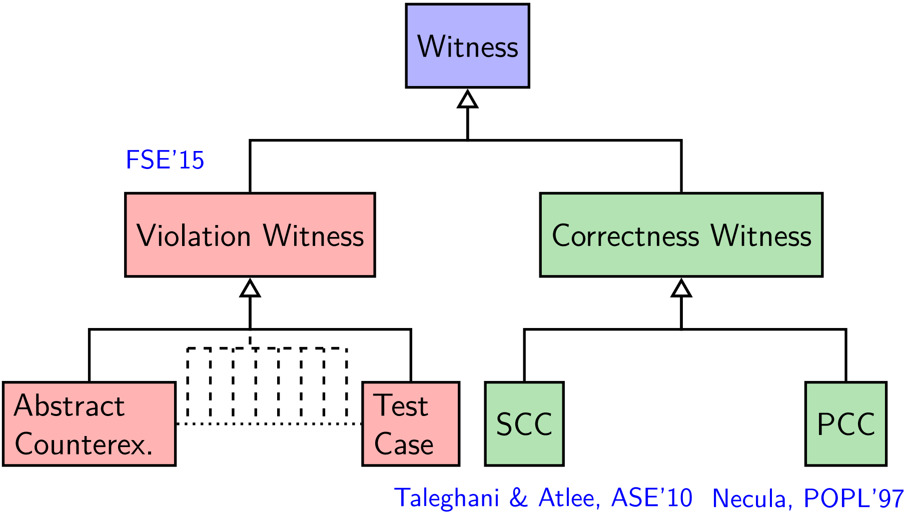
Verification Witnesses: Classification



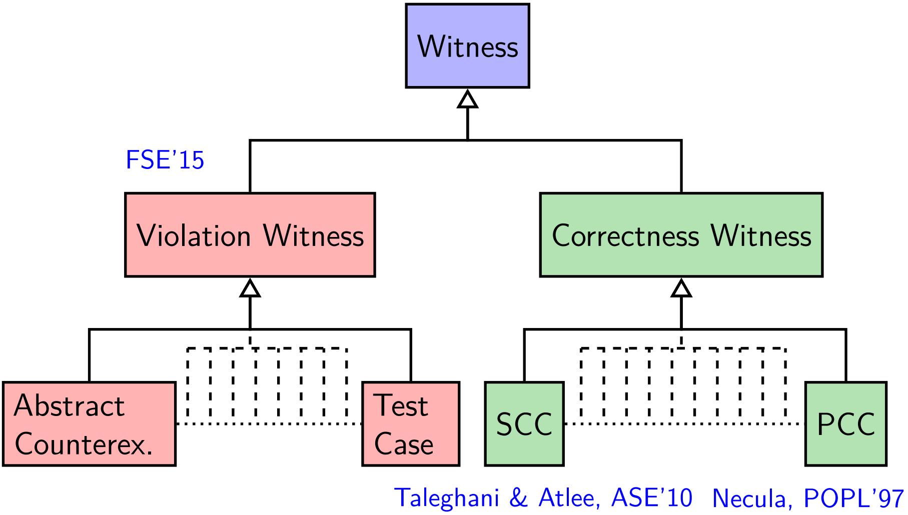
Verification Witnesses: Classification



Verification Witnesses: Classification



Verification Witnesses: Classification



Correctness Witnesses and Proof Certificates

- **Full proofs** seem nice, but in practice become **too large**
- Witnesses **support**, but do **not enforce** full proofs
- **Instead**, correctness witnesses may also represent **proof sketches**

