Static Verification Results
Visualization in the Context of SV-COMP

Vitaly Mordan
mordan@ispras.ru

Ivannikov Institute for System Programming of the Russian Academy of Sciences
Static Verification State of the Art

- More than 31 tools*
- Improvements in effectiveness and efficiency*
- Validation of verification results*
  - Both property violations and correctness proofs*
- Different properties*
  - Potential for extensions (e.g., property automata)
- Verification of C and Java programs*

What about results analysis?

*D. Beyer. Automatic Verification of C and Java Programs: SV-COMP 2019. 2/35
Verification of Industry System

- **N programs**
- **M properties**

**Verifier**
- **P violation witnesses**
- **Q correctness witnesses**

**Validator**

**Results analysis**
- Found bugs
- Incorrect proofs
- Problems in tools
- ...

**Automatic step:**
required resources can be minimized with modern approaches and cloud technologies.

**Manual step:**
required resources are harder to reduce (user experience), but cost much more.
Related Work

• BenchExec* table-generator
  • Score (based on tasks definition)
  • Plots with consumed resources
  • Comparison tables
  • Witness validation results
  • Witnesses are not visualized

• LDV Tools (Klever)**
  • Preset environment models for Linux/BusyBox/etc.
  • Violation witnesses visualization
    • Specific format

* Presented in machine-readable format
** Not supported by SV-COMP tools

Cannot visualize generic witness from SV-COMP tools

* https://github.com/sosy-lab/benchexec
** https://forge.ispras.ru/projects/klever
Suggested Solutions

• Witness Visualizer (user-friendly witnesses)
  • Helps to locate bugs for the users
  • Helps to reveal problems in tools
• Correctness witnesses visualization (idea)
  • Shows main proof hints (for developers)
  • Presents source code coverage (for users)
• Benchmark Visualizer (continuous verification)
  • Visualizes BenchExec results
  • Groups witnesses for each benchmark
Common Witness Format*

- Machine-readable format
- There are still differences among tools

**Witness Visualizer**

- **Verifiers**
  - **GraphML witness**
    - **Unknown**
    - **Safe**
    - **Unsafe**
      - **CPAchecker** different configs
      - **UAutomizer** missing main call
      - **VeriAbs** no source code
      - **ESBMC-kind** missing elements

- **Witness Visualizer**
  - User-friendly witness

* [https://github.com/ispras/cv](https://github.com/ispras/cv)
Requirements to the Witness Visualizer

- Fault tolerance (to the missing elements)
  - Support common witness format (GraphML)
- Quality control (for developers)
  - Provide feedback on the missing elements
- Support violation hints
  - Helps with large witnesses
- Provide operations with witnesses
  - Comparison
- Support both violation and correctness types
Fault Tolerance

- Cannot be tolerated
  - Parsing failures (wrong format)
  - Empty witnesses

- Restorable missing elements
  - Source code (program file + line/offset)
  - Entry point (based on property description)
  - Property violation (last edge)

- Elements, which cannot be restored
  - Call stack
  - Assumptions/controls
Quality Control

• Provide useful feedback to the developers
  • Source files do not exist
  • Call stack is missing
  • Conditions are missing
  • Entry point is missing
  • Produced warnings during visualization

1) No call stack (enterFunction tag)
2) No conditions (control tag)

Warning: some elements are missing
Violation Hints

- Core elements, which describe the given violation
- Reason – visualize large witnesses
  - Highlight violation hints
    - With call stack, source code link, thread id, etc.
  - Hide other elements
- Violation hints extraction
  - From witnesses (“note”, “warning”)
    <data key="note">Acquire mutex_lock</data>*
  - From property
  - From source code**

* Example is based on witnesses from CPA-Lockator tool.
** Based on model comments (applied in LDV Tools).
Violation Hints Usage Example

**Initial witness**

```c
main()
    void *x = NULL;
    int flags;
    int size;
    int i = 0;
    f1(i)
        assume(i < 10)
    f2(i)
    f3(i)
    i := i + 1
...
x = alloc(size, flags)
    return NULL
...
free(x)
```

**Processed witness**

```c
main()
    void *x = NULL;
    int flags;
    int size;
    int i = 0;
    f1(i)
        assume(i < 10)
    f2(i)
    f3(i)
    i := i + 1
...
Allocate memory for x
    Failed to allocate x
    ...
Null ptr dereference on x
```
Operations with Witnesses

