Ultimate Automizer and the Search for Perfect Interpolants
(Competition Contribution)

Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski

University of Freiburg, Germany

Abstract. Ultimate Automizer is a software verifier that generalizes proofs for traces to proofs for larger parts for the program. In recent years the portfolio of proof producers that are available to Ultimate has grown continuously. This is not only because more trace analysis algorithms have been implemented in Ultimate but also due to the continuous progress in the SMT community. In this paper we explain how Ultimate Automizer dynamically selects trace analysis algorithms and how the tool decides when proofs for traces are "good" enough for using them in the abstraction refinement.

1 Verification Approach

Ultimate Automizer (in the following called Automizer) is a software verifier that is able to check safety and liveness properties. The tool implements an automata-based [6] instance of the CEGAR scheme. In each iteration, we pick a *trace* (which is a sequence of statements) that leads from the initial location to the error location and check whether the trace is *feasible* (i.e., corresponds to an execution) or *infeasible*. If the trace is feasible, we report an error to the user; otherwise we compute a sequence of predicates along the trace as a proof of the trace's infeasibility. We call such a sequence of predicates a sequence of *interpolants* since each predicate "interpolates" between the set of reachable states and the set of states from which we cannot reach the error. In the refinement step of the CEGAR loop, we try to find all traces whose infeasibility can be shown with the given predicates and subtract these traces from the set of (potentially spurious) error traces that have not yet been analyzed. We use automata to represent sets of traces; hence the subtraction is implemented as an automata operation. The major difference to a classical CEGAR-based predicate abstraction is that we never have to do any logical reasoning (e.g., SMT solver calls) that involves predicates of different CEGAR iterations.

We use this paper to explain how our tool obtains the interpolants that are used in the refinement step. The Ultimate program analysis framework provides a number of techniques to compute interpolants for an infeasible trace. We group them into the following two categories.

Path program focused techniques (*abstract interpretation [5], constraint-based invariant synthesis*) These techniques do not consider the trace in isolation but in the context of the analyzed program. The program is projected to the statements that occur in the trace; this projection is considered as a standalone program called *path program*. The techniques try to find a Floyd-Hoare style proof for the path program, which shows the infeasibility of all the path program's traces. If such a proof is found, the respective predicates are used as a sequence of interpolants. These interpolants are "good enough" to ensure that in the refinement step all (spurious) error traces of the path program are ruled out.

Trace focused techniques (*Craig interpolation, symbolic execution with unsatisfiable cores [4]*) These techniques consider only the trace. Typically they are significantly less expensive and more often successful than techniques from the first category. However, we do not have any guarantee that their interpolants help to prove the infeasibility of more than one trace.

Recent improvements of Automizer were devoted to techniques that fall into the second category. Our basic paradigms are: (1) use different techniques to compute many sequences of interpolants, (2) evaluate the "quality" of each sequence, (3) prefer "good" sequences in the abstraction refinement.

In contrast to related work [3] we have only one measure for the quality of a sequence of interpolants: We check if the interpolants constitute a Floyd-Hoare annotation of the path program for the trace. If this is the case, we call the sequence a *perfect sequence of interpolants*. If the sequence is perfect, we use it for the abstraction refinement. If the sequence is not perfect, we only use it if no better sequence is available. Our portfolio of *trace focused techniques* is quite large for three reasons.

1. We use different algorithms for interpolation. Several SMT solvers have implemented algorithms for Craig interpolation and we use these as a black box. Furthermore, Ultimate provides an algorithm [4] to construct an abstraction of the trace from an unsatisfiable core provided by an SMT solver. Afterwards, two sequences of predicates, one with the `sp` predicate transformer, the other with the `wp` predicate transformer, are constructed via symbolic execution.
2. We use different SMT solvers. Typically, different SMT solvers implement different algorithms and hence the resulting Craig interpolants or unsatisfiable cores are different.
3. We have several algorithms that produce an abstraction of a trace but preserve the infeasibility of the trace. We can apply these as a preprocessing of the interpolant computation.
   All our algorithms follow the same scheme: We replace all statements of the trace by `skip` statements. Then we incrementally check feasibility of the trace and undo replacements as long as the trace is feasible. Examples for the undo order of our algorithms are: (1) Apply the undo first to statements that occur outside of loops, follow the nesting structure of loops for further undo operations. (2) Do the very same as the first algorithm but start inside loops.

(3) Apply the undo to statements with large constants later. (4) Apply the undo to statements whose SMT representation is less expensive first (e.g., postpone floating point arithmetic).

At first glance it looks like a good idea to apply different techniques to a given trace for as long as no perfect sequence of interpolants was found. This has however turned out to be a bad idea for the following reasons.