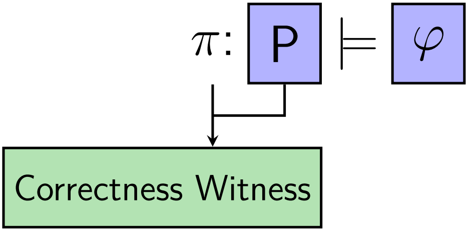
Correctness Witnesses

$$P \models \varphi$$

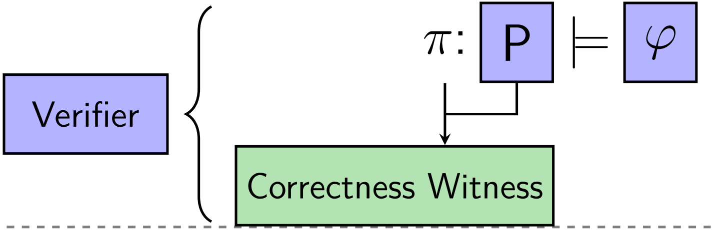
Correctness Witnesses

π: $\boxed{\mathsf{P}} \models \boxed{\varphi}$

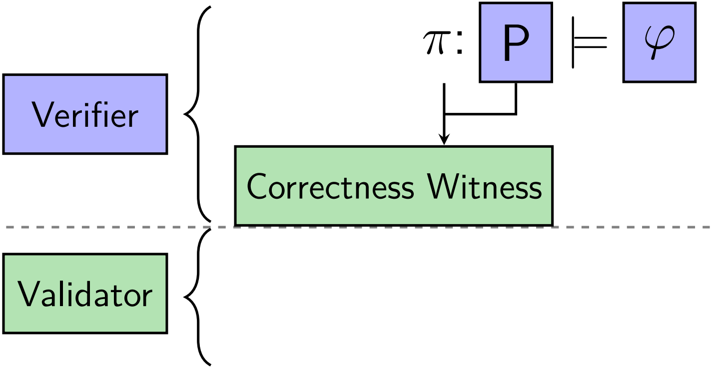
Correctness Witnesses



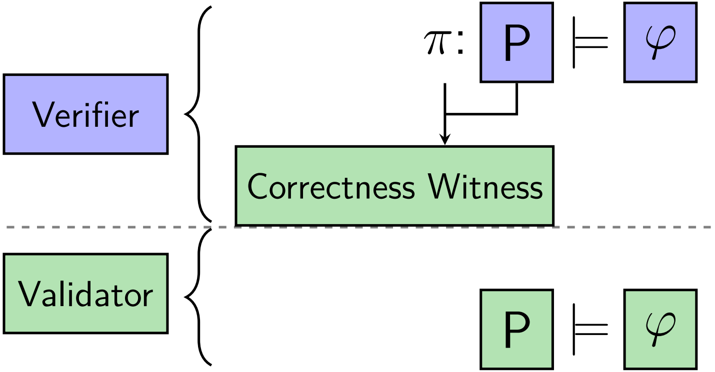
Correctness Witnesses



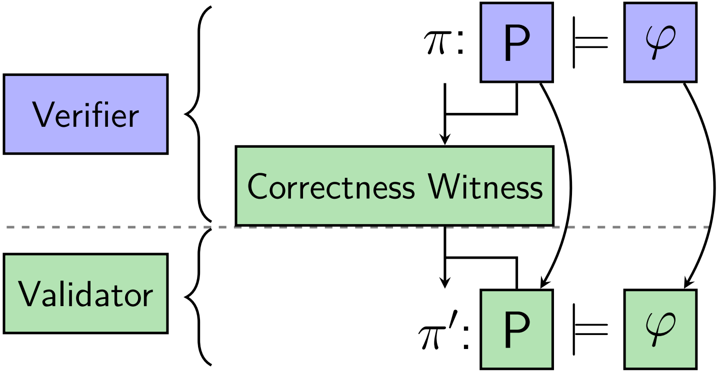
Correctness Witnesses



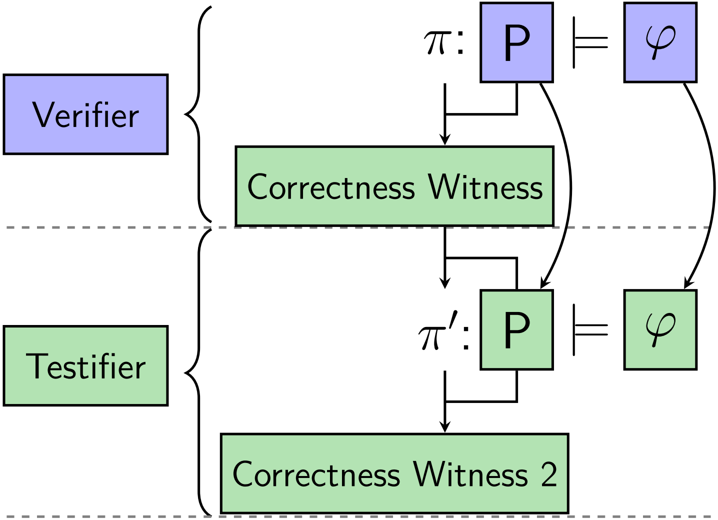
Correctness Witnesses



Correctness Witnesses



Correctness Witnesses



Witness Automata

- Express witness as **automaton**

Witness Automata

- Express witness as **automaton**
- Witness Validation **matches** the **witness** to the **program**

Witness Automata

- Express witness as **automaton**
- Witness Validation **matches** the **witness** to the **program**
- **Decoupled from** specific verification **techniques** and **implementations**

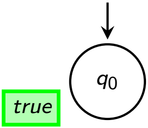
Witness Automata

- Express witness as **automaton**
- Witness Validation **matches** the **witness** to the **program**
- **Decoupled from** specific verification **techniques** and **implementations**
- One **common exchange format** for violation witnesses and correctness witnesses

Example: Inject Invariants

```
int main() {
  unsigned int x = nondet();
  unsigned int y = x;
  while (x < 1024) {
    x = x + 1;
    y = y + 1;
  }
  // Safety property
  assert(x == y);
  return 0;
}
```

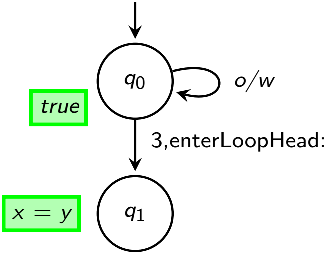
Example: Inject Invariants




```
int main() {
  unsigned int x = nondet();
  unsigned int y = x;
  while (x < 1024) {
    x = x + 1;
    y = y + 1;
  }
  // Safety property
  assert(x == y);
  return 0;
}
```

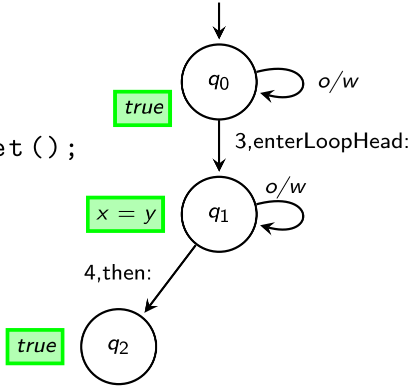
Example: Inject Invariants

```c
int main() {
  unsigned int x = nondet();
  unsigned int y = x;
  while (x < 1024) {
    x = x + 1;
    y = y + 1;
  }
  // Safety property
  assert(x == y);
  return 0;
}
```



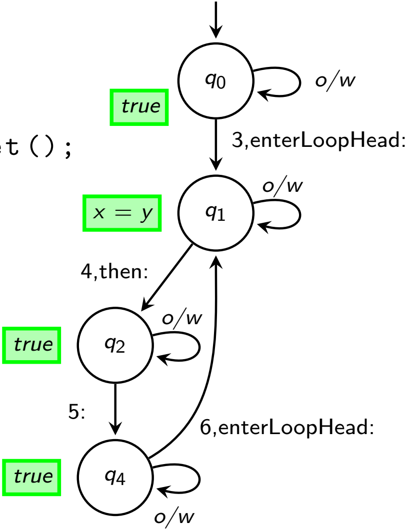
Example: Inject Invariants

```
1  int main() {
2    unsigned int x = nondet();
3    unsigned int y = x;
4    while (x < 1024) {
5      x = x + 1;
6      y = y + 1;
7    }
8    // Safety property
9    assert(x == y);
10   return 0;
11 }
```



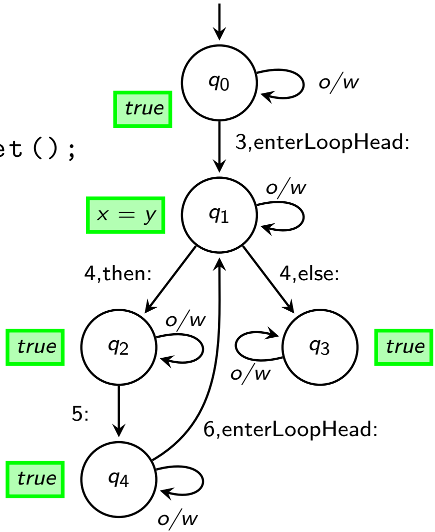
Example: Inject Invariants

```
int main() {
  unsigned int x = nondet();
  unsigned int y = x;
  while (x < 1024) {
    x = x + 1;
    y = y + 1;
  }
  // Safety property
  assert(x == y);
  return 0;
}
```



