Ultimate Automizer and the CommuHash Normal Form
(Competition Contribution)

Matthias Heizmann(✉), Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski

University of Freiburg, Freiburg im Breisgau, Germany
heizmann@informatik.uni-freiburg.de

Abstract. The verification approach of Ultimate Automizer utilizes SMT formulas. This paper presents techniques to keep the size of the formulas small. We focus especially on a normal form, called CommuHash normal form that was easy to implement and had a significant impact on the runtime of our tool.

1 Verification Approach

Ultimate Automizer (in the following called Automizer) is a software verifier that combines a CEGAR scheme and trace abstraction [6] to check safety and liveness properties.

Automizer's algorithm begins by transforming an input program to a program automaton whose transitions are labelled with formulas representing the effects of a statement (or multiple statements), whose accepting states correspond to error locations of the input program, and whose structure is equal to the structure of the control-flow graph of the input program. This program automaton recognizes a language, where every word is a sequence of statements that leads to an error location. If the language is empty, we can conclude that the program is safe. If the language is not empty, our algorithm picks a word from the language and checks whether it is *feasible* (i.e., the sequence of statements corresponds to an execution of the program) or *infeasible*. If the word is feasible we have found an actual counterexample. If it is infeasible we compute a proof of infeasibility for this sequence of statements. Afterwards we generalize this sequence of statements to a new automaton that accepts sequences of statements whose infeasibility can be shown by the very same proof. We then subtract the automaton with the language of infeasible words from the program automaton and obtain a new automaton that represents a smaller language, with which we continue the refinement loop. An important benefit of this approach is that because we perform the refinement step purely with automata operations, we never have to mix infeasibility proofs from different iterations.

This basic approach has not changed since the last competition. In the next section we explain improvements for the handling of SMT formulas.

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 577–581, 2023.
https://doi.org/10.1007/978-3-031-30820-8_39

2 SMT formulas in Ultimate

The ULTIMATE program analysis framework on which ULTIMATE AUTOMIZER is built upon, uses SMT formulas to represent the effect of program statements and to represent sets of states. We call formulas that represent sets of states *state assertions*. State assertions play a major role in the verification approach of AUTOMIZER. The infeasibility proof that we infer for each infeasible sequence of states is a sequence of state assertions and in the generalization step of the overall verification algorithm we have to check thousands of Hoare triples of the form $\{\varphi\}st\{\psi\}$, where φ and ψ are state assertions from infeasibility proofs. In order to check these Hoare triples, we reduce the validity problem for Hoare triples to a satisfiability problem for SMT formulas and let an SMT solver decide the satisfiability. The costs for the overall verification algorithm would be dominated by the costs for these satisfiability checks if we would not take additional actions to keep the size of the SMT formulas low.

We infer the sequence of state assertions by Craig interpolation or by a symbolic execution (via strongest post and weakest precondition) that is supported by unsatisfiable cores [3]. In the latter case the state assertions are usually quantified and we try to get rid of these quantifiers by applying several quantifier elimination techniques. These quantifier elimination techniques make the formulas simpler for SMT solvers but increase their size.

Our most powerful technique for reducing the size of formulas is an algorithm [4] that removes subformulas if the removal does not change the models of the formula. This algorithm however is itself costly because it calls an SMT solver for each subformula.

In order to also reduce the size of formulas without additional SMT solver calls, we utilize the following optimizations whenever we construct a formula.

- We apply the laws for annulment (e.g., $X \vee true$ becomes $true$), identity (e.g., $X \wedge X$ becomes X), idempotency (e.g., $x + 0$ becomes x), double negation (e.g., $\neg\neg X$ becomes X), and complement (e.g., $X \wedge \neg X$ becomes $false$).
- We compute the result for all operations on literals (e.g., $5 \leq 7\%2$ becomes $false$).
- We represent all integer and bitvector terms as polynomials. All terms that cannot be converted to polynomials become "variables" of the polynomial (e.g., $2 \cdot select(a, k) + 3 \cdot (x\%256) + 4$).
- For inequalities over integers and equalities over bitvectors and integers, we move monomials to the side of the relation where it can occur with a positive coefficient. (e.g., $2x - 3y = 0$ becomes $2x = 3y$).
- We work only with inequalities that open to the right. I.e., we transform $>$ to $<$, $\geq$ to $\leq$, *sgt* to *slt*, *sge* to *sle*, *ugt* to *ult*, and *uge* to *ule*.

3 The CommuHash Normal Form

An effect of the quantifier elimination techniques and the optimizations mentioned above is that we construct formulas in many places of our code. A side-effect of this is that we get formulas that have subformulas that differ only in the

order of the parameters of a commutative operator. E.g., we saw formulas like, e.g., $i = k \vee k \neq i$ or $a[i+k] = a[k+i]$. For both formulas the logical equivalence to *true* would have been detected if the operands of the commutative operations $+$ and $=$ would not have occurred in different orders. To minimize this problem we define a normal form that we call CommuHash Normal Form (CHNF). This normal form utilizes the fact that in ULTIMATE every formula has a 32-bit hash code. We say that an SMT formula is in *CommuHash Normal Form* if for every subformula with a commutative operator the operands are sorted according to their hash code in ascending order. To ensure that every formula is in CHNF ULTIMATE sorts the parameters whenever we construct a term whose operand is one of the following SMT operators: `=`, `distinct`, `and`, `or`, `xor`, `+`, `*`, `bvadd`, `bvmul`, `bvand`, `bvor`, `bvxor`.