1. The path program might be unsafe and we just have to unwind a loop a few times until we find a feasible counterexample.
2. The path program might be so intricate that we are unable to find a loop invariant. However, there are cases where the loop can only be taken for a small number of times and our tool can prove correctness by proving infeasibility of each trace individually.
3. The path program might be so intricate that we are unable to find a loop invariant immediately. But if we consider certain unwindings of the loop (e.g., the loop is taken an even number of times) our interpolants will form a loop invariant.

We conclude that per iteration of the CEGAR loop (resp. per trace) we only want to apply a fixed number of techniques. According to our experiments there are some techniques that are on average more successful than others; however, no technique is strictly superior to another. Hence it is neither a good idea to always apply the n typically most successful techniques nor to take n random techniques in each iteration.

We follow an approach that we call *path program-based modulation*. We have a preferred sequence in which we apply our techniques. Whenever we see a *new* trace we start at the beginning of this sequence. Whenever we see a trace that is *similar* to a trace we have already seen, we continue in the sequence of techniques at the point where we stopped for the similar trace. Our notion of similarity is: Two traces are similar if they have the same path program.

Hence we make sure that for every path program every technique is eventually applied to some trace of the path program.

2 Project, Setup and Configuration

Automizer is developed on top of the open-source program analysis framework Ultimate[1]. Ultimate is mainly developed at the University of Freiburg and received contributions from more than 50 people. The framework and Automizer are written in Java, licensed under LGPLv3 , and their source code is available on Github[2].

Automizer's competition submission is available as a zip archive[3]. It requires a current Java installation ($\geq$ JRE 1.8) and a working Python 2.7 installa-

[1] `https://ultimate.informatik.uni-freiburg.de`

[2] `https://github.com/ultimate-pa/ultimate`

[3] `https://ultimate.informatik.uni-freiburg.de/downloads/svcomp2018/UltimateAutomizer-linux.zip`

tion. The archive contains Linux binaries for AUTOMIZER and the required SMT solvers Z3[4], CVC4[5], and MATHSAT[6], as well as a Python script, `Ultimate.py`. The Python script translates command line parameters and results between ULTIMATE and SV-COMP conventions, and ensures that ULTIMATE is correctly configured to run AUTOMIZER. AUTOMIZER is invoked through `Ultimate.py` by calling

```
./Ultimate.py --spec prop.prp --file input.c --architecture
  32bit|64bit --full-output [--validate witness.graphml]
```

where `prop.prp` is the SV-COMP property file, `input.c` is the C file that should be analyzed, `32bit` or `64bit` is the architecture of the input file, and `--full-output` enables writing all output instead of just the status of the property to `stdout`. The option `--validate witness.graphml` is only used during witness validation and allows the specification of a file containing a violation [2] or correctness witness [1].

Depending on the status of the property, a violation or correctness witness may be written to the file `witness.graphml`. AUTOMIZER is not only able to generate witnesses, but also to validate them[7]. In any case, the complete output of AUTOMIZER is written to the file `Ultimate.log`.

The benchmarking tool BENCHEXEC[8] contains a tool-info module that provides support for AUTOMIZER (`ultimateautomizer.py`). AUTOMIZER participates in all categories, which is also specified in its SV-COMP benchmark definition[9] file `uautomizer.xml`. In its role as witness validator, AUTOMIZER supports all categories except `ConcurrencySafety`, which is specified in the corresponding SV-COMP benchmark definition files `uautomizer-validate-*-witnesses.xml`.

References

1. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness Witnesses: Exchanging Verification Results between Verifiers. In *FSE 2016*, pages 326–337, 2016.
2. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness Validation and Stepwise Testification across Software Verifiers. In *ESEC/FSE 2015*, pages 721–733, 2015.
3. D. Beyer, S. Löwe, and P. Wendler. Refinement Selection. In *SPIN 2015*, pages 20–38, 2015.
4. D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski. Craig vs. Newton in Software Model Checking. In *ESEC/SIGSOFT FSE*, pages 487–497. ACM, 2017.
5. M. Greitschus, D. Dietsch, and A. Podelski. Loop Invariants from Counterexamples. In *SAS 2017*, pages 128–147, 2017.
6. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *CAV*, pages 36–52, 2013.

[4] `https://github.com/Z3Prover/z3`
[5] `https://cvc4.cs.nyu.edu/`
[6] `http://mathsat.fbk.eu/`
[7] `https://github.com/sosy-lab/sv-witnesses`
[8] `https://github.com/sosy-lab/benchexec`
[9] `https://github.com/sosy-lab/sv-comp`