Forester: Shape Analysis Using Tree Automata
(Competition Contribution)

Lukáš Holík, Martin Hruška, Ondřej Lengál,
Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Forester is a tool for shape analysis of programs with complex dynamic data structures—including various flavours of lists (such as singly/doubly linked lists, nested lists, or skip lists) as well as trees and other complex data structures—that uses an abstract domain based on finite tree automata. This paper gives a brief description of the verification approach of Forester and discusses its strong and weak points revealed during its participation in SV-COMP'15.

1 Verification Approach

Forester is a tool for (sound) shape analysis of programs with complex dynamic data structures, such as various flavours of lists (including singly/doubly linked lists, nested lists, or skip lists) as well as trees and other complex data structures. The used abstract domain contains *forest automata*, a generalization of finite tree automata, described in [1,2]. The approach attempts to combine the strong points of two other approaches: (i) the scalability of *separation logic* [3], which is due to the concept of separation allowing local reasoning about disjoint parts of the program heap, and (ii) the flexibility of *abstract regular tree model checking* (ARTMC) [4], which uses finite tree automata for symbolic representation of the sets of reachable heap graphs.

The heap representation is based on the *forest decomposition* of the heap. This is a representation of the heap by a tuple of trees such that the roots of the trees correspond to the *cut-points* of the graph. A cut-point is a node that is either referenced from a program variable or that has more than one incoming edge. The trees in the tuple are free of cut-points and their leaves contain either non-pointer values or explicit references to roots of other trees. To represent *sets* of heaps—the elements of the concrete domain—instead of a tuple of trees Forester uses a tuple of tree automata, the so-called forest automaton. Each tree automaton represents a set of cut-point-free trees; the heap graphs represented by a forest automaton can be constructed from the forest automaton by taking a tree from the language of every tree automaton and connecting the references in the leaves of the trees to the roots of the referenced trees.

We associate an abstract transformer manipulating forest automata with every concrete operation. Joins are handled precisely (we split the execution and proceed in the verification run for each branch independently). The abstraction operator, called on loop points, is based on the *finite height abstraction* from ARTMC [4], and its main idea is to introduce loops in the tree automata to allow for a representation of infinite sets of trees with regular structure.

In order to be able to verify programs manipulating heaps where the number of cut-points is unbounded, we use *hierarchical* forest automata. These are forest automata

that can use other (lower-level) forest automata as symbols, in a hierarchy of a finite height. These lower-level forest automata are called *boxes*. A box is essentially used to represent a repeated structure of the heap graph that contains some cut-points. The boxes to be used in a verification run are devised using the learning algorithm from [2].

In order to use Forester, it is necessary to properly model all external functions; Forester itself implements models of the two basic functions for memory allocation, `malloc` and `free`.

2 Tool Architecture

Forester is implemented in C++ as a GCC plugin that uses the Code Listener [5] infrastructure as the front-end for preprocessing the intermediate representation used in GCC (called GIMPLE) into a compiler-independent representation. Further, it uses the VATA library [6] as the back-end for manipulating tree automata. Forester translates the input program obtained from Code Listener into its internal representation, in which every program statement is represented by a sequence of abstract transformers that manipulate the symbolic representation of the program. The translated program is then subject to symbolic execution, during which Forester detects memory errors (invalid dereferences or frees, occurrence of garbage) and reachability of an error line.

3 Strengths and Weaknesses

The main strong point of Forester is that it gives sound results on all verification tasks that we run. In particular, Forester was able to find shape invariants for the most difficult programs in the Memory Safety category, i.e. programs manipulating 2 and 3 level skip lists, trees (including the Deutsch-Schorr-Waite tree traversal algorithm), and (nested) singly/doubly linked lists.

However, the overall performance of Forester on the benchmarks of SV-COMP'15 was significantly hindered by the following two causes. The first cause is the still quite high degree of immaturity of Forester in dealing with real-life C code with all its caveats—in the case Forester encounters some unsupported feature of C (such as the `union` data type, function pointers, or the use of arrays), it returns the UNKNOWN answer. The other cause is the incompleteness of the verification procedure and the current inability of the tool to distinguish spurious counterexamples from real ones; if a potentially spurious counterexample is found by Forester, it again returns UNKNOWN. However, it is possible to use the option `--false` to switch Forester into a mode in which it reports all found counterexamples and allows their subsequent analysis, either by a user or by e.g. a bug hunter.

4 Tool Setup and Configuration

An archive with the source code of the Forester competition release[1] can be downloaded from the project web page. The file `README-FORESTER-SVCOMP-2015` in the root directory of the archive contains information about how to build and run the tool. After Forester is successfully built, the `fa_build` directory contains a Python

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/download/forester-2014-10-31-9d3ad64.tar.gz`

script `sv_comp_run.py` that executes the tool and transforms its output to the format expected by SV-COMP. The script expects the path to the file with the program under verification as an argument; further, the path to a file with a description of the properties to be verified can be specified using the `--properties` option. For the case the answer of Forester is FALSE (i.e. a real error is encountered in the program under verification), Forester returns the name of the property that has been violated. Moreover, a mandatory `--trace` option is required to specify the path to the file where the witness leading from the entry point to the statement that caused the violation is to be saved. On the other hand, if Forester finds a shape invariant of the program without encountering a property violation, it returns TRUE.

Furthermore, if the `--time` option is given, Forester also writes to the standard output the CPU time that the verification run took. It is also possible to generate graphical representations of abstract program configurations at some line of code into a sequence of files named according to the template `filename-XXXX.dot` by inserting the statement `__VERIFIER_plot("filename")` to the desired line of code in the processed program.

Forester participates in the following two categories of SV-COMP'15: Heap Manipulation and Memory Safety.

5 Software Project and Conclusion

Forester is developed by the VeriFIT group at Brno University of Technology and distributed under the GNU General Public License version 3. The source code of Forester is in a `git` repository shared with Predator (a memory analyzer based on symbolic memory graphs [7]), which is developed in the same group.

This is the first submission of Forester to SV-COMP. In the future, we wish to focus on the following two points: (a) extending the set of the supported features of C, and (b) developing the ability to properly identify spurious counterexamples and to use them to refine the abstraction used.

Acknowledgement. This work was supported by the Czech Science Foundation (projects 14-11384S and 202/13/37876P), the BUT FIT project FIT-S-14-2486, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

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. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In *Proc. of CAV'07*, LNCS 4590, Springer, 2007.
4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. International Journal on Software Tools for Technology Transfer, **14**(2), Springer, 2012.
5. 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.
6. 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.
7. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In *Proc. of SAS'13*, LNCS 7935, Springer, 2013.