Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata
(Competition Contribution)

Matthias Heizmann(✉), Yu-Wen Chen, Daniel Dietsch, Marius Greitschus, Alexander Nutz, Betim Musa, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski

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

Abstract. Ultimate Automizer is a software verifier that implements an automata-based approach for the verification of safety and liveness properties. A central new feature that speeded up the abstraction refinement of the tool is an on-demand construction of *Floyd-Hoare automata*.

1 Verification Approach

Ultimate Automizer is a software verifier of the Ultimate program analysis framework[1]. The tool implements the automata-theoretic verification approach [3,4] that is outlined in Fig. 1 and is able to analyze reachability of error functions, memory safety, absence of overflows and termination. In this section, we briefly explain the overall algorithm and discuss a feature that speeded up the tool significantly, namely the on-demand construction of Floyd-Hoare automata, in detail.

1 $\mathcal{A}_0^{\mathsf{abs}}$:= constructCFA()
2 **for** $i = 0, 1, 2, \ldots$
3 **if** $(\mathcal{A}_i^{\mathsf{abs}} = \emptyset)$
4 **return** property holds
5 take error trace $\pi_i \in \mathcal{A}_i^{\mathsf{abs}}$
6 **if** (π_i is feasible)
7 **return** property violated
8 construct automaton $\mathcal{A}_i^{\mathsf{fh}}$ s.t. $\pi_i \in \mathcal{A}_i^{\mathsf{fh}}$ and $\mathcal{A}_i^{\mathsf{fh}}$ accepts only infeasible traces
9 $\mathcal{A}_{i+1}^{\mathsf{abs}} := \mathcal{A}_i^{\mathsf{abs}} \backslash \mathcal{A}_i^{\mathsf{fh}}$

Fig. 1. Overall verification algorithm

We initially construct an automaton, called control flow automaton (CFA), that resembles the control flow graph and whose acceptance condition reflects the property that is checked. E.g., for reachability problems, the *error location* of the program is the accepting state of the CFA. The alphabet Σ of the CFA consists of all program statements that occur in the control flow graph. We call a word over the alphabet Σ a *trace* and a word that is accepted by the CFA an *error trace*. The input program violates the given property if and only if there exists a feasible error trace, i.e., an error

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

A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 394–398, 2017.
DOI: 10.1007/978-3-662-54580-5_30

trace that corresponds to a real program execution. In our algorithm we construct automata $\mathcal{A}_i^{\mathsf{abs}}$ that overapproximate the set of feasible error traces. Our initial abstraction $\mathcal{A}_0^{\mathsf{abs}}$ is the CFA. All subsequent abstractions $\mathcal{A}_i^{\mathsf{abs}}$ are constructed in a CEGAR-style refinement loop (depicted in Fig. 1).

A central step of this algorithm is the construction of the automaton $\mathcal{A}_i^{\mathsf{fh}}$ in line 8. This automaton defines the set of (spurious) error traces that are eliminated in the current iteration. If this automaton accepts only few traces, the overall algorithm is more likely to diverge. For soundness, we require that $\mathcal{A}_i^{\mathsf{fh}}$ does not accept any feasible error trace. To account for that we construct $\mathcal{A}_i^{\mathsf{fh}}$ as a Floyd-Hoare automaton [4] which is a kind of automaton over the alphabet of program statements that accepts only infeasible traces. More details on the construction are given below.

Construction of Difference. In line 9 of the algorithm we construct a new abstraction for the set of feasible error traces. This new abstraction is an automaton $\mathcal{A}_{i+1}^{\mathsf{abs}}$ whose set of traces is the set-theoretic difference of the traces from the old abstraction $\mathcal{A}_i^{\mathsf{abs}}$ and the traces from the Floyd-Hoare automaton $\mathcal{A}_i^{\mathsf{fh}}$. The automaton $\mathcal{A}_i^{\mathsf{fh}}$ is deterministic and total. We construct $\mathcal{A}_{i+1}^{\mathsf{abs}}$ as the product automaton of $\mathcal{A}_i^{\mathsf{abs}}$ and $\mathcal{A}_i^{\mathsf{fh}}$ where a state of the product is accepting iff its first component is accepting and the second component is not accepting. In our implementation, we construct this product incrementally. We start with the initial state of the product and construct successively all reachable states and transitions. This allows us to construct the Floyd-Hoare automaton $\mathcal{A}_i^{\mathsf{fh}}$ on-demand as we explain next.

On-Demand Construction of Floyd-Hoare Automata . The input for the construction is a set of predicates Pred. We obtain this set by computing sequences of interpolants along infeasible error traces. Conceptually, the Floyd-Hoare automaton $\mathcal{A}_i^{\mathsf{fh}}$ is the automaton (defined in Fig. 2) whose states are the input predicates and all conjunctions of the input predicates. By construction, this automaton accepts only infeasible traces.

$$
\begin{aligned}
\mathcal{A}_i^{\mathsf{fh}} &:= (\Sigma, Q, \delta, q_0, Q_{\mathsf{fin}}) \\
Q &:= \{\textstyle\bigwedge P \mid P \in 2^{\mathsf{Pred}}\} \\
\delta(\varphi, st) &:= \textstyle\bigwedge\{p \in \mathsf{Pred} \mid \text{Hoare triple} \\
&\qquad\qquad \{\varphi\}st\{p\} \text{ is valid}\} \\
q_0 &:= \mathsf{true} \\
Q_{\mathsf{fin}} &:= \{\mathsf{false}\}
\end{aligned}
$$

Fig. 2. Definition of Floyd-Hoare automaton for i-th iteration

Usually, the automaton $\mathcal{A}_i^{\mathsf{abs}}$ is very sparse and hence only few transitions of $\mathcal{A}_i^{\mathsf{fh}}$ contribute to the difference operation (line 9). Since we construct the reachable state-space of the difference incrementally, we can construct the Floyd-Hoare automaton $\mathcal{A}_i^{\mathsf{fh}}$ on-demand. At the beginning, we construct only the initial state. Whenever the difference operation asks for successors of a state φ under a symbol st, we check if this transition was already added. If not, we compute the successor state and add transition and successor state if necessary. The successor state is the conjunction of all input predicates $p \in \mathsf{Pred}$ such that the Hoare triple $\{\varphi\}st\{p\}$ is valid.

