Excel@FIT 2015
http://excel.fit.vutbr.cz

Verification of Pointer Programs Based on Forest Automata

Martin Hruška*

Abstract
In this work, we develop an existing method for shape analysis based on so called forest automata, and we also improve its implementation, the tool Forester. Forest automata are based on tree automata and Forester includes a simple implementation of tree automata. Our first contribution is replacing this implementation by a general purpose tree automata library VATA, which contains highly optimized implementations of the used automata operations. The version of Forester using the VATA library participated in the competition SV-COMP 2015. Secondly, we have extended the forest automata based verification method with a counterexample analysis. The results of the counterexample analysis can be then used to check whether a found error is a spurious or a real one what could be used for refinement of predicate abstraction. We are currently working on its application in counterexample based abstraction refinement.

Keywords: Forest Automata — Formal Verification — Static Analysis — Complex Data Structures — Tree Automata — Backward Run — Predicate Abstraction

Supplementary Material: *N/A*

*xhrusk16@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The importance of computers in our everyday life has largely increased over the past few decades. A lot of us can only hardly imagine doing their jobs without a help of an appropriate computer program and we also spend a plenty of time using computers (e.g. personal computers or mobile devices) in leisure time. The computer programs are also often used in very critical instances like autopilot in an airplane. But the growing number of applications of computer programs brings also the need for their greater safety and security.

However, a guarantee of software correctness is not an easy task because the programs often go through many states during the computation and it could be very time and space consuming or even impossible to check whether no undesirable behavior may appear in any of those states. One of the approaches to ensure the software quality is *testing* (and dynamic analysis) which is based on running the program, in the different contexts, with the different inputs and matching a program behavior and the outputs with the expected ones. This method can satisfy many of the requirements for the software quality and often covers the great space of the program behaviors. On the other side, it is only possible to prove presence of the errors using testing not their absence [1]. Moreover finding some errors during testing does not mean that all errors are eliminated.

The mentioned weakness of testing can be resolved by *formal verification*. Formal verification aims at using rigorous mathematical methods to check whether a given system meets a given specification [2]. There are three main branches of formal verification. The first one is *model checking* which systematically explores the states of a model (e.g. model of a program) to prove that the model satisfies the property. The second one is *static analysis* which is done over a source code (or some modification of it) of a system without a need of its execution. One of the important and very widely used approaches in static analysis is called *abstract interpretation* where the analysis is performed

by applying abstract transformers corresponding to the original program semantics over an abstract domain. The last approach is *theorem proving*. It proves the program safe in a standard mathematical way – starting from axioms and using inference rules to verify the properties of a given system. Theorem proving ca be partially automated.

This work deals with a specific part of static analysis called *shape analysis* which is focused on the verification of programs manipulating complex data structures (like the different kinds of lists and trees), typically allocated on the heap. The properties checked are for example checking whether no dangling pointers are dereferenced (no invalid dereferences), whether all allocated memory on a heap is also freed during the program execution (no memory leaks) or whether there is not freed pointer without assigned memory (no invalid free). There are different approaches to this kind of static analysis with the different advantages. For example, the approach based on *separation logic* [3, 4] provides great scalability of a verification procedure. On the other side, the automata based approach, particularly *abstract regular tree model checking* (ARTMC) [5], is superior in its flexibility and generality. This work focuses on the verification procedure based on the concept of forest automata (FA) which combines benefits of the both mentioned approaches.

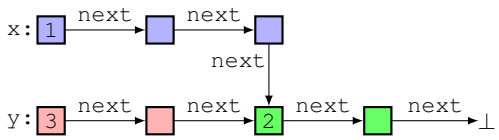
Forest automata (FA) have been introduced in [6] and they are an extension of finite automata or, more precisely, extension of finite tree automata (TA). They are used as an abstract domain in a verification procedure which performs symbolic execution of the analyzed program is performed. A prototype of this verification procedure has been implemented in a tool called *Forester* [7]. Forester verifies programs written in C language and it detects safety violations like invalid dereferences, invalid frees, memory leaks and also reachability of an error program location. It is able to verify non-trivial data structures like skip-lists of the second and the third level. However, the current implementation is far from perfect. E.g., Forester currently does not support the complete C language syntax and it is not also yet possible to check whether a found error is real or spurious. It is also needed to refactor some parts of Forester code before any extension of the tool. A general goal of this work is to improve Forester in the areas described further.

