Forester: From Heap Shapes to Automata Predicates
(Competition Contribution)

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

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Brno, Czech Republic
ihruska@fit.vutbr.cz

Abstract. This paper describes the participation of Forester in the SV-COMP 2017 competition on software verification. We briefly present the verification procedure used by Forester, the architecture of Forester, and changes in Forester done since the previous year of SV-COMP, in particular the fully-automatically refinable abstraction for hierarchical forest automata.

1 Verification Approach

Forester implements an automated shape analysis that uses forest automata (FAs) to represent sets of reachable shapes of the heap of the analysed program. In particular, heap configurations are viewed as (directed) graphs, decomposed into tuples of trees, and sets of such decompositions are encoded by FAs that themselves have the form of tuples of tree automata (TAs). The tree decomposition is based on detecting the so-called *cut-points* of the heap graphs, which are nodes either pointed by a variable or having more than one incoming edge. The tree decomposition is then obtained by cutting a heap graph at the cut-points and redirecting each incoming edge of a cut-point to a new leaf node labelled by a reference to the tree with the cut-point as the root.

In order to allow for representing data structures with an unbounded number of cut-points, a notion of *hierarchical FAs* (HFAs) is introduced. An example of a structure for whose representation plain FAs are insufficient and HFAs are needed is the doubly-linked list (DLL). Indeed, each internal DLL node is a cut-point since it is pointed to by its predecessor and successor nodes. An HFA can use other HFAs, called *nested HFAs* or *boxes*, as symbols of its alphabet. Boxes can represent (repeating) sub-graphs of heap graphs, possibly encapsulating (hiding) an unbounded number of cut-points. A special *folding* operation is then used to pack a part of an HFA into a box and add the box to the alphabet of the resulting HFA. On the contrary, when an analysed program accesses a part of a heap folded into a box, the box is *unfolded* by plugging its content back to the wrapping HFA. A more detailed description of these operations can be found in [1,2].

M. Hruška —Jury member.

A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 365–369, 2017.
DOI: 10.1007/978-3-662-54580-5_24

The verification procedure implemented in Forester symbolically executes the program in the abstract domain of HFAs. At loop points, HFAs are abstracted, implementing the idea of *abstract regular model checking* [3]. The abstraction is applied component-wise, i.e., to individual TAs, collapsing some of their states, which over-approximates the set of reachable heap configurations. The abstraction speeds up the reachability analysis and enables termination on infinite state spaces, but can also yield spurious counterexamples. To recognize them, Forester was, in the previous SV-COMP [8], modified to run backwards (not using any abstraction) along a suspected error trace. Together with using predicate language abstraction of TAs—which collapses TA states intersecting with the same predicate languages, and which can be refined by adding more predicate languages—a *counterexample-guided abstraction refinement* (CEGAR) [6] loop is obtained.

The backward run is performed over a trace consisting of micro-instructions used by Forester. The trace leads from the beginning of the analysed program to a line where the given specification was found broken. Forester then precisely reverts all micro-instructions along the trace starting from its end. For example, when a new state of an FA was created in the forward run, Forester removes it in the backward run. The abstraction is reverted by intersecting FAs from the forward and backward run. If the intersection is empty, Forester reports a spurious counterexample, derives new predicates to refine the abstraction, and restarts the analysis. The new predicate languages are encoded by TAs selected from the FA obtained in the backward run at the point where the empty intersection with the forward run was detected. Otherwise, if the backward run reaches the beginning of the trace, the counterexample is reported as real.

For SV-COMP 2017, we extended the backward run and predicate language abstraction from plain FAs (done in [8]) to HFAs, which requires one to take into account boxes. In particular, if the original algorithms were used, it may happen that some subgraphs would be folded into a box in the forward run, while they would not be folded into this box in the backward run, meaning that the general structure of the FAs would be different. The intersection operation (which does not consider the semantics of boxes) would then determine that languages of the corresponding HFAs do not intersect. This would significantly decrease the precision of the operation. One option how to address this issue and increase the precision would be to modify the intersection operation to take into account the semantics of boxes and make it try to unfold them on the fly. We take a different approach, which enables us to successfully a larger class of programs.

Our way of dealing with the issue is to keep the HFAs obtained during the backward run *compatible* with the HFAs in the forward run. The compatibility intuitively means that the two HFAs partition the same heaps in the same way, in other words, if a heap is accepted by both HFAs, it is decomposed into the same components and the same boxes in both HFAs. When compatibility is enforced, we can (i) avoid inner inspection of boxes during the intersection operation, (ii) enable precise reversion of micro-instructions, and, as a side-effect, (iii) use a simple standard TA intersection operation performed component-wise

on the HFAs. To maintain the HFAs in the backward compatible, we needed to significantly alter instructions used therein (previously, no structural constraints were imposed on the FAs; in order to deal with their different interconnection structure, a more complex intersection operation was needed).