- Witnesses comparison
  - Distinguish different witnesses (error paths)
  - Filter several witnesses*
  - Can be done for validated witnesses only

* For example, SV-COMP tool *CPA-Lockator* can produce several witnesses for concurrency properties.
SV-COMP Tools Violation Witnesses

<table>
<thead>
<tr>
<th>SV-COMP Tool</th>
<th>Witness elements</th>
<th>Source code</th>
<th>Violation Hints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Call stack</td>
<td>Entry point</td>
<td>Assumptions/controls</td>
</tr>
<tr>
<td>2LS</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>AProVE</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>CBMC</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>CBMC-Path</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>CPA-BAM-BnB</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>CPA-Lockator</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>CPA-Seq</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DepthK</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DIVINE-explicit</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>DIVINE-SMT</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>ESBMC-kind</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Lazy-CSeq</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Map2Check</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>PeSCo</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Pinaka</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>PredatorHP</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Skink</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMACK</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Symbiotic</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>UAutomizer</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>UKojak</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>UTaipan</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VeriAbs</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>VeriFuzz</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>VIAP</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Yogar-CBMC</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Yogar-CBMC-Parallel</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

* 4 verifiers for Java programs were excluded from this comparison, because they do not produce witnesses.
Example of a Witness with Violation Hints

- Input/output memory map operations: `ioremap`, `pci_ioremap_bar`, ...
- Input/output memory unmap operation: `iounmap`
Example of a Witness with Violation Hints

- Input/output memory map operations: `ioremap`, `pci_ioremap_bar`, ...
- Input/output memory unmap operation: `iounmap`
Example of a Witness with Missing Elements

Witness was produced by ESBMC-kind tool.
Witness Visualizer Application Area

- Demonstration of a generic witness
  - Supports any SV-COMP tool
- Feedback to the developers
  - Missing elements, warnings, etc.
- Large witnesses visualization
  - Based on extracted violation hints
- Comparison of witnesses
  - Required for several witnesses
Suggested Solutions

- **Witness Visualizer (user-friendly witnesses)**
  - Helps to locate bugs for the users
  - Helps to reveal problems in tools

- **Correctness witnesses visualization (idea)**
  - Shows main proof hints (for developers)
  - Presents source code coverage (for users)

- **Benchmark Visualizer (continuous verification)**
  - Visualizes BenchExec results
  - Groups witnesses for each benchmark
Correctness Witnesses

- Present main verification result (proof)
  - Ensure the absence of missed bugs
- Hard to visualize (graph structure)
General Ideas of the Visualization

- Support of common format* (GraphML)
- Witness preprocessing
  - Convert to the plain structure
- Extract main proof hints
  - Conditions, invariants, etc.
- Get source code coverage

Implementation of the Suggested Ideas

- Proof hints
  - Conditions
  - Invariants (common and local)

- Witness preprocessing
  - Sort all elements by line/thread/source file
  - Combine all assumptions for conditions
  - Extract common invariants

- Witness comparison
  - Is not supported (only 1 (?) witness is expected)
Correctness Witness Model Example

“Developer” view

Main proof hints

Conditions
  condition(cond1)
  condition(cond2)
...
Common invariants
  invariant(inv1)
...
Invariants
  Multiple invariants
    invariant(inv2)
    invariant(inv3)
...

“User” view

Source code coverage

Line 1 - covered condition
Line 2 - covered line
Line 3 - uncovered
...
Condition line

Invariant scope

All branches are covered

Some branches were not covered
Sometimes SV-COMP tools may produce empty correctness witnesses.
Correctness Witnesses Visualization

- Suggested general ideas of the visualization
  - Based on plain structure and proof hints
- Suggested implementation of the ideas
  - Based on conditions and invariants
  - Other implementations can be suggested
- Idea to extract source code coverage
- Can be useful for both developers and users
Suggested Solutions

- Witness Visualizer (user-friendly witnesses)
  - Helps to locate bugs for the users
  - Helps to reveal problems in tools
- Correctness witnesses visualization (idea)
  - Shows main proof hints (for developers)
  - Presents source code coverage (for users)
- Benchmark Visualizer (continuous verification)
  - Visualizes BenchExec results
  - Groups witnesses for each benchmark
Benchmark Visualizer*

- Web-interface** for verification results visualization
  - Easy to setup and use
- Database
  - Benchmark verification results
  - User marks: bugs, incorrect proofs, etc.