Example: Inject Invariants

```
int main() {
  unsigned int x = nondet();
  unsigned int y = x;
  while (x < 1024) {
    x = x + 1;
    y = y + 1;
  }
  // Safety property
  assert(x == y);
  return 0;
}
```



Experiments

Tasks and Limits

- Benchmark set: Competition on Software Verification 2016 (SV-COMP'16)
- CPU time: 15 min
- Memory: 15 GB

Configurations

- CPAchecker with *k*-induction
- UltimateAutomizer with automata-based trace abstraction

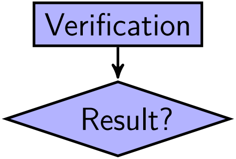
Producing and Consuming Witnesses
SV-COMP

Table 8: Confirmation rate of witnesses

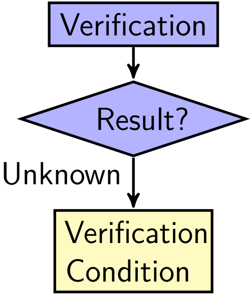
Result	TRUE			FALSE		
	Total	Confirmed	Unconfirmed	Total	Confirmed	Unconfirmed
UAUTOMIZER	3 558	3 481	77	1 173	1 121	52
SMACK	2 947	2 695	252	1 929	1 768	161
CPA-SEQ	3 357	3 078	279	2 342	2 315	27

Verifiable Witnesses. For SV-COMP, it is not sufficient to answer with just TRUE or FALSE: each answer must be accompanied by a verification witness. For correctness witnesses, an unconfirmed answer TRUE was still accepted, but was assigned only 1 point instead of 2 (cf. Table 2). All verifiers in categories that required witness validation support the common exchange format for violation and correctness witnesses. We used the two independently developed witness validators that are integrated in CPACHECKER and UAUTOMIZER [7,8].