Checking Hoare Triples Using a Cache and Unified Predicates. We can check Hoare triples using an SMT solver. However, these calls to an SMT solver can be costly and we try to reduce their number as follows. First, we keep a cache in which we store for each Hoare triple that has been checked so far whether it was valid or not. In order to have only one representative for logically equivalent predicates, we unify all predicates and all conjunctions of predicates that were constructed as states of the Floyd-Hoare automaton $\mathcal{A}_i^{\mathsf{fh}}$. In this unification process, we check for all pairs of formulas φ, ψ whether the implications $\varphi \models \psi$ and $\psi \models \varphi$ hold and store the results. If we now have to check the validity of a Hoare triple, we first check if one of the rules depicted in Fig. 3 is applicable. Only if none of these rules is applicable we use an SMT solver for the Hoare triple check.

$$\frac{\begin{array}{c}\varphi \models \varphi' \\ \psi' \models \psi \\ \{\varphi'\}st\{\psi'\} \text{ is valid}\end{array}}{\{\varphi\}st\{\psi\} \text{ is valid}}\ \mathsf{ImplPos} \qquad \frac{\begin{array}{c}\varphi \models \psi \\ \mathsf{vars}(\varphi) \cap \mathsf{write}(st) = \emptyset\end{array}}{\{\varphi\}st\{\psi\} \text{ is valid}}\ \mathsf{DataPos}$$

$$\frac{\begin{array}{c}\varphi' \models \varphi \\ \psi \models \psi' \\ \{\varphi'\}st\{\psi'\} \text{ is not valid}\end{array}}{\{\varphi\}st\{\psi\} \text{ is not valid}}\ \mathsf{ImplNeg} \qquad \frac{\begin{array}{c}\varphi \not\models \psi \\ \mathsf{vars}(\varphi) \cap \mathsf{read}(st) = \emptyset \\ \mathsf{vars}(\psi) \cap \mathsf{read}(st) = \emptyset \\ \mathsf{vars}(\psi) \cap \mathsf{write}(st) = \emptyset\end{array}}{\{\varphi\}st\{\psi\} \text{ is not valid}}\ \mathsf{DataNeg}$$

Fig. 3. Rules that allow us to infer validity of Hoare triples without calling an SMT solver. The set $\mathsf{vars}(\varphi)$ contains all variables that occur in the formula φ, the sets $\mathsf{read}(st)$ and $\mathsf{write}(st)$ contain all variables that are read (resp. written) by the statement st.

2 Software Architecture

Ultimate Automizer uses several SMT solvers. For the unification of predicates, the simplification of formulas and the Hoare triple checks we use Z3[2] because this solver can handle several SMT theories in combination with quantifiers. For the analysis of error traces we use CVC4[3], MathSAT[4], SMTInterpol[5], and Z3. These solvers each provide interpolants or unsatisfiable cores, which both can be used by Ultimate to extract predicates from infeasible traces. Furthermore, Ultimate Automizer uses several components of the Ultimate program analysis framework. The termination analysis is performed by the Buchi Automizer [5] component. This component requires

[2] https://github.com/Z3Prover.
[3] https://cvc4.cs.nyu.edu.
[4] http://mathsat.fbk.eu.
[5] https://ultimate.informatik.uni-freiburg.de/smtinterpol/.

ranking functions [6] and nontermination arguments [7] which are provided by LassoRanker[6]. LassoRanker uses SMTInterpol for the synthesis of ranking functions and Z3 for the synthesis of nontermination arguments. For our interprocedural analysis, we use nested word automata; in the termination analysis these automata have a Büchi acceptance condition. Data structures and algorithms for these automata are provided by the Automata Library. Ultimate also provides support for violation witnesses [2] and correctness witnesses [1]. Our competition candidate is able to produce and to validate both kinds of witnesses[7].

3 Tool Setup and Configuration

A zip archive that contains the tool and all above mentioned SMT solvers is available at the website of Ultimate Automizer[8]. The tool can be started by the following command,

```
./Ultimate.py prop.prp inputfile 32bit|64bit simple|precise
```

where `Ultimate.py` is a Python script, `prop.prp` the SV-COMP property file, and `inputfile` a C program. The other parameters determine the architecture and the memory model, respectively.

4 Software Project

The Ultimate program analysis framework is mainly developed at the University of Freiburg and received contributions from more than 50 people. The framework is written in Java and the source code is available on Github[9].

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: FSE. ACM (2016)
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: ESEC/FSE, pp. 721–733. ACM (2015)
3. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 471–482. ACM, New York (2010)
4. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_2

[6] https://ultimate.informatik.uni-freiburg.de/LassoRanker/.
[7] https://github.com/sosy-lab/sv-witnesses.
[8] https://ultimate.informatik.uni-freiburg.de/automizer/.
[9] https://github.com/ultimate-pa.

5. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_53
6. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 172–186. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54862-8_12
7. Leike, J., Heizmann, M.: Geometric nontermination arguments. CoRR, abs/1609.05207 (2016)