In order to evalutate the effect of the CommuHash Normal Form we conducted an experiment in which we compared the default version of ULTIMATE AUTOMIZER to a version in which we disabled the sorting of parameters. We ran both versions on the benchmarks of the MemSafety category. In this category we typically have to deal with large formulas because the state assertions of proofs have to encode alias information about the program's pointers. We ran both versions on all 3440 benchmarks of the category. The CPU was an AMD Ryzen Threadripper 3970X, the time limit was 90s, the memory limit was 8000 MB and for each benchmark two CPU cores were used. In each run there were no incorrect results. The run without CHNF produced 1347 correct results, the run with CHNF produced 1439 correct results. Figure 1 shows a comparison of the runtimes for each benchmark in which at least one setting produced a result. We see that on average the run with CHNF needs less time. In fact on average the speedup is 31%.

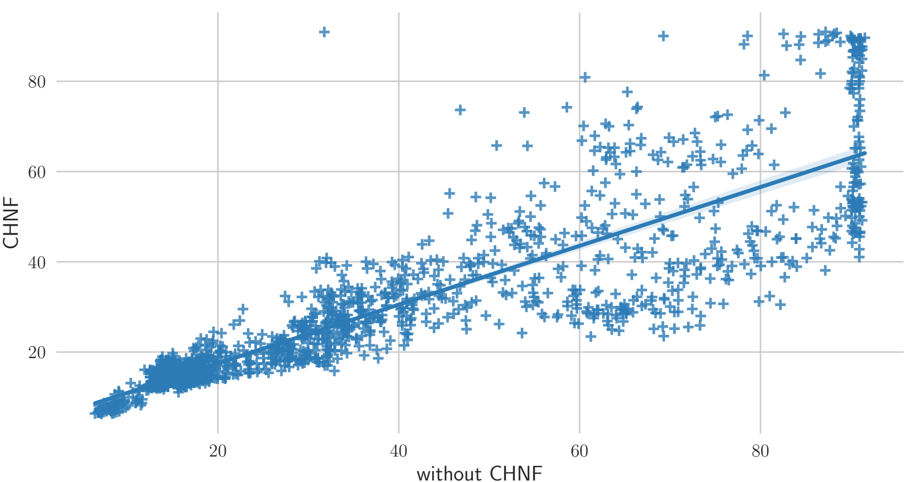

Fig. 1: Comparison of the runtime with and without CHNF

4 Project, Setup and Configuration

Automizer is a part of the open-source program analysis framework Ultimate[1]. Both are written in Java and licensed under LGPLv3. We use version 0.2.3 of Automizer [5] for SV-COMP, which requires Java 11 and Python 3.6. The release 0.2.3 contains binaries for Automizer and the SMT solvers Z3, CVC4, and Mathsat, as well as the Python wrapper script `Ultimate.py`. The Python script provides an interface to the competition environment, in particular to the BenchExec[2] tool-info module `ultimateautomizer.py`. Automizer also participates as witness validator and can validate violation [2] or correctness witnesses [1]. We participate in all categories [3] as verifier, but our witness validator does not yet support concurrency witnesses. Hence, our validator does not participate in `ConcurrencySafety` [4].

Automizer can be run 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 of verbose output to `stdout`. The witness that should be validated is specified with `--validate`. If Automizer generates a result, a witness is written to the file `witness.graphml`. Automizer's output is always written to the file `Ultimate.log`.

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 326–337. ACM (2016), https://doi.org/10.1145/2950290.2950351
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 721–733. ACM (2015), https://doi.org/10.1145/2786805.2786867
3. Dietsch, D., Heizmann, M., Musa, B., Nutz, A., Podelski, A.: Craig vs. newton in software model checking. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. pp. 487–497. ACM (2017), https://doi.org/10.1145/3106237.3106307

[1] https://github.com/ultimate-pa/ultimate
[2] https://github.com/sosy-lab/benchexec
[3] Specified by `uautomizer.xml` at https://github.com/sosy-lab/sv-comp.
[4] Specified by `uautomizer-validate-*-witnesses.xml`.

4. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: Cousot, R., Martel, M. (eds.) Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6337, pp. 236–252. Springer (2010), https://doi.org/10.1007/978-3-642-15769-1_15
5. Heizmann, M., Dietsch, D., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer SV-COMP 2023 Competition Contribution (Dec 2022), https://doi.org/10.5281/zenodo.7480181
6. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 36–52. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_2

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.