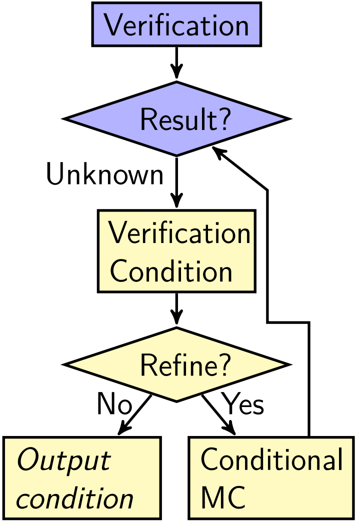
Stepwise Testification: Classification



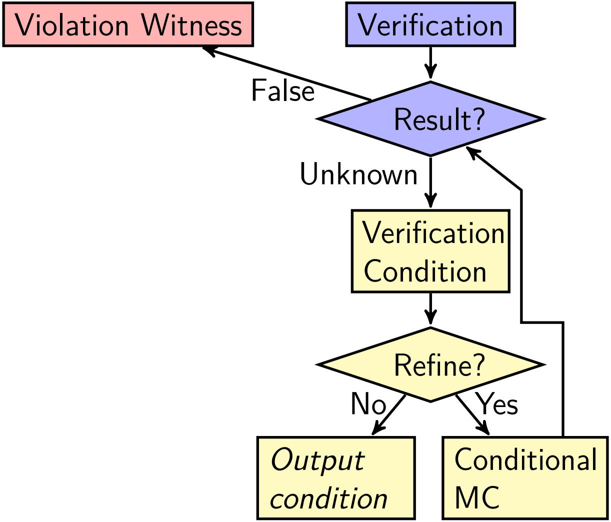
Stepwise Testification: Classification



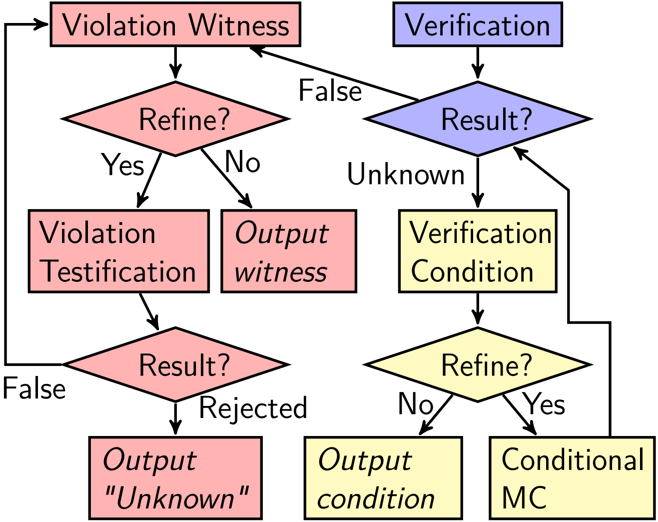
Stepwise Testification: Classification



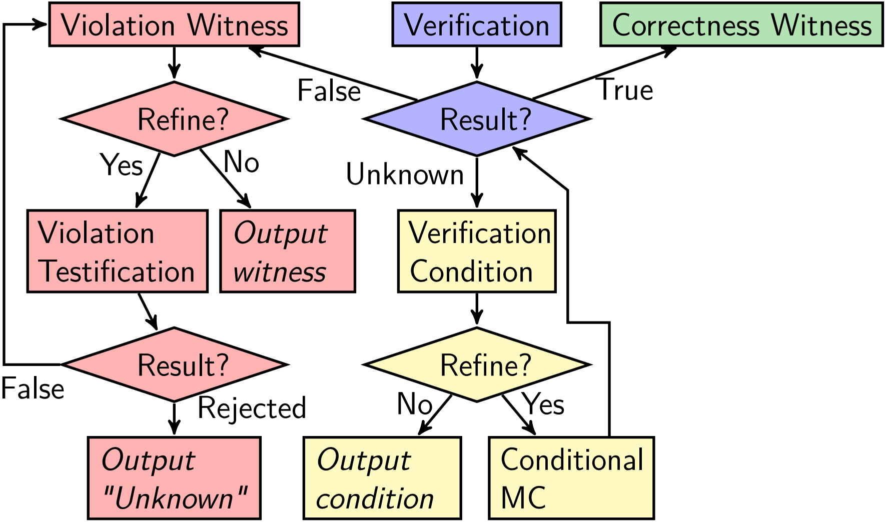
Stepwise Testification: Classification



Stepwise Testification: Classification

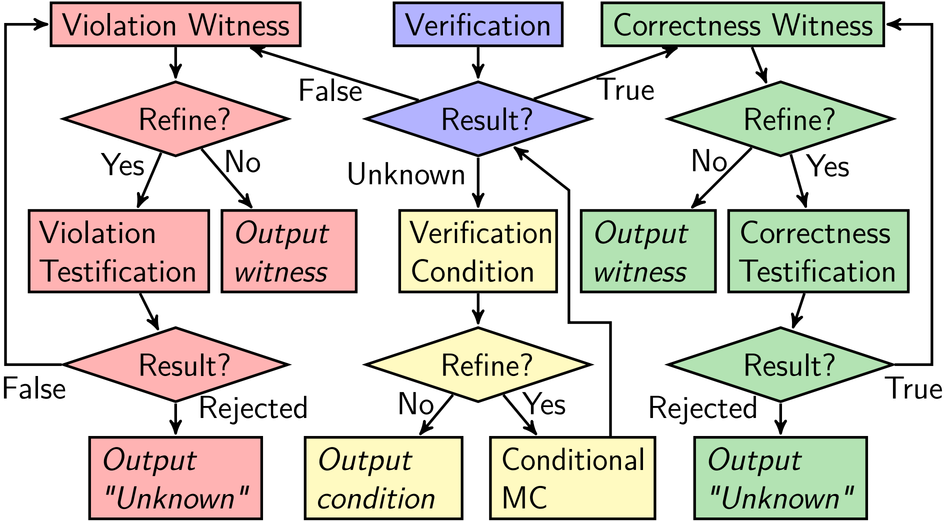


Stepwise Testification: Classification



UNIVERSITÄT PASSAU LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Stepwise Testification: Classification



Conclusion

Correctness-Witnesses ...

1. are **easy to implement** for verifiers that already support **violation witnesses**

Conclusion

Correctness-Witnesses ...

1. are **easy to implement** for verifiers that already support **violation witnesses**
2. enable information exchange **across different software verifiers**

Conclusion

Correctness-Witnesses ...

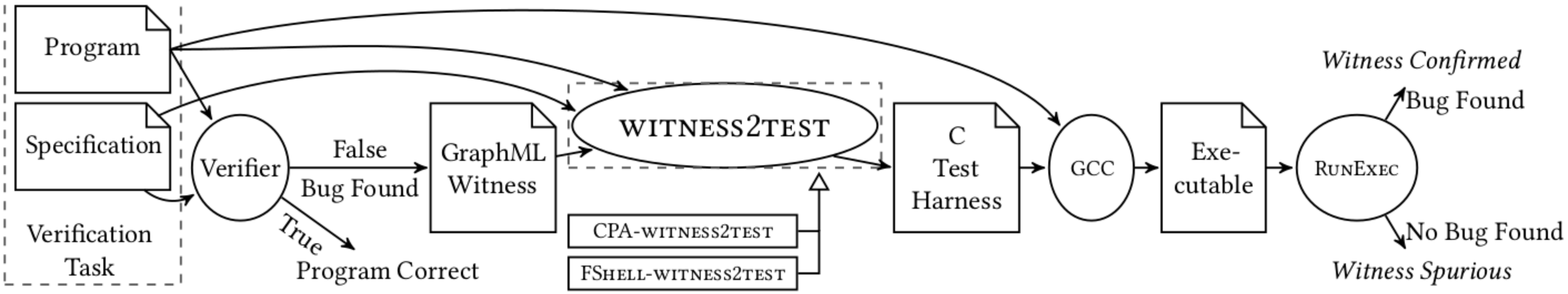
1. are **easy to implement** for verifiers that already support **violation witnesses**
2. enable information exchange **across different software verifiers**
3. **efficiently increase confidence** in results by **validation**

