Correctness Witnesses: Exchanging Verification Results Between Verifiers

Replication Package

Dirk Beyer [1], Matthias Dangl [1], Daniel Dietsch [2], Matthias Heizmann [2]

[1] University of Passau, Germany [2] University of Freiburg, Germany

ABSTRACT

Standard verification tools provide a counterexample to witness a specification violation, and, since a few years, such a witness can be validated by an independent validator using an exchangeable witness format. This way, information about the violation can be shared across verification tools and the user can use standard tools to visualize and explore witnesses. This technique is not yet established for the correctness case, where a program fulfills a specification. Even for simple programs, it is often difficult for users to comprehend why a given program is correct, and there is no way to independently check the verification result. We close this gap by complementing our earlier work on violation witnesses with correctness witnesses. While we use an extension of the established common exchange format for representing correctness witnesses, the techniques for producing and validating correctness witnesses are completely different. The overall goal to make proofs available to engineers is probably as old as programming itself, and proof-carrying code was proposed two decades ago — our goal is to make it practical: We consider witnesses as first-class exchangeable objects, stored independently from the source code and checked independently from the verifier that produced them, respecting the important principle of separation of concerns. At any time, the invariants from the correctness witness can be used to reconstruct a correctness proof to establish trust. We extended two state-of-the-art verifiers, CPAchecker and UltimateAutomizer, to produce and validate witnesses, and report that the approach is promising on a large set of verification tasks.

This document describes our replication package for the artifact-evaluation process.

1. DESCRIPTION

In the following, we describe our replication package along the dimensions of the scorecard outlined in the artifact-evaluation guidelines [1].

1.1 Insightfulness

In our paper, we address machine-readable and exchangeable witnesses for the correctness of programs, which solves the current and pressing problem of validating the verification results of an untrusted verifier for the verification verdict True. Our replication package gives researchers the option to reproduce our experimental results and conduct their own experiments with our implementations.

1.2 Usefulness

In the paper, we argue that an exchange format for witnesses becomes more useful as its adaption by tools spreads. Our replication package provides guidance for tool developers to integrate support for correctness witnesses into their verifiers and develop new validators. The implementations provided with our replication package, namely CPAchecker and UltimateAutomizer are the first example of two verifiers based on completely different frameworks that are able to communicate information about program proofs to each other. No such form of communication is possible without a common exchange format, and was therefore previously impossible.

1.3 Usability

We give a detailed description of the differences between our new format for correctness witnesses and the existing format for violation witnesses [1] we extended. Our supplementary web page [2] provides, lists, and describes all experimental data. Additionally, the supplementary web page contains a tutorial, which gives detailed instructions on how to install the tools in the precise versions used in our evaluation, and on how to apply our tools to obtain and validate correctness witnesses. Using the tools is as easy as downloading and unpacking the corresponding archives from the web page and following the steps of the tutorial. Also available on the web page is a virtual-machine image that contains a copy of our paper and the two tools. Another way to obtain our two open-source tools is to check them our directly from their repositories [3] [4]. This way, you can easily obtain the latest bug fixes and new features. The revisions used for our evaluation are documented in our paper.

1.4 License

The source code of UltimateAutomizer is licensed under the LGPLv3 license with a linking exception to Eclipse RCP and Eclipse CDT. The source code of CPAchecker, our supplementary webpage, experimental data, and virtual machine image are licensed under the Apache License, Version 2.0.

[1]http://www.cs.ucdavis.edu/fse2016/calls/artifacts/

[2]http://sosy-lab.org/~dbeyer/correctness-witnesses/

[3]https://svn.sosy-lab.org/software/cpachecker/trunk/

[4]https://github.com/ultimate-pa/ultimate

2. REPLICATION PACKAGE

This section describes the replication package for our paper ”Correctness Witnesses: Exchanging Verification Results Between Verifiers”. Our supplementary web page [5] provides all experimental data, and a virtual machine that contains our implementations and has been prepared such that our results can be replicated easily. In this section, we will give an overview over the proposed exchange-format for correctness witnesses, such that the reader may understand our approach and derive an own implementation of our concepts. Further details on how to replicate our experiments inside or outside of our virtual machine can be found on the supplementary web page [5], which also contains a tutorial.

2.1 Witness Exchange Format

Our exchange format for correctness witnesses extends the exchange format for violation witnesses [1] to add the possibility to attach invariants to witness-automata states.

