Run Forester, Run Backwards! (Competition Contribution)

Lukáš Holík[1], Martin Hruška[1], Ondřej Lengál[2],
Adam Rogalewicz[1], Jiří Šimáček[1], and Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan

Abstract. This paper briefly describes the Forester tree automata-based shape analyser and its participation in the SV-COMP'16 competition on software verification. In particular, it summarizes the verification approach used by Forester, its architecture and setup for the competition, as well as its strengths and weaknesses observed in the competition run. The paper highlights the newly added counterexample validation and use of refinable predicate language abstraction.

1 Verification Approach

Forest Automata. Forester implements a fully automated and sound *shape analysis* based on the notion of *forest automata* (FAs) [1]. FAs can represent sets of reachable configurations of programs with complex dynamic linked data structures (such as various kinds of lists, trees, skip lists, as well as combinations of such data structures). They have a form of tuples of *tree automata* (TAs). These tuples of TAs encode sets of heap graphs decomposed into tuples of *tree components*, whose leaves may refer back to the roots of the components (including roots of other components). The decomposition is based on cutting a heap graph at each *cut-point*, i.e., a node which is either pointed by some pointer variable or which has multiple incoming pointer edges.

In order to encode complex heap graphs, FAs may be *hierarchically structured* in such a way that a higher-level FA may use other, lower-level FAs as alphabet symbols. These nested automata, called *boxes*, encode *repetitive graph patterns* and can be automatically learned using the approach proposed in [2].

In order to be as efficient as possible, Forester never determinises the TAs it works with. All needed operations, including inclusion checking and size reduction, are therefore implemented on *non-deterministic TAs*. For that, techniques such as antichain-based inclusion checking and simulation-based reduction are used.

Counterexample Analysis and Refinement. In Forester, FAs are used within the framework of *abstract regular tree model checking* (ARTMC) [3]. ARTMC accelerates the computation of sets of reachable program configurations, represented by FAs, by abstracting their component TAs, which is done by collapsing some of their states.

For deciding which TA states should be collapsed when performing ARTMC, multiple approaches have been proposed in the literature [3]. When Forester first participated in SV-COMP in 2015, it supported the simplest of these approaches based on collapsing states accepting the same *languages of trees up to some height* only. No checking of validity of counterexamples and no abstraction refinement was implemented then.

In the version of Forester participating in SV-COMP'16, an approach for checking validity of counterexamples was added. It is based on a *backward execution* of the program being verified along the suspected counterexample. For that, it was needed to add a support for *reverse execution* of all program statements over FAs. Moreover,

a support for *intersection of FAs*, not needed before, had to be added. Intersection of FAs is a feature needed to either derive a concrete program trace from the forward and backward symbolic executions, or determine that no such a trace exists since the intersection gets empty at some point in the traces. It turns out that intersecting FAs is a quite complex task, which has to, e.g., deal with the fact that the two FAs being intersected may use a different decomposition of the heap graphs they represent.

Moreover, Forester has also been extended with the most advanced abstraction mechanism known in the context of ARTMC, namely *predicate language abstraction*. In its case, one collapses those TA states whose languages intersect the same predicate languages (represented also by TAs). The predicate languages to be used are learned in a *counterexample guided refinement* (CEGAR) loop from the TAs that are generated within backward executions of the program along spurious counterexample traces. Currently, the first execution of Forester uses the finite height abstraction, which is then refined in the further runs by combining it with the predicate language abstraction.

More details on the mentioned checking of validity of counterexamples and the refinable predicate language abstraction used in the context of FAs are still to be published, but a preliminary description can be found in [6].

2 Tool Architecture

Forester is implemented as a *GCC plugin* using the interface over GCC provided by the Code Listener infrastructure [4]. GIMPLE instructions used in the intermediate GCC code are translated to instructions of a specialised register machine that Forester uses to symbolically execute programs in the abstract domain of FAs. Forester uses the VATA [5] library to handle non-deterministic TAs from which FAs are built. Both Forester and VATA are implemented in C++.

3 Strengths and Weaknesses

The strengths of Forester are the following: (1) Forester is based on a sound verification approach, (2) its abstract domain allows one to analyse a large variety of classes of shape graphs, ranging from various kinds of (nested) lists, trees, to skip lists, and their combinations, (3) it can provide the user with error witnesses, (4) it newly analyses the counterexamples and refines the abstraction based one them, and (5) its internals (e.g., entailment checking) are built upon a well-understood automata theory and technology, which is constantly being developed by a wide community of researchers. Compared to the previous participation of Forester in SV-COMP in 2015, due to our enhancements, we were able to correctly mark 4 new bug-free benchmarks and 12 new erroneous benchmarks from the challenging Heap Data Structures category.

Among the main weaknesses of Forester is its weak support of handling non-pointer data such as integers or arrays. Therefore it participates in the Heap Data Structures category only, but even in this category it still loses some points due to not handling non-pointer features properly. Another weakness of Forester is that it does not support some advanced C language constructions. In particular, Forester currently loses the most points in the Heap Data Structures category by not implementing any support for pointers to functions. Due to this, Forester cannot analyse nearly 80 test cases. Another feature of C not fully supported by Forester are pointers to unstructured memory. Although a basic support for handling them is in place, Forester still has problems in tracking the size of an allocated unstructured memory block.

4 Tool Setup and Configuration

An archive with the SV-COMP'16 version of Forester is available at the web page of Forester[3]. The archive contains the source code of Forester and VATA. Instructions for compiling and running Forester are in the file `README-FORESTER-SVCOMP-2016` in the root directory of the archive. After compilation, the directory `fa_build` with scripts for running Forester is created. The script for running Forester in SV-COMP is named `sv_comp_run.py`. It is also used in the BenchExec wrapper script of Forester.

The parameters of `sv_comp_run.py` are the following. The mandatory parameter of the script is the path to the file with the program under verification. The file for storing the witness leading to a counterexample is specified by the parameter `--trace`. The path to the property file is defined by the parameter `--properties`.

When Forester is run within the BenchExec framework, most of the parameters are set automatically by its wrapper script. The only exception is the parameter `--trace`, which must be defined manually in the `forester.xml` file used as the input of BenchExec. The wrapper script of Forester for BenchExec is called `forester.py`. Both files are available from the official page for SV-COMP'16 results reproduction (`http://sv-comp.sosy-lab.org/2016/systems.php`).

The output of Forester printed to the standard output has a similar format to the specification given by the rules of SV-COMP'16, specified in detail in the mentioned `README` file. Forester participates only in the Heap Data Structures category.

5 Software Project and Contributors

Forester has been developed at Brno University of Technology since 2010. The authors of this paper are currently the only people involved to development of Forester. Forester and the VATA library are both licensed under GPL.

Acknowledgement. This work was supported by the Czech Science Foundation under the project 14-11384S. Martin Hruška is a holder of the Brno Ph.D. Talent Scholarship, funded by the Brno City Municipality.

References

1. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. Formal Methods in System Design, **41**(1), Springer, 2012.
2. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS 8044, Springer, 2013.
3. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. Software Tools for Technology Transfer, **14**(2), Springer, 2012.
4. Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11, Part I*, LNCS 6927, Springer, 2011.
5. Lengál, O., Šimáček, J., Vojnar, T.: VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS'12*, LNCS 7214, Springer, 2012.
6. Hruška, M.: Verification of Pointer Programs Based on Forest Automata, MSc. thesis, Brno University of Technology, 2015.

[3] `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester`