JTACO: Test Execution for Faster Bounded Verification

Alexander Kampmann[2], Juan Pablo Galeotti[1], and Andreas Zeller[1]

[1] Software Engineering Chair, Saarland University, Saarbrücken, Germany
lastname@cs.uni-saarland.de
[2] Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
kampmann@st.cs.uni-saarland.de

Abstract. In bounded program verification a finite set of execution traces is exhaustively checked in order to find violations to a given specification (i.e. errors). SAT-based bounded verifiers rely on SAT-Solvers as their back-end decision procedure, accounting for most of the execution time due to their exponential time complexity.
In this paper we sketch a novel approach to improve SAT-based bounded verification. As modern SAT-Solvers work by augmenting partial assignments, the key idea is to *translate* some of these partial assignments into JUNIT test cases during the SAT-Solving process. If the execution of the generated test cases succeeds in finding an error, the SAT-Solver is promptly stopped.
We implemented our approach in JTACO, an extension to the TACO bounded verifier, and evaluate our prototype by verifying parameterized unit tests of several complex data structures.

1 Introduction

Bounded verification [5] is a fully automatic verification technique. Given a program P and its specification $\langle Pre, Post\rangle$, a bounded verification tool exhaustively checks correctness for a finite set of executions. In order to constrain the number of program executions to be analyzed, the user selects a scope of analysis by choosing: (a) a bound to the size of domain (e.g., `LinkedList`, `Node`, etc.), and (b) a limit to the number of loop unrollings or recursive calls.

Bounded verification tools [3,5,8,10,12,16] rely on translating P, precondition Pre and postcondition $Post$ into a propositional formula ψ such that

$$\psi = Pre \wedge P \wedge \neg Post.$$

If an assignment of variables exists such that ψ is true, ψ is *satisfiable*, and the satisfying assignment represents an execution trace violating the specification $\langle Pre, Post\rangle$. On the other hand, if ψ is *unsatisfiable* (i.e. there is no satisfying assignment for ψ), the specification holds within the user-selected scope of analysis. However, a violation might still be found if a greater scope of analysis is chosen. In order to decide on the satisfiability of ψ, the bounded verifier relies on a SAT-Solver, a program specialized in solving the satisfiability problem for propositional formulas.

```
public static void testRemove(int v1, int v2, int v3) {
  BinarySearchTree t = new BinarySearchTree();
  t.add(v1);
  t.add(v2);
  t.add(v3);
  assert t.find(v2);
  t.remove(v2);   // should remove all occurrences
  assert !t.find(v2);
}
```

Fig. 1: A parameterized unit test for a binary search tree class.

TACO [8] targets the bounded verification of sequential Java programs. For example, for the parameterized unit test shown in Figure 1, TACO will search for values for the integer parameters `v1`, `v2` and `v3` to falsify any of the assertions. Apart from checking regular `assert` statements, TACO also verifies more complex program specifications written in behavioural formal languages such as JML [2] or JFSL [17]. Although TACO is specially tailored for verifying complex specifications in linked-data structures (such as the well-formedness of red-black trees), the burden of writing such specifications is by no means small. As a light-weight alternative, applying bounded verifiers to parameterized unit tests might still help finding errors, but requires less effort from the user.

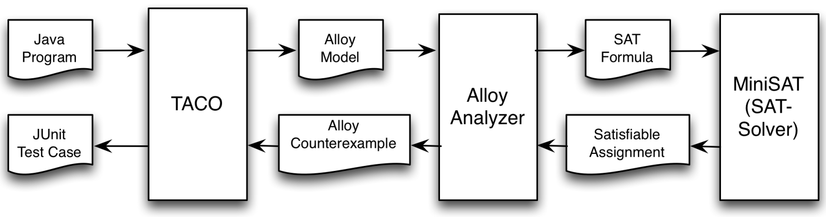

Fig. 2: A high-level view of the TACO architecture

Figure 2 presents a high-level overview of the TACO architecture. In order to translate the Java program into a propositional formula ψ, TACO uses the ALLOY language [11] as an intermediate representation. The analysis starts when the target Java program and its specification are translated into an ALLOY model. The ALLOY analyzer is then invoked to check the correctness of the model. This is done by translating the ALLOY representation into a propositional formula that is later solved using the MINISAT SAT-Solver. In case MINISAT [7] finds a solution to ψ, the satisfying assignment is returned to the ALLOY analyzer, that builds an ALLOY instance as a counterexample to the violated property. Finally, TACO translates the ALLOY counterexample into a JUNIT test case for later inspection by the user.

