Ultimate Taipan with Dynamic Block Encoding

(Competition Contribution)

Daniel Dietsch(✉), Marius Greitschus(✉), Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, Andreas Podelski, Christian Schilling, and Tanja Schindler

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

Abstract. Ultimate Taipan is a software model checker that uses trace abstraction and abstract interpretation to prove correctness of programs. In contrast to previous versions, Ultimate Taipan now uses dynamic block encoding to obtain the best precision possible when evaluating transition formulas of large block encoded programs.

1 Verification Approach

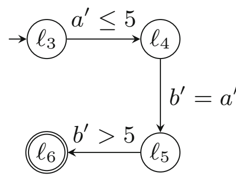
Ultimate Taipan (or Taipan for brevity) is a software model checker which combines trace abstraction [9,10] and abstract interpretation [5]. The algorithm of Taipan [8] iteratively refines an abstraction of a input program by analyzing counterexamples (cf. CEGAR [4]).

The initial abstraction of the program is an automaton with the same graph structure as the program's control flow graph, where program locations are states, transitions are labeled with program statements, and error locations are accepting. Thus, the language of the automaton consists of all traces, i.e., sequences of statements, that, if executable, lead to an error. In each iteration, the algorithm chooses a trace from the language of the current automaton and constructs a path program from it. A path program is a projection of the (abstraction of the) program to the trace. The algorithm then uses abstract interpretation to compute fixpoints for the path program. If the fixpoints of the path program are sufficient to prove correctness, i.e., the error location is unreachable, at least the chosen trace and all other traces that are covered by the path program are infeasible. The computed fixpoints constitute a proof of correctness for the path program and can be represented as a set of state assertions. From this set of state assertions, the abstraction is refined by constructing a new automaton whose language only consists of infeasible traces and then subtracting it from the current abstraction using an automata-theoretic difference operation. If abstract interpretation was unable to prove correctness of the path program, the algorithm obtains a proof of infeasibility of the trace using either interpolating SMT solvers or a combination of unsatisfiable cores and strongest post or weakest pre [6]. If the currently analyzed trace is feasible,

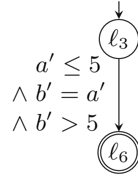
© The Author(s) 2018
D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 452–456, 2018.
https://doi.org/10.1007/978-3-319-89963-3_31

```
1  procedure foo() {
2    var a, b : int;
3    assume a <= 5;
4    assume b == a;
5    assert b <= 5;
6  }
```

(a) Program in Boogie.



(b) No block encoding.



(c) Large block encoding.

Fig. 1. Example program.

the trace represents a program execution that can reach the error. If the current automaton becomes empty after a difference operation, all potential error traces have been proven to be infeasible.

Dynamic Block Encoding. Large block encoding [1] is a technique to reduce the number of locations in a control flow graph. As Taipan relies on trace abstraction, the number of locations determines the performance of the automata operations, which impact the overall performance significantly. It is therefore beneficial to use a strong block encoding that removes as many locations as possible. Unfortunately, the resulting transitions can lead to a loss of precision during the application of an abstract post operator. Consider the example program and its control flow graph with different block encodings shown in Fig. 1. Each control flow graph consists of a set of program locations LOC, an initial location (ℓ_3 in Fig. 1), a set of error locations ($\{\ell_6\}$ in Fig. 1), and a transition relation $\rightarrow \subseteq LOC \times TF \times LOC$ which defines the transitions between the locations and labels each transition with a *transition formula* from the set of transition formulas TF. Transition formulas encode the semantics of the program as first-order logic formulas over various SMT theories. In Ultimate, a transition formula ψ is a tuple $(\varphi, IN, OUT, AUX, pv)$ where φ is a closed formula over the three disjoined sets of input (IN), output (OUT), and auxiliary (AUX) variables, and $pv : IN \cup OUT \rightarrow \mathcal{V}$ is an injective function that maps variables occurring in φ to program variables. We write output variables as primed variables and input variables as unprimed variables.

Taipan computes a fixpoint for each location of a control flow graph by (repeatedly) applying an abstract post operator $post^\#$ to these transition formulas. To this end, an abstract domain $\mathcal{D} = (A, \alpha, \gamma, \sqcup, \sqcap, \nabla, post^\#)$ is used, where A is a complete lattice representing all possible abstract states containing the designated abstract states $\top$ and $\bot$, α is an abstraction function, γ is a concretization function, $\sqcup$ is a join operator, $\sqcap$ is a meet operator, ∇ is a widening operator, and $post^\# : A \times TF \rightarrow A$ is an abstract transformer which computes an abstract post state σ' from a given abstract pre-state σ and a transition formula ψ. Taipan uses a combination of octagons [11] and sets of divisibility congruences [7] as abstract domain, but for brevity we explain the example using intervals.

In rows 1 to 3 of Table 1, we apply $post^\#$ of the interval domain in sequence to each of the transition formulas from Fig. 1b. In rows 4a and 4b we apply

Table 1. Application of $post^{\#}$ for transition formulas from Fig. 1.