Since FA are highly related to the finite tree automata, the first goal of this work is to change the underlying implementation of finite tree automata in Forester to the VATA library – a state-of-the-art library for TA manipulation [8]. Particularly, this consists creating an interface between Forester and VATA to employ VATA as a TA library for Forester backend which should bring better maintainability and modularity than having a special TA library implementation within Forester as it is now. The second goal of this work is to design and implement *backward run* for FA based verification which enables checking spuriousness of an error found in a program. The error could be spurious because too strong abstraction over FA is used. The information gained by backward run could be used for the refinement of *predicate abstraction* (one kind of abstraction over FA) to prevent verification procedure from getting the same spurious error again. This gradual refinement is known as *counterexample-guided abstraction refinement* (CEGAR) [9]. However, Forester does not currently use the mentioned predicate abstraction but *height abstraction* because predicate abstraction needs backward run to be fully functional. Height abstraction is less precise and less flexible compared to predicate abstraction. Implementing predicate abstraction enables analysis of even more complex data structures like red-black trees. A part of the second goal is an implementation of predicate abstraction using backward run in Forester.

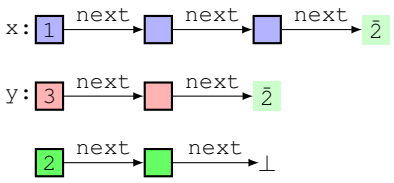
This paper is structured as follows. In Section 2, a high level overview of shape analysis based on forest automata is given. After the overview section we continue with description of backward run and predicate abstraction takes place in Section 3. Section 4 provides description of modification of Forester to use the VATA library as a backend. Finally, conclusions of the work is given in Section 5.

2. Overview of Verification Method

This section provides a high level overview of the verification procedure based on forest automata. As it was already mentioned we consider only C programs manipulating dynamic data structures. The simple examples of such structures are singly-linked lists or binary trees. Consider a singly-linked list containing an (integer) data member and a pointer to the next item in the list and also consider two variables x and y pointing the singly-linked list. An instance of the singly-linked list allocated on a heap can be viewed as an oriented graph. The nodes of the graph correspond to the allocated memory cells and the edges of the graph are related to the pointers referencing another allocated memory cell of the list. Each node can be labeled by a name of the pointer variables that reference the given node or it could be labeled by a data contained in the node. The example of such a graph is shown in Figure 1. We omit the data nodes in Fig-



(a) A heap graph



(b) A tree decomposition of the heap graph

Figure 1. Figure shows tree decomposition of a heap graph. The heap graph representing singly-linked list is shown in Figure 1a. The nodes could be labeled by a pointer variable that references allocated memory cell corresponding to the node. Moreover, the cut-points are labeled by an order number that they have in an ordering over the cut-points. In this case, there are three cut-points with an order number 1, 2 and 3. The graph is split to three trees shown in Figure 1b by the heap decomposition. The new trees could contain the references to another trees, in this case tree 1 and tree 3 have the nodes referencing tree 2.

ure 1 for sake of clarity. The figure shows that the variables x, y points two singly-linked list (the blue and the red nodes) that have common part (the green nodes). The original heap graph is shown in Figure 1a. The tree decomposition, which is described later, is in Figure 1b.

We can decompose the graph to a set of trees by the following method [6]. First we identify so called *cut-points* — the nodes referenced by one or more pointer variables or the nodes having more than one incoming edges. The cut-points are then numbered by a depth-first traversal of the graph starting from the nodes referenced by the variables. The traversal and the numbering defines an ordering over the cut-points. Then the graph is split to the trees with cut-point as the roots. Finally, we redirect the edges leading to the nodes marked as the cut-points to the new nodes because the cut-points are now roots of the trees.