Given n propositional variables, there are 2^n possible assignments of values to those variables. SAT-Solvers are programs designed for efficiently deciding the satisfiability of a formula (i.e., either it is satisfiable or it is unsatisfiable). Nevertheless, as the worst-case time complexity of SAT is exponential on the number of propositional variables in ψ, it is often the case that most of the verification budget is spent in the execution of the SAT-Solver. On many occasions, when the time bound or the resources at hand are exhausted, the verification effort has to be cancelled. This leads to the unpleasant situation that significant computational resources might have been spent, while the user obtained no feedback from that investment.

In previous work [6], we already explored the idea of profiting from observing the internal state of the SAT-Solver during its execution. By assuming the ψ is unsatisfiable we approximate an UNSAT core [15] by measuring the activity of the SAT-Solver during the progress of the SAT-Solving process. In this paper, we aim at optimizing the bounded verifier when the underlying ψ is satisfiable. More specifically, we try to *lift* a partial assignment collected from the SAT-Solver into a JUNIT [1] test case. If the execution of the JUNIT test case leads to a violation of the specification, the whole SAT-Solver could be stopped. The intuition behind this is that in some cases the SAT-Solving process might not be as performant as executing the code in order to check the validity of ψ.

The contributions of this article include:

- An approach to combine SAT-Solving and JUNIT test case execution based on monitoring the internal state of the SAT-Solver.
- JTACO, an extension to the TACO bounded verifier implementing the aforementioned approach.
- An evaluation of the JTACO approach on a small benchmark of parameterized unit tests handling several complex data structures.

2 From Partial Assignments to JUNIT test cases

Most modern SAT-Solvers (like MINISAT) are based on variants of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [4]. The DPLL algorithm maintains and extends a partial assignment of the propositional variables to binary values. Each propositional variable can be *assigned* (meaning the algorithm has determined a provisional binary value for this variable), or *unassigned*.

In Figure 3 we show the pseudocode of the DPLL algorithm. It starts by calling procedure `search_new_value()` to extend the current assignment by deciding a new binary value for an unassigned variable. The function `propagate()` applies boolean constraint propagation (BCP) until no more values for variables can be inferred or the current decisions led to an unsatisfiable clause (namely, a *conflict*). The function `analyze_conflict()` determines the set of variable assignments that implied the conflict, returning the highest level of decision for all the variables involved (i.e., the `conflict_level` variable). Finally, `backtrack()` undoes the current assignment to the conflict level. This is usually referred to as *back jumping*.

Any partial assignment that led to a conflict during the `propagate()` phase is known to be unsatisfiable. In our previous work [13], we observed that only a fraction of the propositional variables of ψ are actually modelling the initial state of the execution trace. This is due to the fact that, given the *static* nature of the ALLOY language, TACO

```
1  while (true) {
2    search_new_value();
3    propagate();
4    if (status==CONFLICT) {
5      conflict_level = analyze_conflict();
6      if (conflict_level==0) {
7        return UNSAT;
8      } else {
9        backtrack(conflict_level);
10     }
11   } else if (status==SAT) {
12     return SAT;
13   }
14 }
```

Fig. 3: A sketch of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm

models field and variable updates by introducing several versions of the same variable (much like in a SSA-like form[3]).

JTACO extends MINISAT by dumping the partial assignment whenever a conflict occurs (line 5 of Figure 3). Then, JTACO tries to *lift* the obtained partial assignment to a JUNIT test case. In order to lift a partial assignment v, JTACO first removes all those variables in v that are not used to model the initial state of the execution trace. However, the resulting filtered partial assignment might be insufficient for generating a JUNIT test case if any of the propositional variables modelling a given initial value is missing. For example, consider the argument `v1` of the parameterised unit test shown in Figure 1. If k propositional variables model the initial value of `v1` (namely $v1_0, v1_1, \ldots, v1_{k-1}$)[4], we need to know the value of all the propositional variables to conclude the initial value of `v1`.

```
@Test
public void test1() {
  int int0 = -8;
  int int1 = -8;
  int int2 = 0;
  try {
    BinarySearchTree.testRemove(int0, int1, int2);
  } catch (AssertionError err) {
    fail("An assertion did not hold:" + err);
  }
}
```

Fig. 4: A failing JUNIT test case generated by JTACO.

