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.

M. Seidl and N. Tillmann (Eds.): TAP 2014, LNCS 8570, pp. 134–141, 2014.

```
1 public static void testRemove(int v1, int v2, int v3) {
2     BinarySearchTree t = new BinarySearchTree();
3     t.add(v1);
4     t.add(v2);
5     t.add(v3);
6     assert t.find(v2);
7     t.remove(v2);       // should remove all occurrences
8     assert !t.find(v2);
9 }
```

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.

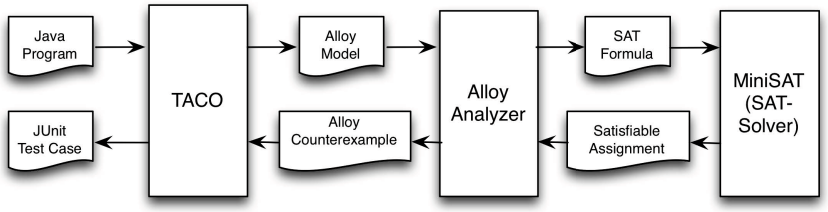

Fig. 2. A high-level view of the TACO architecture

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-