Source-Code Guards. From the existing format for violation witnesses we adopt all source-code guards, such as `startline` and `endline`, which are used to map a transition in the witness automaton to lines in the original program. The source-code guard `control` also continues to be used to distinguish between different branches in the program. Valid values for this guard are `condition-false` and `condition-true`, where for a conditional branching in the original program, the then-branch is referred to by the value `condition-true`, and the else-branch is referred to by the value `condition-false`. Given such a `control` guard, an observer-automaton transition matches if the observed analysis takes the control-flow edge corresponding to the specified branch, but not its counterpart.

State-Space Guards. In our format for correctness witnesses, we forbid the usage of the state-space guards used to restrict the state-space exploration in violation witnesses, because the validation of invariants requires an unrestricted exploration of the state space to ensure that violations cannot be hidden from the validator. Specifically, this ban affects the guard `assumption`, which is currently the only type of state-space guard used in violation witnesses.

States and Invariants. In violation witnesses, automata states can be declared as `entry`, `sink`, or `violation` states, where the default value is `false`. In correctness witnesses, no state must be declared as a violation state or sink state, because a violation state would contradict the purpose of the witness, and because a sink state would restrict the exploration of the state space, which could be used to hide violations from the validator.

To attach invariants to automata states, we introduce two new keys for state data tags, namely `invariant` and `invariant.scope`. Valid values for the `invariant` data tag are expressions of the input programming language, such as `(x == y && x > 0)`. All variables used in these invariants must appear in the original program code. Name conflicts between local variables that have the same name as local variables of other functions or global variables can be resolved by using a data tag with the key `invariant.scope` and, as its value, the name of function the invariant is intended to be interpreted in. This mechanism is symmetric to the dualism of `assumption` and `assumption.scope` in violation witnesses.

[5]http://sosy-lab.org/~dbeyer/correctness-witnesses/

```
44 <graph edgedefault="directed">
45  <data key="witness-type">
        ↪ correctness_witness</data>
46  <data key="sourcecodelang">C</data>
47  <data key="producer">CPAchecker
        ↪ 1.5-svn</data>
48  <data key="programfile">
        ↪ example-safe.c</data>
49  <data key="programhash">
        ↪ 6079971a175b6038f483a816d335d0a069862081
        ↪ </data>
50  <data key="memorymodel">precise</data>
51  <data key="architecture">32bit</data>
52  <node id="q0">
53   <data key="entry">true</data>
54  </node>
55  <node id="q1">
56   <data key="invariant">(y == x)</data>
57   <data key="invariant.scope">main</data>
58  </node>
59  <edge source="q0" target="q1">
60   <data key="startline">6</data>
61   <data key="control">condition-true</data>
62  </edge>
63  <node id="q2"/>
64  <edge source="q0" target="q2">
65   <data key="startline">6</data>
66   <data
         ↪ key="control">condition-false</data>
67  </edge>
68  <node id="q3"/>
69  <edge source="q1" target="q3">
70   <data key="startline">7</data>
71  </edge>
72  <edge source="q3" target="q0">
73   <data key="startline">8</data>
74  </edge>
75 </graph>
76</graphml>
```

Figure 1: Correctness-witness automaton for the introductory safe example program of the paper (Fig. 1a) in GraphML format; header omitted for brevity

2.2 Example Witness

Fig. 1 shows the witness automaton produced by CPAchecker for the example C program from the introduction to our paper, stripped down to the important parts and without the GraphML header [2], which has been omitted for brevity. Lines 52 to 54 show the entry state q_0. Lines 55 to 58 show that, as described in the introduction to our paper, the witness contains the invariant `(y == x)` at state q_1, using the new data-tag key `invariant`, and that the scope of the variables in this expression is the function `main`, using the new data-tag key `invariant.scope`. Lines 59 to 62 represent the transition from state q_0 to q_1, corresponding to the then-branch of line 6 in the C program. Lines 63 and 68 declare the states q_2 and q_3, respectively. The transition from q_0 to q_2 in lines 64 to 67 corresponds to the else-branch of line 6 in the C program. Lines 69 to 74 show the transitions from q_1 over q_3 back to q_0, corresponding to the loop body.

3. REFERENCES

[1] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.

[2] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report. In *Graph Drawing*, LNCS 2265, pages 501–512. Springer, 2001.