The tuple of trees (created by graph decomposition) is called forest. Such a forest is accepted by a forest automaton as a member of its language. A forest automaton is a tuple of finite tree automata which are automata accepting trees instead of words (in comparison with finite automata). The tree automata within a forest automaton are interconnected by references (that are also symbols in their alphabet). Each of the tree automata accepts a set of trees and so we get a language of a forest automaton by their Cartesian product what is a set of tuples of trees (connected by the mentioned references). These tuples of trees then represents a state of a heap.

Forest automata can be viewed also as an abstract domain in context of abstract interpretation whereas a concrete domain is a set of heaps with allocated data structures [10]. It is also possible to define abstract transformers corresponding to the (concrete) program operations over forest automata. The verification procedure is then performed as a symbolic execution where the abstract transformers are gradually applied to abstract domain. When an error like an invalid memory reference or invalid free is detected during the symbolic execution it implies that it is possible that there is an error in the program (but the error could be also spurious because the forest automata overapproximate a state space of the original program).

It is possible to represent a dynamic data structure with bounded number of cut-points by the mentioned approach. But it cannot be used for data structures with an unbounded number of cut-points like doubly-linked lists. The unboudness comes from the fact that each node of a doubly-linked list is a a cut-point because it is pointed by the next and also the previous member of the list. This problem can be solved by introducing so-called *boxes* [11]. Boxes are symbols hiding the repeating sub-graphs causing unboudness. Then these sub-graphs are replaced by an hyperedge labeled by an appropriate box and so a new hierarchical hypergraph with bounded number of cut-points is created. More precisely, boxes can be viewed as forest automata representing the repeating subgraphs. So it is still possible to represent the hypergraph with forest automata where boxes are symbols of forest automata. We get hierarchical forest automata this way because forest automata of lower level are included as symbols in alphabets of forest automata of higher level. Note that a forest automaton of certain level can contain in its alphabet only automata of lower level than its own level.

More technical details about a heap decomposition and forest automata can be found in [6, 11].

3. Backward Run and Predicate Abstraction

We have already covered how to represent dynamic data structures by forest automata in the previous section. Forest automata are able to handle also infinite state spaces rising from a possible unboudness of the mentioned dynamic data structures. However, there is

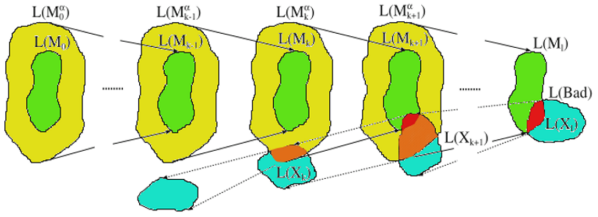


Figure 2. Figure is taken from [5]. Figure shows a forest automaton language $L(M_k)$ before (green area) and language $L(M_k^\alpha)$ after (yellow area) abstraction where k is the order of the state within symbolic execution. One oval consisting green and yellow part represents language of a forest automaton in a symbolic state of symbolic execution. Figure also shows how a forest automaton (and it language) is gradually changed by abstract transformers during a symbolic execution what is illustrated by the lines connecting ovals. A found error is shown as a red area in the last oval. The backward run starts from this error and it finds that an error is spurious because an intersection (the orange areas) of forest automata language from backward (the blue ovals) and forward (the green and yellow areas) run is empty at the level of the symbolic state with the language $L(M_{k-1}^\alpha)$.

still the problem of state explosion that often makes nearly impossible to finish verification procedure of a system in real time and moreover, not even termination of the verification procedure is guaranteed at all. Therefore the abstraction over the forest automata is introduced to increase probability of termination and to accelerate the method. The abstraction speeds up the computation by merging more states of forest automata to one abstract state. This also causes the overapproximation of the set of reachable states of an analyzed system. It can lead to reaching the error states in an abstracted state space which are not present in the real system (the real system is a computer program in this case). We can prevent reaching such spurious counterexamples by a refinement of the abstraction what avoids repeating of the abstraction of the states that lead to these errors. The application of an abstraction over forest (and tree) automata in this work is based on *abstract regular tree model checking* [5] and the refinement of the abstraction is based on *counterexample-guided abstraction refinement* [9] as it was mentioned in the introduction section.