Differences with BenchExec table-generator
- Witnesses in user-friendly format
- Means for manual results analysis
- Coverage (if presented)

* https://github.com/ispras/cv
** https://github.com/mutilin/klever (branch cv-v2.0)
### Benchmark Visualizer Database

**Data base**

- Benchmark verification results $T$
- Benchmark verification results $T+1$
- Witnesses marks

**Results transitions**

<table>
<thead>
<tr>
<th>Safe $\rightarrow$ Unknown</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsafe $\rightarrow$ Unknown</td>
<td>25</td>
</tr>
<tr>
<td>Unknown $\rightarrow$ Safe</td>
<td>6</td>
</tr>
<tr>
<td>Unknown $\rightarrow$ Unsafe</td>
<td>2</td>
</tr>
</tbody>
</table>

**Witness clusters**

- Common: 23
- Lost: 27
- New: 24

**Consumed resources**

- Common: 23
- Lost: 27
- New: 24

**Speedups manual analysis**

- Benchmark verification results $T+1$
- Add new results
- Compare new witnesses with already marked ones
Manual Results Analysis*

- User analyses a witness to determine a bug
- User creates a mark for a bug
  - Witness + comparison criteria + description
- User adjusts the created mark

The mark is applied to a witness, which corresponds to other bugs.

There is a witness, which corresponds to the same bug, but the mark was not applied to it.

* Based on violation witnesses.
Benchmark Visualizer Example

- Violation witnesses
  - Unsafes: 59
    - Details:
      - Stags: 1
      - False positives: 13
      - Rule: 5
      - EnvironmentModel: 8
      - Unknown: 3
      - Without marks: 42

- Correctness witnesses
  - Safes: 11
    - Details:
      - Unknown: 7
      - Low coverage: 7
      - Without marks: 4

- Verifier logs
  - Unknowns: 28
    - Components:
      - CPAChecker: 28
        - Assertion: 19
        - Parsing: 1
        - Soft time limit: 23
        - Time limit: 23

- Coverage per property
  - Coverage: 56.61% by functions / 20.81% by lines
    - Details:
      - Covered by all properties: 56.61% / 20.81%
      - Property ‘unreach-call’: 56.61% / 20.81%

- Consumed resources plots
  - Consumed resources: 35835 s / 15 GB
    - Details:

- Auxiliary data / filters

**Configuration**
- CPU cores limit: 2
- CPU time limit: 900s
- Memory limit: 16000000000
- Options: Show

**CPAChecker, ldv, ldv-linux-4.0-rc1-mav (2019_08_28_14_57_18)**
- Description: SV-COMP benchmarks results
- Last change: 2 weeks, 5 days ago (uploader uploader)
- Status: Solved (components tree)

**Attributes filters**
- Rule specification: unreach-call
- Verification object: 98 elements
**Benchmark Visualizer Application Area**

- **Continuous verification of industry systems**
  
  ![Diagram of the verification process]

  1. New revision
  2. Verification tasks* (programs+properties)
  3. Launch on cloud/locally
  4. BenchExec (any verifier)
  5. Witnesses
  6. Validator

  - Marks for future results analysis
  - Found bugs
  - Incorrect proofs
  - Problems in tools

- **SV-COMP tasks**
  - Similar work-flow with the given verification tasks

* May require additional preparation by the user or other tool.
Conclusion

- Witness Visualizer
  - Converts witnesses to user-friendly format
- Correctness witnesses visualization
  - New ideas of visualization
  - A simple implementation of the ideas
- Benchmark Visualizer
  - Successfully applied to continuous verification
  - Can be applied for SV-COMP tasks
Future Plans

• Witness Visualizer improvements
  • Restore missing elements where possible
  • Improve feedback to the developers

• Correctness witnesses visualization
  • Suggest other views based on proof hints
  • Implement automatic coverage extraction

• Benchmark Visualizer improvements
  • Regression verification
  • Verification tasks preparation
The joint 4rd International Workshop on CPAchecker (CPA'19) and 9th Linux Driver Verification (LDV) Workshop
October 2, 2019, Frauenchiemsee, Germany

Thank you

Vitaly Mordan
mordan@ispras.ru