Ultimate Taipan: Trace Abstraction and Abstract Interpretation
(Competition Contribution)

Marius Greitschus(✉), Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski

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

Abstract. Ultimate Taipan is a software model checker for C programs. It is based on a CEGAR variant, trace abstraction [7], where program abstractions, counterexample selection and abstraction refinement are based on automata. Ultimate Taipan constructs path programs from counterexamples and computes fixpoints for those path programs using abstract interpretation. If the fixpoints are strong enough to prove the path program to be correct, they are guaranteed to be loop invariants for the path program. If they are not strong enough, Ultimate Taipan uses an interpolating SMT solver to obtain state assertions from the original counterexample, thus guaranteeing progress.

1 Verification Approach

Ultimate Taipan unifies the strengths of trace abstraction [7] and abstract interpretation [5]. Trace abstraction follows a counterexample-guided abstraction refinement (CEGAR) [4] approach for verifying programs. The initial abstraction is a program automaton constructed from the control flow graph (CFG) of the program. In the program automaton, accepting locations, called *error locations*, represent the violations of reachability properties of the program. Thus, the language accepted by the program automaton corresponds to all sequences of statements, i.e., to all *traces*, that lead to an error location. In each iteration, a trace τ is chosen from the current program automaton and analyzed for feasibility. If τ is feasible, it represents a concrete counterexample to the correctness of the program, as the error location is reachable. If τ is infeasible, a proof for its infeasibility is constructed. The proof is again encoded as an automaton, whose language consists of infeasible traces. Next, this automaton is *generalized* by adding transitions such that it accepts all traces of the program which are infeasible for the same reason as τ. This generalized automaton is then removed from the current program automaton by computing the automata-theoretic difference and the next iteration of trace abstraction starts. If the program automaton represents the empty language, i.e., if there are no traces of the program automaton left to analyze, the program is proven to be correct, because no error location is reachable.

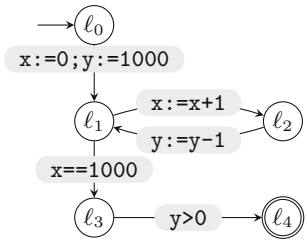
A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 399–403, 2017.
DOI: 10.1007/978-3-662-54580-5_31

The efficiency of this approach relies on the reasons for infeasibility of a trace. If the analyzed trace contains statements that are part of a loop, these reasons should ideally form an inductive loop invariant. If this is not the case, the trace abstraction algorithm *diverges*, i.e., the loop is unrolled until all traces of the loop have been excluded, either by discovering a suitable loop invariant, or by complete unrolling.

In Ultimate Taipan, we combat divergence by analyzing *path programs* [2] instead of traces [6]. A path program is a projection of the original program to the trace, i.e., a program in which only those statements occur that also occur in the trace. Hence, the path program may contain loops if the trace contains statements which are part of a loop. After choosing a trace of the program automaton, we construct a path program from the trace and use abstract interpretation to compute fixpoints for each path program location. If the fixpoint for an error location is *false*, this error location is unreachable and the computed fixpoints provide the proof of infeasibility for the whole path program, including the initially chosen trace. The advantage is that we are guaranteed to obtain inductive loop invariants if our abstract interpreter can prove the path program. In the case where abstract interpretation is not strong enough to prove infeasibility of the path program, we fall back on the classical analysis of single traces.

```
1  int x:=0, y:=1000;
2  while(*){
3    x:=x+1;
4    y:=y-1;
5  }
6  if (x==1000) {
7    assert y<=0;
8  }
```

(a) C code.



(b) Program automaton.

Fig. 1. Example C code and its corresponding program automaton. The location ℓ_0 of the automaton is the initial location, ℓ_4 is the error location.

Example. Consider the program and its corresponding program automaton in Fig. 1. In the program automaton, the violation of the assertion in line 7 is encoded by the accepting error location ℓ_4. Ultimate Taipan analyzes this program by first picking a counterexample trace starting in the initial location and ending in location ℓ_4, e.g. $\tau_1 =$ `x:=0;y:=1000` `x==1000` `y>0`. This trace is infeasible because the first and the second statement contradict each other. A state assertion capturing this fact is, e.g., $x = 0$. In the next step, an automaton recognizing all traces that are infeasible because of this fact is constructed and subtracted from the current program automaton.