[3] In the Single Static Assignment (SSA) form, each variable is assigned exactly once.

[4] Alloy encodes integer values using two's complement.

If all the propositional variables encoding the initial state (i.e., arguments and field values) are assigned, then JTACO writes a JUNIT test case by decoding the values from the partial assignment (as the one shown in Figure 4). Subsequently, JTACO compiles the written test case and executes it. By construction, if the test case fails then the initial state led to a violation of the specification (in this case, failing the `assert` statement in line 8 of Figure 1). We refer to the original partial assignment that led to a failing test case as a *failing* partial assignment. The importance of finding such a partial assignment relies on the fact that, as soon as it is detected, the entire SAT-Solving process can be stopped.

To conclude, whenever a conflict occurs during the execution of the MINISAT's DPLL algorithm, JTACO filters those variables from the partial assignment that are not modelling initial values of the execution trace. If all the variables of the initial state are assigned in the resulting filtered partial assignment, a JUNIT test case is written, compiled and executed. If the JUNIT test case fails, then a violation to the specification has been found, and the verification process is stopped. In any other scenario (i.e., unassigned initial variables, JUNIT execution succeeding) the DPLL algorithm resumes.

3 Evaluation

We ran our prototype of JTACO on an Intel Core Duo T6600 with a scope of analysis of 4 elements per domain and 3 loop unrollings. We selected a benchmark of container classes taken from [8]. Since all these classes have already been verified using TACO, we wrote 9 additional faulty parameterized unit tests that are expected to fail (like asserting an AVL tree is empty after an insertion). Additionally, we add a faulty implementation of binary search trees plus a parameterized unit test capable of exhibiting the failure. The rationale behind these decisions is twofold: first, by aiming at the case where the formula ψ is satisfiable, we focus on the scenario in which our approach could make gains. Secondly, by resorting to parameterized unit tests (instead of regular JML or JFSL specification violations) we avoid the problem of synthesizing runtime predicates for asserting the validity of the specifications.

We evaluate our approach by addressing the following research questions:

- **RQ1:** During the MINISAT execution, when does the first failing partial assignment occur (i.e. the first partial assignment leading to a failing JUNIT test case)?
- **RQ2:** Does JTACO outperform TACO in terms of execution time?

Experimental Results: Figure 5 presents the average results of 10 executions of TACO and JTACO on the subjects of the selected case study. The first and second column list the subject name and the parameterized unit test. The third column shows the average time for verifying the program using TACO (only MINISAT time is considered). The fourth column presents the point in time when the first failing partial assignment was found during the MINISAT execution. The fifth column shows the total JTACO time (i.e. MINISAT time plus the time for collecting partial assignments, lifting them to JUNIT test cases and executing them). The last column presents the speed-up of JTACO with respect to TACO.

Subject	parameterized Unit Test	TACO time (ms)	1st Failing Partial Assignment (ms)	JTACO time (ms)	Speed-up
SinglyLinkedList	testRemove	144.7	81.1 (56.0%)	1545.7	0.09x
DoublyLinkedList	testRemove	496.2	202.4 (40.0%)	1940.3	0.25x
	testIndexOf	167.8	17.7 (10.0%)	408.4	0.41x
NodeCachingLinkedList	testIndexOf	1065.5	16.7 (1.5%)	150.5	7.07x
	testRemove	1823.9	20.7 (1.1%)	172.6	10.56x
BinaryTree	testRemove	567.4	7.9 (1.3%)	101.0	5.61x
BuggyBinaryTree	parameterized	315.2	10.4 (3.3%)	176.9	1.78x
	parameterizedSmall	136.5	9.0 (6.5%)	163.0	0.83x
AvlTree	testFindMax	3886.3	1072.5 (27.0%)	12928.9	0.30x
	testIsEmpty	2743.9	16.0 (0.5%)	146.2	18.76x
BinaryHeap	testFindMinDecrease	679.3	10.7 (1.5%)	100.0	6.76x

Fig. 5: Average execution times for 10 runs of TACO and JTACO on the benchmark

In all cases the failing partial assignment occurs very early during the MINISAT execution: ranging from 0.5% to 56% of the total solver time. Under these circumstances, the fundamental idea of lifting these partial assignments into JUNIT test cases seems validated.