The operations that are the most challenging to revert in the backward run are the following: folding (which is, in fact, performed together with abstraction in a loop of the form fold, abstract, fold, abstract, and so on until a fixpoint is reached), unfolding, and normalization. The normalization removes cut-points that are no longer needed, glues together TAs that stop being separated by cut-points, and orders component TAs in an FA in order to transforms the given HFA into a so-called *canonicity-respecting form* needed for testing inclusion. The reversion of folding then needs to guarantee that the sub-graphs in the folded box will appear in the correct components after the operation (taking into account that folding can be done multiple times during a single abstraction). On the other hand, the reversion of unfolding needs to guarantee that the unfolded box will be folded back into a box within the correct component. Lastly, the reversion of normalization needs to cut and re-order components into correct places. A more precise description of the described methods can be found in [7].

2 Tool Architecture

FORESTER is implemented in C++11 as a GCC plugin using the Code Listener framework [4]. The representation of a program obtained through Code Listener is translated into FORESTER's own internal microcode, which is symbolically executed. FORESTER uses the VATA library [5] for representation and manipulation with *nondeterministic TAs* (NTAs). VATA contains an optimized implementation of efficient algorithms for dealing with TAs, including operations such as state reduction of NTAs and testing their language inclusion, which is a crucial operation in FORESTER for determining whether an execution branch has reached a fixpoint.

3 Strengths and Weaknesses

One of the most important features of Forester is that it is sound (wrt the intermediate code obtained from GCC, which may have already removed some possible behaviours of the original code; e.g., GCC already fixes the order of evaluation of a function's parameters), i.e., if it answers *TRUE*, there is indeed no bug in the program. Moreover, due to the recent improvements in FORESTER regarding counterexample-based abstraction refinement [7], the number of false positives (i.e., wrong answers *FALSE*) on the benchmark of SV-COMP 2017 is significantly reduced. Concretely, the new version gets no false positives, which gives us approximately 40% more points than we would have obtained with the version of FORESTER from SV-COMP 2016, in particular on examples that contain DLLs and need to perform abstraction refinement. FORESTER can also output *UNKNOWN* if it establishes that it cannot give a correct answer.

This happens when the tool exceeds the time given by the SV-COMP rules—e.g., when searching for a shape invariant not expressible using HFAs—or upon detection of an unsupported feature of C. Forester specialises almost exclusively in pointer manipulations and inference of complex shape properties of pointer structures. It does not implement advanced syntactic features such as function pointers, heavily used in the LDV benchmark, but also more basic features such as arrays, unions, recursion, arithmetic, or bit operations.

The formalism of HFAs allows Forester to represent in a quite precise way the invariant of rather complex data structures, such as skip lists of 2 or 3 levels, various flavours of nested lists, or trees with parent and root pointers. The used representation is, moreover, quite compact, and kept small via simulation-based reduction of NTAs.

4 Tool Setup, Configuration, and Witnesses

The distribution of Forester for SV-COMP 2017 is available from the web page of Forester[1] from the link highlighted as the SV-COMP 2017 binary version. The tool is provided in the form of a shared object library `libfa.so` together with a Python wrapper `sv_comp_run.py`. The file `README-FORESTER-SVCOMP-2017` describes the dependencies of Forester and parameters of the Python script.

The `sv_comp_run.py` script is run as follows:

```
sv_comp_run.py [--help] <source>
   --properties <prp> --trace <trace>
```

where `<trace>` is the output file for a (violation/correctness) witness, `<prp>` is the path to the property file, and `<source>` is the verified program. When Forester is run within the BenchExec framework, most of the parameters are set automatically by its BenchExec wrapper script. The only exception is the parameter `--trace`, which must be defined manually in an option node of the XML input file of BenchExec.

The format of a violation witness is an automaton, represented using GraphML (an XML schema), that represents a buggy trace through the program, while the format of a correctness witness is (again) a GraphML automaton whose states correspond to loop points in the program, and are further annotated (using an XML node with the key `automaton`) by a representation of the set of FAs over-approximating the set of reachable program configurations at the given state. Forester participates only in the MemSafety-Heap and ReachSafety-Heap categories and opts out from the rest.

5 Software Project and Contributors

Forester has been under development at Brno University of Technology since 2010. Forester and the Vata library are both licensed under GPLv3.

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

The source code of FORESTER is available at https://github.com/martinhruska/forester/. The authors of this paper are currently the only people involved in its development.

Acknowledgement. Supported by the Czech Science Foundation (project 17-12465S), the BUT FIT project FIT-S-17-4014, and the IT4IXS: IT4Innovations Excellence in Science project (LQ1602). 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 Syst. Des. **41**(1), 83–106 (2012)
2. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 740–755. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_52
3. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. STTT **14**(2), 167–191 (2012)
4. Dudka, K., Peringer, P., Vojnar, T.: An easy to use infrastructure for building static analysis tools. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2011. LNCS, vol. 6927, pp. 527–534. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27549-4_68
5. Lengál, O., Šimáček, J., Vojnar, T.: VATA: a library for efficient manipulation of non-deterministic tree automata. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 79–94. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_7
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
7. Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Vojnar, T.: Counterexample validation and interpolation-based refinement for forest automata. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 288–309. Springer, Cham (2017). doi:10.1007/978-3-319-52234-0_16
8. Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Run Forester, Run Backwards!. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 923–926. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_61