FramewORk for Embedded System verificaTion (Competition Contribution)

Pablo Gonzalez-de-Aledo and Pablo Sanchez

University of Cantabria, Santander (Cantabria), Spain
{pabloga,sanchez}@teisa.unican.es

Abstract. Forest is a bounded model checker that implements symbolic execution on top of the LLVM intermediate language and is able to detect errors in programs developed in C. Forest transforms a program into a set of SMT formulas describing each feasible path and decides these formulas with an SMT solver. This enables it to prove the satisfiability of reachability conditions such as the ones presented in SV-COMP. Forest implements different ways of representing SMT formulas: linear arithmetic, polynomials and generic bit-accurate and not bit-accurate representations.

1 Overview

As many bounded model checkers, to verify a property for a given piece of code, Forest unfolds the execution of the code up to a certain depth and transforms each path into an SMT formula. Before this transformation, assertions and special functions are converted into conditions, so verification clauses can be expressed as reachability properties (in the SV-COMP framework, if a state can be reached from the start of the main procedure in which an LTL clause can be satisfied, then the program is unsafe). The transformation from source to SMT can be done using different theories (integers, linear formulas, polynomials, etc.), and formulas can be decided using different solvers (Boolector, Z3, CVC4 ...). For the competition, the theory of integers and real numbers has been chosen, and formulas are decided with Z3 [3]. This is a trade-off between accuracy and solving time.

2 Architecture

As a framework for automated program verification through symbolic execution, verification under Forest comprises the following steps, which are illustrated in Figure 1.

1. **Configuration:** The 'forest' binary orchestrates the remaining tools and steps, and configures the framework according to command-line parameters or configuration files (xml files).

DOI: 10.1007/978-3-662-46681-0_36

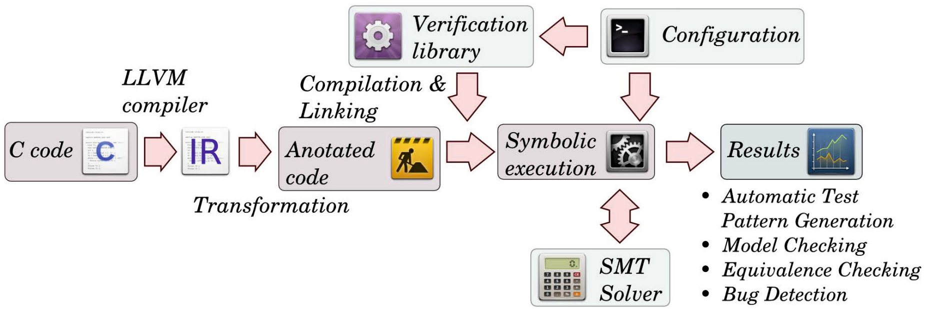

Fig. 1. FOREST architecture

2. **Front-End and intermediate representation:** As a front-end we use llvm-gcc, which transforms the source code to an llvm intermediate representation. In this representation, branch instructions are performed in two steps; first the result of the comparison is stored in a register. This register is then used in a jump instruction to implement the branch. This unifies comparisons so they can be handled as binary instructions.
3. **Annotation:** The intermediate representation is transformed via a transformation pass that instruments every operation with calls to back-end functions. These back-end functions dynamically compute the strongest post-condition from the `'start'` state for every instruction so the effect of this instruction in the state can be considered when running the program.
4. **Static Heuristic:** The control-flow-graph of the intermediate representation is obtained and a heuristic is computed indicating possible paths from the entry point of the program to the destination. Yen's algorithm [1] is used to compute the k-shortest paths from entry to error location.
5. **Linking:** The transformed intermediate representation is linked with a verification library. This library implements the semantics of every operation in the intermediate representation and performs the symbolic execution as explained in the following step.
6. **Execution:** When executed, the program forks on every condition encountered in execution and the heuristic computed in step 4 is used to guide the exploration toward the error location. The A* algorithm [4] is used to search for paths between entry and error. While the program is run, the inserted functions from step 3 compute the strongest post-condition from the starting state, and this condition is passed to an SMT solver when a conditional branch is encountered. The effect of forking the execution on every branch instruction is that the program "unfolds" into a binary tree in which every process executes a different feasible path. Feasible paths are then added to A* set of candidate paths to continue exploration.

3 Strenghts and Weaknesses

As a bounded model checker, FOREST cannot generate proofs of correctness for unbounded programs. In these cases, we unfold the loops up to a certain depth, and check for satisfiability in an under-approximation of the program possible behaviours. This may be unsound in certain benchmarks such as `array_call3`, where FOREST fail to detect the error due to this limitation. Orthogonally to this problem, approximating the behavior of variables with integers and real types can also produce errors. This happens in the test 'verisec_sendmail', in which the reachability of the `error` state depends on an integer overflow. This bug is not detected using integer representation but can be spotted if we use the option `-solver bitvector`. The strengths of symbolic execution are its applicability in a wide spectrum of applications, the possibility of obtaining partial results and the speed of finding bugs when the program has some.

4 Tool Setup

The version of FOREST submitted to the competition can be downloaded executing the following command in a `x86_64` Linux machine

```
wget teisa.unican.es/forest/images/install.sh -O - | bash
```

This should download and execute a script that installs the tool in the current path and performs some tests. A correct installation can be assessed if all tests are correct and terminate in time. The command-line options to be used in SV-COMP have been condensed to the '`-svcomp`' parameter. The file to analyse can be indicated with '`-file`'. Complete installation instructions can be obtained removing the tailing '`| bash`' from the previous command.

5 Software Project

FOREST is maintained by Pablo González de Aledo. This work has been supported by Project TEC2011-28666-C04-02 and grant BES-2012-055572, awarded by the Spanish Ministry of Economy and Competitivity. We gratefully acknowledge the help of Franck Cassez for the revision of this article and for his pertinent advice, and Fernando Herrera for testing the tool under different Linux distributions and machines.

References

1. Yen, J.Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. The Quarterly of Applied Mathematics 27, 526–530
2. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization 2004, pp. 75–86 (2004)
3. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics SSC4 4, 100–107 (1968)