In the next iteration, Ultimate Taipan's algorithm picks the trace $\tau_2 =$ `x:=0;y:=1000` `x:=x+1` `y:=y-1` `x==1000` `y>0`. Like τ_1, τ_2 is infeasible. The statements `x:=0`, `x:=x+1`, and `x==1000` contradict each other. This time,

the analysis of this single trace uses $x = 1$ as state assertion. If the algorithm continued in this fashion, it would need to unroll the loop completely, because the inductive loop invariant, which has to relate the variables `x` and `y`, would not be discovered. The reason for this is that the analysis of a single trace uses an interpolating SMT solver that favors "easier" and more concise state assertions over the relational one.

Hence, Ultimate Taipan uses a different method to extract state assertions. When analyzing τ_2, which contains statements that are part of a loop body, Ultimate Taipan constructs a path program from the trace and analyzes this path program with an abstract interpreter. Note that the path program corresponding to τ_2 is coincidentally the same as the program automaton. When using a relational abstract domain, e.g. octagons, in abstract interpretation, the state assertion $x \geq 0 \wedge y \leq 1000 \wedge x + y = 1000$ is found, which is an inductive loop invariant and thus suitable to prove unreachability of the error location in the path program. Therefore, when the path program is excluded from the program automaton, the resulting program automaton becomes empty and the program is proven to be safe.

2 Strengths and Weaknesses

Ultimate Taipan uses an abstract interpreter for proving infeasibility of traces containing loops. By not analyzing the whole program but a smaller path program, the imprecision that typically comes with abstract interpretation is mitigated and allows Ultimate Taipan to find inductive loop invariants in many cases. Because Ultimate Taipan needs to compensate for cases where the used abstract domain is not able to infer a proof of the path program, an interpolating SMT solver is still required. The combination of proofs obtained by the SMT solver with the proofs obtained from the abstract interpreter, e.g. during generation of correctness witnesses for the whole program, is expensive. In our current version, the computed fixpoints contain information about all variables in the path program, which leads to large SMT formulas.

3 Software Project

Ultimate Taipan is implemented on top of the program analysis framework Ultimate[1]. Nearly all components except the refinement algorithm and the abstract interpretation engine were already provided by Ultimate. We developed a new abstract interpretation plugin and integrated the refinement algorithm in the CEGAR loop of Ultimate Automizer. Ultimate provided the parsing back end and the verification condition generation as well as the construction of the control flow graph, the various automata operations, internal data structures for logic, the SMT solver SMTInterpol [3], and an interface to external SMT solvers compatible with the SMT-LIBv2 or v2.5

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

format. Like Ultimate, Ultimate Taipan is written in Java and the source code is available on GitHub[2]. Ultimate Taipan is licensed under LGPLv3[3].

4 Tool Setup and Configuration

Ultimate Taipan's website[4] provides a zip archive containing the competition submission. This archive contains an executable version of Ultimate Taipan for Linux platforms as well as the necessary theorem provers Z3[5] and CVC4[6]. Ultimate Taipan itself only requires a current Java installation (≥JRE 1.8).

The archive also contains a Python script, `Ultimate.py`, which maps the SV-COMP interface to Ultimate's command line interface and automatically selects the correct settings and the correct toolchain for Ultimate Taipan. This script requires a working Python 2.7 installation. In the SV-COMP scenario, the input to the script is a C program `input`, a property file `prop.prp`, the architecture setting `32bit` or `64bit`, and the memory model (either `simple` or `precise`) that should be assumed for the input file. Ultimate Taipan can be invoked with the following command.

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

The output of Ultimate Taipan is written to the file `Ultimate.log` and the result is written to `stdout`. When using the BenchExec[7] benchmarking framework to evaluate Ultimate Taipan, the output will automatically be translated by the tool-info module `ultimatetaipan.py` contained in BenchExec.

If the checked property does not hold, a human readable counterexample is stored in the file `UltimateCounterExample.errorpath` and a violation witness [1] is written to `witness.graphml`.

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: ESEC/FSE 2015, pp. 721–733 (2015)
2. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI 2007, pp. 300–309 (2007)
3. Christ, J., Hoenicke, J.: Cutting the mix. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 37–52. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21668-3_3
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)

[2] https://github.com/ultimate-pa/ultimate/.
[3] https://www.gnu.org/licenses/lgpl-3.0.en.html.
[4] https://ultimate.informatik.uni-freiburg.de/taipan.
[5] https://github.com/Z3Prover/z3.
[6] https://cvc4.cs.nyu.edu/.
[7] https://github.com/sosy-lab/benchexec.

5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252 (1977)
6. Greitschus, M., Dietsch, D., Podelski, A.: Refining Trace Abstraction using Abstract Interpretation. arXiv:1702.02369 [cs.LO] (2017)
7. 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