Work in Progress: Execution-Based Witness-Validation

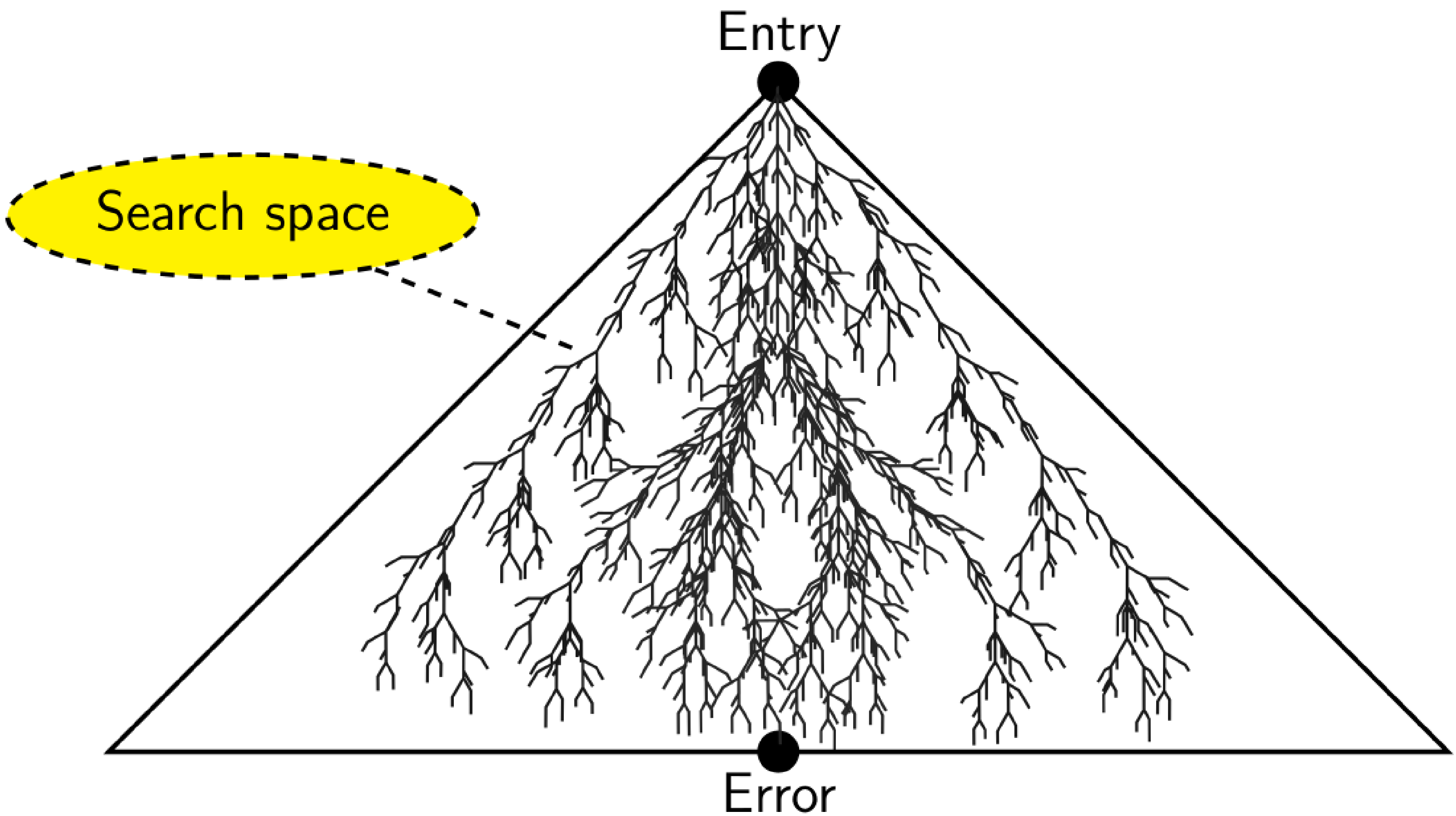
Dirk Beyer

LMU
LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

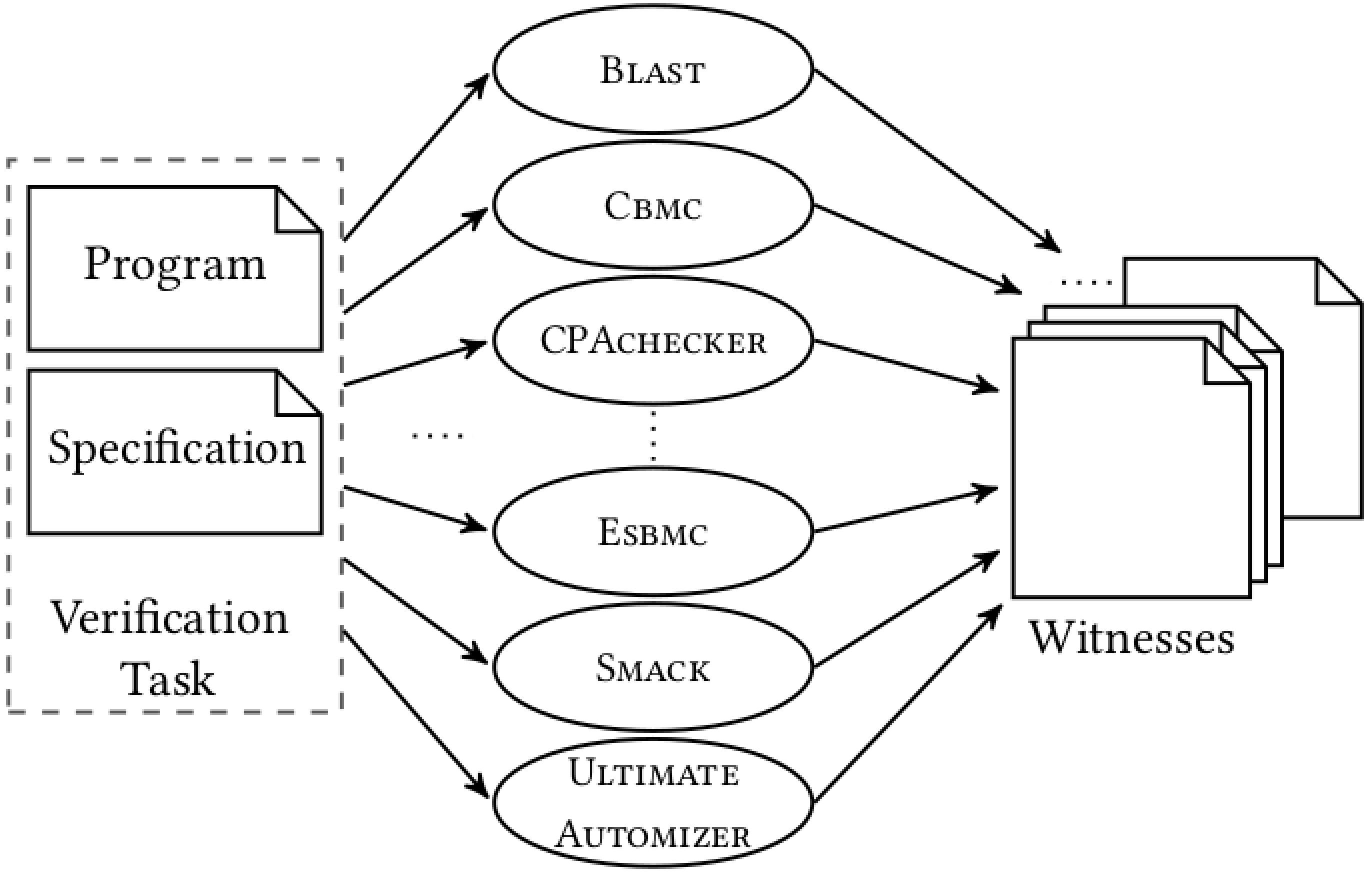
Practical Impact: Get Tests from Verification Tools