Forester now implements so called height abstraction. This abstraction merges states whose languages (note that a language of state is a set of trees) are equal when the trees of the languages are compared only to the given depth. The main problem of this abstraction technique is the low ability of refinement. The only way of refinement is by the parameter defining the depth of tree comparison what does not ensure that a found spurious error will not be found again. The abstraction allowing more fine tuning is so called *predicate abstraction* [5] that introduces a set of predicates and merges the states in which the same predicates hold. When a spurious counterexample is found, then the set of predicates is extended by a predicate that prevents reaching the spurious counterexample again and the verification procedure is rerun with the new predicate.

Finding the new predicate and checking spuriousness of an error is done by *backward run*. A backward run starts from the forest automata modelling a heap with a found error and reverting gradually each abstract operation over the forest automata in backward order to the (forward) symbolic execution. When an automata with an empty language is get by a reverse abstract transformation then the error is spurious. The last automaton of backward run with a non empty language is then used as a new predicate. Backward run is illustrated by Figure 2 and the illustration is further described in the caption of the figure.

An operation often needed during backward run is a forest automata intersection that is necessary for some reverse abstract transformers. The intersection is a general operation over forest automata, not directly related to backward run. Although the intersection is one of the basic operations in automata theory, it has not been implemented yet and its realization would enrich the whole forest automata theory.

We focuses on implementation backward run and predicate abstraction for non-hierarchical forest automata. It includes a design of the reverse operations for abstract transformers over the abstract domain and the application of predicate abstraction on forest automata. Moreover, as it was already mentioned the intersection of forest automata is needed in some reverse operations of backward run. So it is also part of this work to design it and to implement it. As it was said, the realization of the intersection is not only useful for backward run but it is nice general contribution to the theory of forest automata.

3.1 Evaluation

The implementation of the backward run has been already finished and evaluated on the SV-COMP benchmark. The implementation of backward run helps in confirmation of the new 6 real errors in the programs in the memory safety category and of the new 8 real errors in the heap manipulation category. Moreover, the 3 spurious counterexamples were detected in heap manipulation category. The test cases with

the spurious counterexamples will be further resolved by implementing predicate abstraction. The results are summarized in Table 1. The analyzed programs in SV-COMP benchmark includes e.g., programs from LDV (Linux Drivers Verification) project containing implementation of alternating singly-linked list or manipulation with mutex locks, or the programs implementing bubble sort over a list implementation from the Linux kernel. Note that Forester does not successfully process all the programs in the test set because it does not currently support all the C language constructions. It causes that the number of the found errors, spurious or real, is not as high as it would be when Forester could analyze all the programs in the benchmark.

4. Forester and VATA

This section describes the extension of the Forester tool to enable employing the VATA library efficient implementation of tree automata and the operations over them, especially checking inclusion of languages of tree automata. Forester had its own implementation of tree automata and their operations (in module called TA), however not as efficient as VATA. It is also less maintainable then using a standalone library that implements the state-of-the-art algorithms.

VATA provides two encodings of tree automata – explicit and semi-symbolic. They differ mainly in the way of representation of transition relation of tree automata. The explicit encoding stores the transitions in the explicit hash tables while the semi-symbolic uses a multi-binary decision diagram. We use the implementation of explicit encoding of TA in this work because it is currently the only one that supports the most of needed operations over TA. It is also more efficient than the semi-symbolic encoding for the purposes of Forester because no large alphabets are used during the verification procedure. So the advantage of the semi-symbolic encoding would not be fully utilized here.

Forester implementation is currently far from maturity and the high structural dependencies is one of its bottlenecks. So the first thing in refactoring was the reduction of number of the dependencies between classes, especially reduction of the dependencies on the existing Forester module implementing tree automata, what makes easier to port Forester to the VATA library. Applying the *Law of Demeter* [12] to the code manipulating tree automata is one of approaches how to do it. It practically means that classes using TA should explicitly implement methods providing information about TA instead of providing instance of TA object itself. Applying the Law of Demeter reduces knowledge needed about implementation of TA module across the whole project. Another way of reducing structural dependencies is making all possible methods and members of TA private (in sense of C++ language [13]). This creates an explicit tight interface to tree automata module so it is easier to see which methods needs to implement a new tree automata implementation. The last from the biggest changes in this work is using generic programming. It should make possible to make operations over tree automata without depending on a concrete implementation. In this case it is done by refactoring employing *auto* type deduction in C++ in combination with the *Iterator* pattern. It is used for example when one needs to iterate over all transitions of tree automata or all transitions which have the same state as parent and performs a operation with the transitions. This last change employs principles of generic programming where