	Pre-state	Transition formula	Post state
1	$\{a : \top, b : \top\}$	$a' \leq 5$	$\{a : [-\infty, 5], b : \top\}$
2	$\{a : [-\infty, 5], b : \top\}$	$b' = a'$	$\{a : [-\infty, 5], b : [-\infty, 5]\}$
3	$\{a : [-\infty, 5], b : [-\infty, 5]\}$	$b' > 5$	$\{a : [-\infty, 5], b : \bot\}$
4a	$\{a : \top, b : \top\}$	$a' \leq 5 \wedge b' > 5 \wedge b' = a'$	$\{a : [-\infty, 5], b : \bot\}$
4b	$\{a : \top, b : \top\}$	$b' = a' \wedge a' \leq 5 \wedge b' > 5$	$\{a : [-\infty, 5], b : [-\infty, 5]\}$

the same operator to the only transition formula of Fig. 1c, but process the conjunction in different orders. Although the logical $\wedge$-operator is commutative, the result differs. This is due to different ways of computing the abstract post state. We can express $post^{\#}(\sigma, A \wedge B) = \sigma'$ either as $post^{\#}(\sigma, A) \sqcap post^{\#}(\sigma, B)$, as $post^{\#}(post^{\#}(\sigma, A), B)$, or as $post^{\#}(post^{\#}(\sigma, B), A)$. The interval domain cannot express the equality relation between two variables (i.e., the conjunct $b' = a'$), therefore, the first way will compute $post^{\#}(\{a : \top, b : \top\}, b' = a') = \{a : \top, b : \top\}$, effectively rendering the constraint useless. The second and third way may succeed, depending on the ordering of conjuncts. In general, the ordering is important, but in our example, it does not matter as long as $b' = a'$ is not first.

In Taipan, we solve this problem by introducing the notion of *expressibility* to an abstract domain. We augment each abstract domain with an expressibility predicate *ex* which decides for each non-logical symbol of a transition formula (i.e., each relation, function application, variable, and constant) whether it can be represented in the domain. For example, the interval domain can represent all relations that contain at most one variable, while octagons can represent all relations of the form $\pm x \pm y \leq c$. We then apply $post^{\#}$ on conjuncts of a transition formula in an order induced by *ex*, thus effectively choosing a new *dynamic* block encoding. For $post^{\#}(\sigma, \varphi)$, our algorithm computes σ' by first converting the formula φ to DNF s.t. $\varphi = \varphi_0 \vee \varphi_1 \vee \ldots \vee \varphi_n$. For each disjunct $\varphi_i = \varphi_i^0 \wedge \varphi_i^1 \wedge \ldots \wedge \varphi_i^m$, we compute $post^{\#}(\sigma, \varphi_i) = \sigma_i'$ as follows:

1. Partition the conjuncts in two classes. The first class contains conjuncts for which *ex* is true, the second for which *ex* is false.
2. Compute the abstract post for the conjunction of all expressible conjuncts first: $\bigsqcap_{ex(\varphi_i^k)} post^{\#}(\sigma, \varphi_i^k) = \sigma''$.
3. Compute the abstract post for all non-expressible conjuncts successively using the post state of the k-th application as pre-state of the $k+1$-th application, and the post state of the last application as final result σ_i' for the disjunct φ_i: $post^{\#}_{\neg ex(\varphi_i^k)}(\sigma_k, \varphi_i^k) = \sigma_{k+1}$.

The result for $post^{\#}(\sigma, \psi)$ is then $\bigsqcup_{i=0}^{n} \sigma_i' = \sigma'$.

2 Project, Setup and Configuration

Taipan is a part of the open-soure program analysis framework Ultimate[1], written in Java, licensed under LGPLv3[2], and open source[3]. The Taipan competition submission is available as a zip archive[4]. It requires a current Java installation (≥JRE 1.8) and a working Python 2.7 installation. The submission contains an executable version of Taipan for Linux platforms, the binaries of the required SMT solvers Z3[5], CVC4[6], and Mathsat[7], as well as a Python script, `Ultimate.py`, which maps the SV-COMP interface to Ultimate's command line interface and selects the correct settings and the correct toolchain. In SV-COMP, Taipan is invoked through `Ultimate.py` with

```
./Ultimate.py --spec prop.prp --file input.c --architecture
              32bit|64bit --full-output
```

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 complete output of Taipan is also written to the file `Ultimate.log`. Depending on the status of the property, a violation [3] or correctness [2] witness may be written to the file `witness.graphml`.

The benchmarking tool BenchExec[8] supports Taipan through the tool-info module `ultimatetaipan.py`. Taipan participates in all categories, as specified by its SV-COMP benchmark definition[9] file `utaipan.xml`.

References

1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD 2009, pp. 25–32. IEEE (2009)
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: FSE 2016, pp. 326–337 (2016)
3. 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)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15

[1] https://ultimate.informatik.uni-freiburg.de.
[2] https://www.gnu.org/licenses/lgpl-3.0.en.html.
[3] https://github.com/ultimate-pa/ultimate/.
[4] https://ultimate.informatik.uni-freiburg.de/downloads/svcomp2018/UltimateTaipan-linux.zip.
[5] https://github.com/Z3Prover/z3.
[6] https://cvc4.cs.nyu.edu/.
[7] http://mathsat.fbk.eu/.
[8] https://github.com/sosy-lab/benchexec.
[9] https://github.com/sosy-lab/sv-comp.

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. Dietsch, D., Heizmann, M., Musa, B., Nutz, A., Podelski, A.: Craig vs. Newton in software model checking. In: ESEC/FSE 2017, pp. 487–497 (2017)
7. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Abramsky, S., Maibaum, T.S.E. (eds.) CAAP 1991. LNCS, vol. 493, pp. 169–192. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53982-4_10
8. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 128–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_7
9. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_7
10. 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). https://doi.org/10.1007/978-3-642-39799-8_2
11. Miné, A.: The Octagon Abstract Domain. CoRR, abs/cs/0703084 (2007)

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.