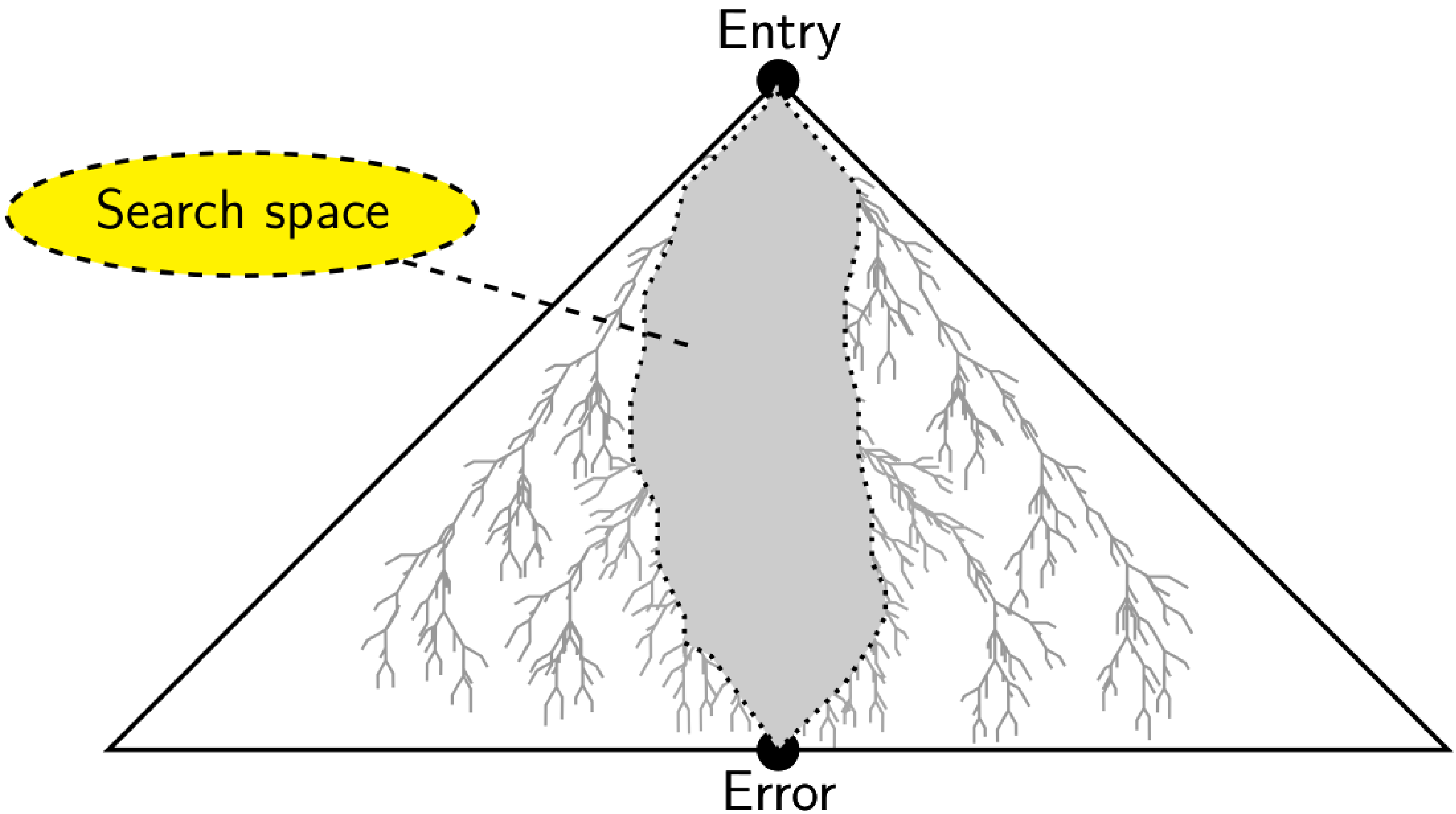
Search-Space Reduction for Stepwise Testification