After refactoring, it is possible to apply the design pattern *adapter* [14] to create an interface between Forester and the VATA library. This makes possible to include VATA without need of rewriting Forester to the names of methods and data members used in VATA. It creates also only one place (particularly adapter class) connecting Forester and VATA instead of including VATA into many of the Forester classes and so it prevents creating too strong relation between them.

The main part of the adapter patter is newly implemented class *Adapter* taking the role of Adaptor. We decided to use the implementation approach to Adaptor preferring composition over inheritance. It is more suitable for our purposes since we often needs to rename methods (a name of a method in Forester differs from the name of the semantically same method in VATA) or convert a data type of parameter of tree automata operation (e.g. from vector to set).

Now we provide more technical details about implementation. The class *VATAAdapter* instantiates the class *ExplicitTreeAut* from VATA as its private data member and redirects to it the method calls from Forester (the names of methods of VATAAdapter are the same as they were in the original Forester TA module). *VATAAdapter* also sometimes performs the mentioned conversion of the data types. There are methods implemented by adapter not presented in VATA like

Table 1. SV-COMP Benchmark Evaluation of Backward Run.

Category	Real	Spurious
HeapManipulation	8	3
MemorySafety	6	0

method *unfoldAtRoot* performing an unfolding, an operation specific for forest automata. The methods of this kind are very Forester specific so it is not sensible to add them to a general purpose library as VATA and it is better to implement them in an interface class like VATAAdapter.

We originally supposed that it would be possible to keep the original TA module along VATA adapter to be able to easily switch between them. However it emerged that this would bring high overhead in some situations. E.g., a conversion of some data types would be needed in this case what is overhead compared to the implementation where the data types compatible with VATA are used directly in the Forester code. Hence we decided to remove the original tree automata module and further support only the version of Forester with the VATA library.

5. Conclusions

The main goals of this work were (i) to implement version of Forester tool that uses the VATA library for tree automata representation and manipulation and (ii) to extend verification procedure based on forest automata with backward run for detection of the spurious errors found in the analysed program. The theory of forest automata, the related theory of tree automata and the verification procedure based on forest automata has been studied to fulfill the two goals. The connection of Forester and the VATA library was designed and implemented after the analysis of the both tools. Forester had to be refactored for this purposes.

The first goal has been already reached and Forester using the VATA library successfully participated in competition SV-COMP 2015 [15]. The second goal is partly finished. The backward run with needed forest automata intersection for non-hierarchical forest automata has been completed and evaluated on SV-COMP benchmark. The implementation of predicate abstraction is currently in progress.

The future work after finishing basic predicate abstraction could be generalizing predicate abstraction and backward run to the hierarchical forest automata what enables Forester to analyze more test cases. Forester code needs also to be further refactored and greater support for C language construction should be implemented to make Forester able to analyze more complex programs.

References

[1] Edsger W. Dijkstra. Structured Programming. In *Software Engineering Techniques*. NATO Science Committee, 1970.

[2] Tomáš Vojnar. *FAV lectures*. Brno, CZ.

[3] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer Berlin Heidelberg, 2007.

[5] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[6] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 424–440. Springer Berlin Heidelberg, 2011.

[7] Web pages of Forester. Forester. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`, 2012 [cit. 2014-12-31].

[8] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS 2012*, volume 7214, pages 79–94, 2012.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[10] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 224–239. Springer International Publishing, 2013.

[11] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully Auto-

mated Shape Analysis Based on Forest Automata. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 740–755. Springer Berlin Heidelberg, 2013.

[12] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Softw.*, 6(5):38–48, 9 1989.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] Web pages of SV-COMP 2015. SV-COMP 2015. `http://sv-comp.sosy-lab.org/2015/`, 2014 [cit. 2014-12-31].