Regarding the performance of TACO and JTACO, surprisingly JTACO is outperformed on half of the subjects. A closer inspection revealed that, although a failing partial assignment was indeed found very quickly, the MINISAT process was mostly delayed by JTACO's generation and execution of test cases. In other words, the cost of generating and executing JUNIT test cases was higher than the benefit of stopping the SAT-Solver execution sooner, at least in a single-core environment. Observe that JTACO was faster if the first failing partial assignment occurred within the first 5% of the MINISAT execution time.

4 Conclusions and Further Work

Given the exponential complexity of the SAT-solving process, techniques for decreasing the execution time of bounded verification tools are paramount. In this work we presented an approach for generating JUNIT test cases from partial assignments collected during the SAT-Solver execution.

Besides general improvements such as robustness and maturity for the JTACO tool, our future work will focus on the following issues:

- **Runtime-checking of JML/JFSL specifications:** The current prototype of JTACO only handles properties that are expressed as `assert` statements. In order to fully support JML and JFSL specifications we need to automatically synthesize runtime predicates for a significant fragment of the specification language (e.g., handling the runtime-checking of all the constructs used in [8]). Additionally, we also need to extend the lifting mechanism (introduced in Section 2) for handling initial values that do not satisfy the precondition of the method under analysis.

– **Bounded verification of correct programs:** If ψ happens to be unsatisfiable, all the overhead invested in dumping partial assignments is lost. Due to this fact, JTACO could become a practical approach only if the SAT-Solver's overhead is reasonable in case the formula is unsatisfiable.
– **A multi-core JTACO:** In a multi-core environment, several processes could be spawned for lifting different partial assignments without blocking the main MINISAT process. We expect that a multi-core JTACO might boost JTACO performance. Our future work will include a comparison against other parallel SAT-solving tools [9,14].

The current prototype of the JTACO tool, as well as all subjects required to replicate the results in this paper are publicly available. For details, see:

`http://www.st.cs.uni-saarland.de/jtaco/`

Acknowledgments

This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing" and MEALS 295261. Eva May provided helpful comments about this work. We thank the anonymous reviewers for their comments and suggestions.

References

1. Bech, K., Gamma, E.: JUnit: A programmer-oriented testing framework for Java (May 2014), http://junit.org
2. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 342–363. Springer (2005)
3. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
4. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
5. Dennis, G., Yessenov, K., Jackson, D.: Bounded verification of voting software. In: Shankar, N., Woodcock, J. (eds.) VSTTE. Lecture Notes in Computer Science, vol. 5295, pp. 130–145. Springer (2008)
6. D'Ippolito, N., Frias, M.F., Galeotti, J.P., Lanzarotti, E., Mera, S.: Alloy+HotCore: A fast approximation to unsat core. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ASM. Lecture Notes in Computer Science, vol. 5977, pp. 160–173. Springer (2010)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
8. Galeotti, J.P., Rosner, N., Pombo, C.L., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Tonella, P., Orso, A. (eds.) ISSTA. pp. 25–36. ACM (2010)
9. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT 6(4), 245–262 (2009)

10. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV. Lecture Notes in Computer Science, vol. 3576, pp. 301–306. Springer (2005)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis (Revised Edition). The MIT Press (2012)
12. Near, J.P., Jackson, D.: Rubicon: bounded verification of web applications. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) SIGSOFT FSE. p. 60. ACM (2012)
13. Parrino, B.C., Galeotti, J.P., Garbervetsky, D., Frias, M.F.: TacoFlow: optimizing sat program verification using dataflow analysis. SoSyM: Software and Systems Modeling (2014)
14. Rosner, N., Galeotti, J.P., Bermúdez, S., Blas, G.M., Rosso, S.P.D., Pizzagalli, L., Zemín, L., Frias, M.F.: Parallel bounded analysis in code with rich invariants by refinement of field bounds. In: Pezzè, M., Harman, M. (eds.) ISSTA. pp. 23–33. ACM (2013)
15. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM. Lecture Notes in Computer Science, vol. 5014, pp. 326–341. Springer (2008)
16. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst. 29(3) (2007)
17. Yessenov, K.: A light-weight specification language for bounded program verification. Master's thesis, MIT (2009)