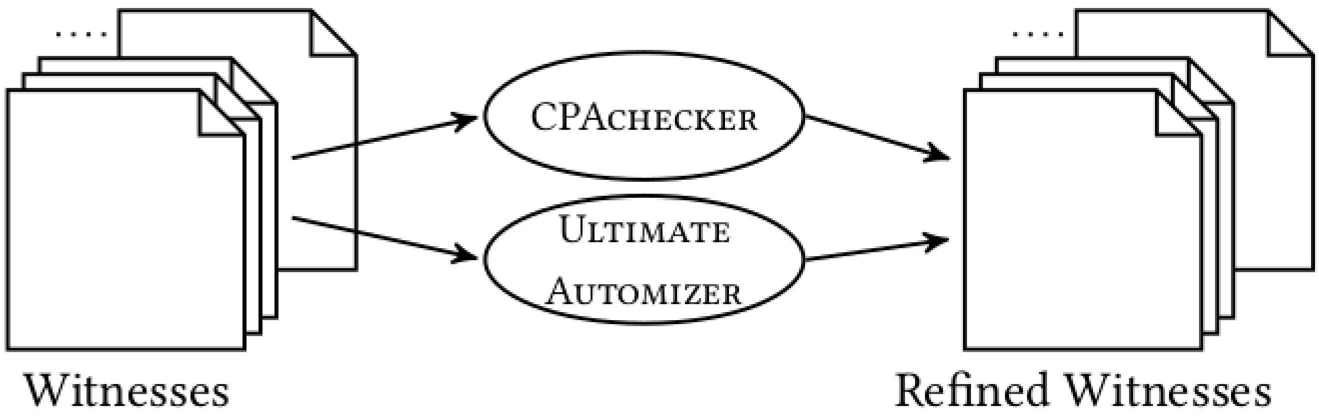
Produce Witnesses



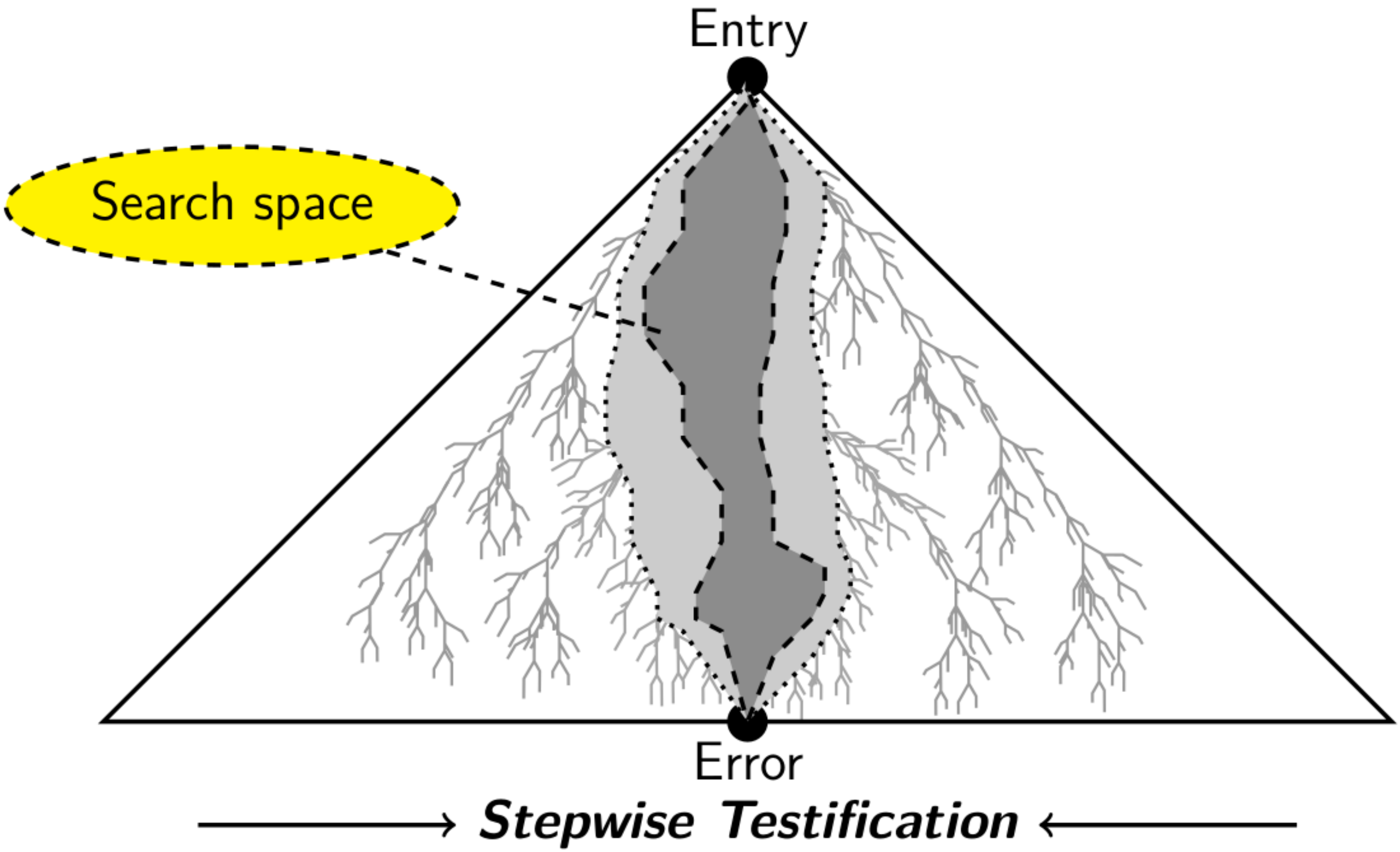
Search-Space Reduction for Stepwise Testification



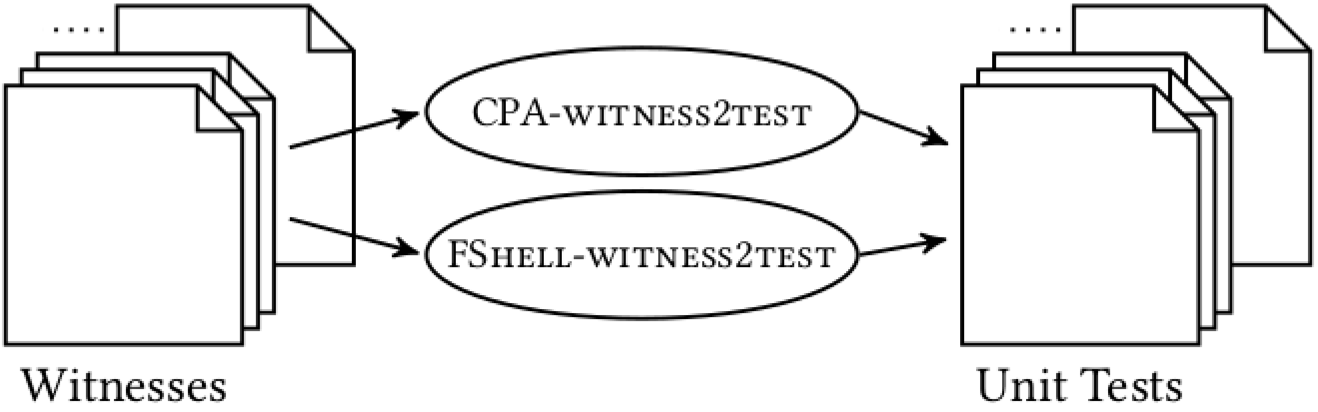
Refine Witnesses



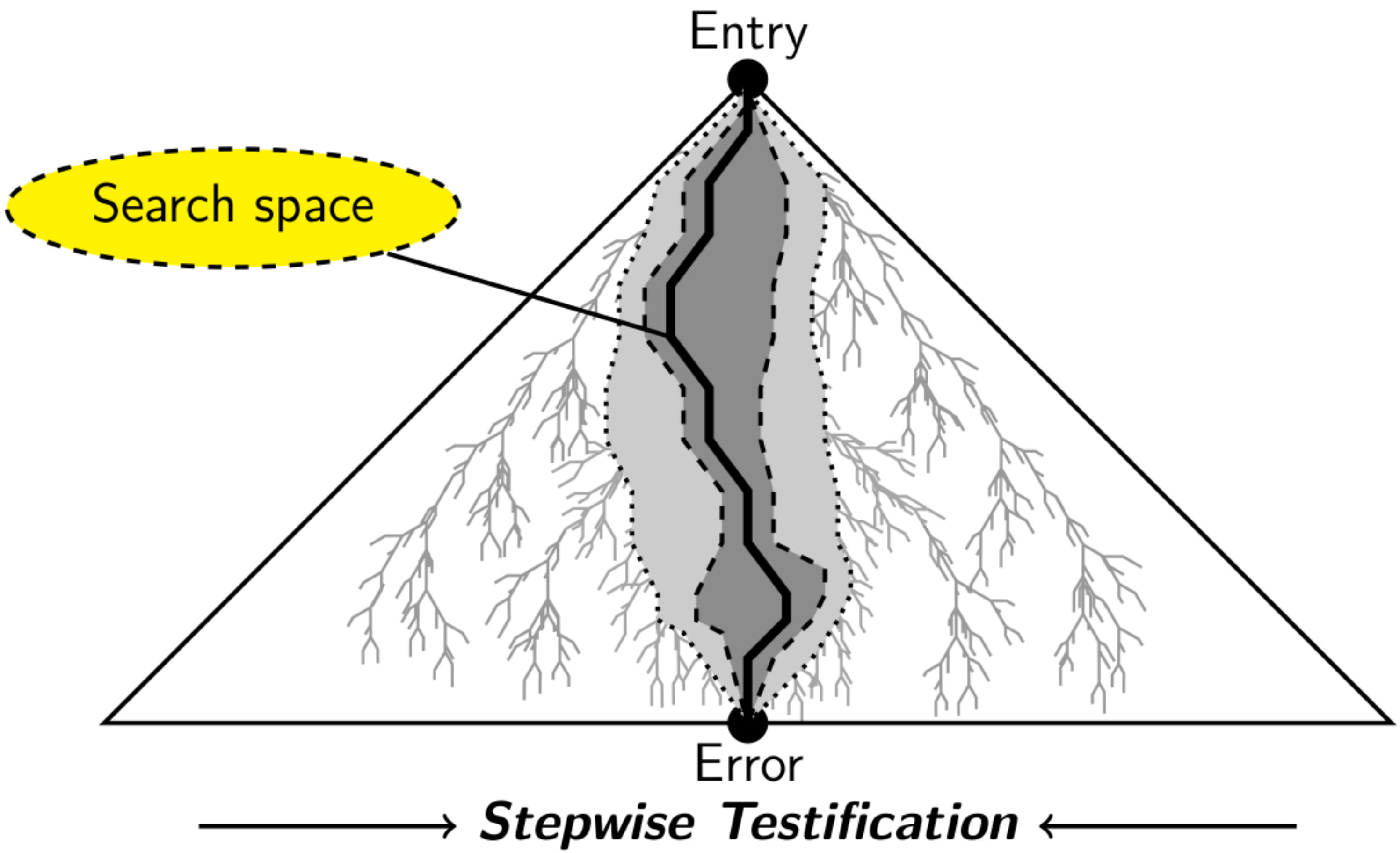
Search-Space Reduction for Stepwise Testification



Produce Unit Tests From Witnesses



Search-Space Reduction for Stepwise Testification



Conclusion

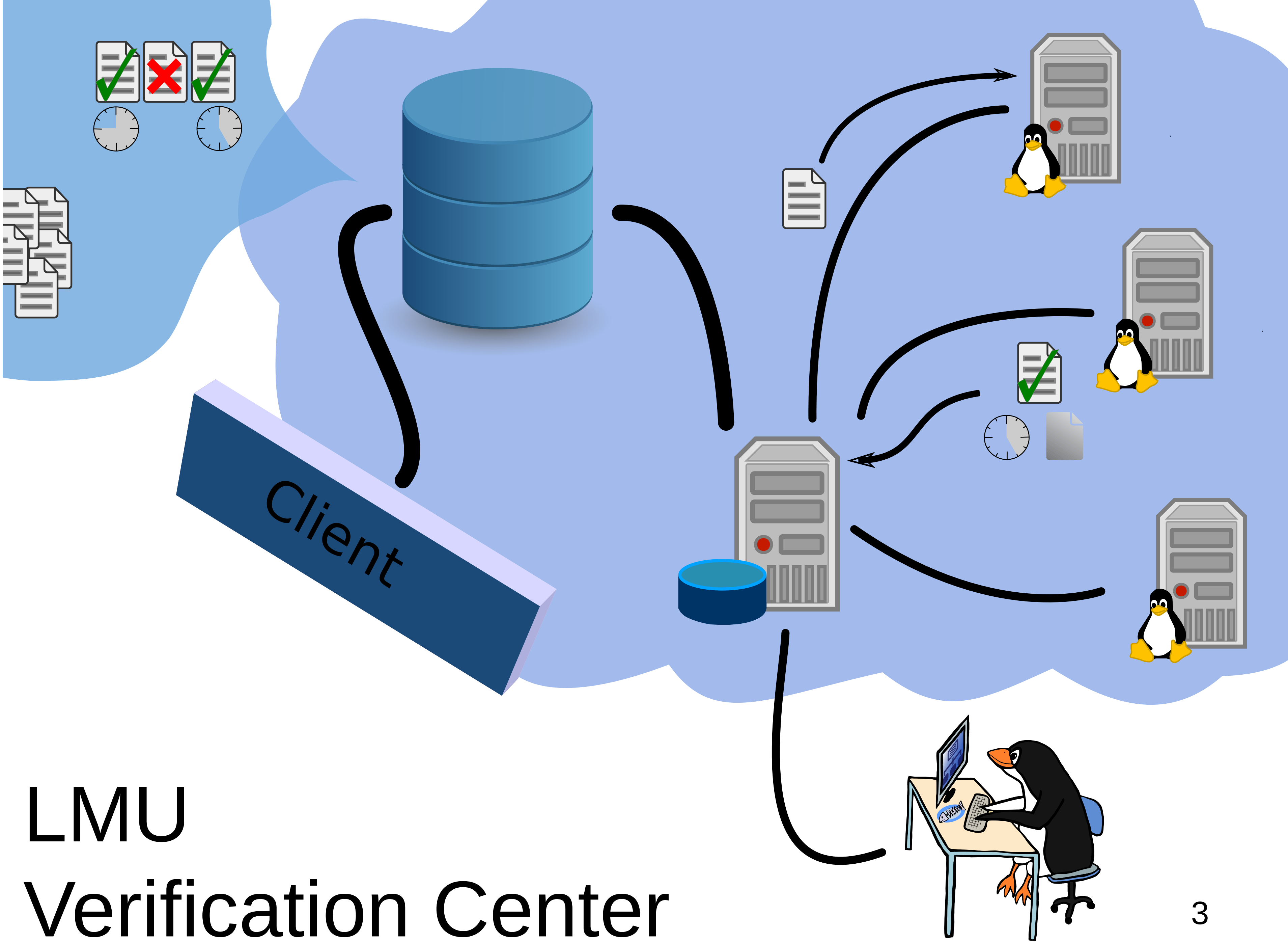
- Turn outcome of verification tools into objects that the developers can deal with
- Imagine a tool-independent format for test cases
- Complement test suites with test cases from verification tools

Summary – Part 3

- Test from Verification

Conclusion

- Combine different verification tools
  → Conditional Model Checking
- Store intermediate results
  → Regression Verification, e.g., Precision Reuse
- Store witnesses of the verification result
  → Witness-Based Results-Validation
- Use test cases as interface
  → Pass Tests from Verifiers to Development
- Reuse: save data, collect data, share data
  → Stateful Verification



LMU
Verification Center