Pattern-based Verification for Trees[⋆]

Milan Češka, Pavel Erlebach, and Tomáš Vojnar

FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic.
e-mail: {ceska,erlebach,vojnar}@fit.vutbr.cz

Abstract. Pattern-based verification trying to abstract away the concrete number of repeated memory structures is one of the approaches that have recently been proposed for verification of programs using dynamic data structures linked with pointers. It proved to be very efficient and promising on extended linear data structures. In this paper, we overview some possibilities how to extend this approach to programs over tree structures.

1 Introduction

This paper addresses the problem of formal verification of *programs manipulating dynamic data structures linked by pointers* (such as lists, trees, etc.). In such programs, many mistakes can be easily made since the source code is usually not very transparent, and the functionality of the program is not apparent at the first sight. Consequently, a possibility of proving correctness of such programs is highly desirable. However, the verification of these programs is also quite complex as (1) they are infinite-state due to working with unbounded data structures, and (2) the objects that are manipulated here are in general unrestricted graphs.

The research on formal verification of programs with dynamic data structures is nowadays quite live, and there have appeared many different approaches in this area, such as [10, 11, 1, 5, 2, 9], differing in their formal roots, degree of automation, and the kind of program data structures and properties they can verify. Among the recently proposed methods for verification of programs with dynamic data structures there is also the so-called *pattern-based verification* [12, 4]. This method is based on detecting repeated adjacent subgraphs in heap graphs and collapsing them into a single summary occurrence (thus ignoring their precise number). The method has especially been studied in the context of verifying extended linear data structures, i.e. data structures with a linear skeleton and possibly some additional edges on top of it. Such structures, including non-circular as well as circular singly-linked and doubly-linked lists, possibly with additional pointers to the head, tail, etc., are very common in practice. Pattern-based verification on these structures has been made fully automated [4, 8, 7, 3]. The method can verify properties like absence of null pointer dereferences, dangling pointers, absence of garbage, but also more complex safety properties (e.g. shape invariance).

On the above described kind of extended linear structures, pattern-based verification turned out to be quite efficient. A natural question is then whether and how it can be

⋆ This work was supported by the Czech Grant Agency within projects 102/05/H050 and 102/07/0322, by the Czech-French project Barrande 2-06-27, and by the Czech Ministry of Education under the project *Security-Oriented Research in Information Technology*, contract CEZ MSM 0021630528.

extended to programs over tree-like structures. That is why, in this paper, we discuss the problems that appear in pattern-based verification when one proceeds from linear structures to tree-like ones (including tree structures with parent pointers and perhaps other additional pointers, such as root pointers, pointers to separately allocated data nodes, etc.). These problems include a need to work with a different kind of patterns and also a state space explosion problem that appears when one tries to straightforwardly transform the methods from the linear setting into the tree setting. We outline a way of how to cope with this state explosion. The results we describe do not yet provide a fully satisfying solution, but provide a significant and promising improvement over the trivial generalisation of pattern-based verification from linear structures to trees. Due to space limitations, the paper provides just an overview of the results—more details can be found in [6].

Plan of the Paper. The paper is organized as follows. In Section 2, the main principle of the original linear variant of pattern-based verification is briefly recalled. In Section 3, the extension of the method to trees is discussed. In Section 4, some early experimental results are presented. Finally, we conclude by a discussion of the achieved results in Section 5.

2 Pattern-Based Verification

In pattern-based verification, sets of memory configurations are represented as *abstract shape graphs* (ASGs) which are abstract memory graphs with basically two types of nodes: simple nodes corresponding to particular concrete nodes allocated in the memory (a list node, a tree node, etc.) and *summary nodes* behind which two or more "similar" concrete nodes are hidden (the exact number is abstracted away—note that such nodes cannot be, e.g., pointed by a pointer variable as this would clearly make them distinguishable from each other). The simple nodes are labelled with pointer variables pointing to them whereas summary nodes are labelled by the associated *pattern*. The pattern represents a memory subgraph whose two or more adjacent, mutually interconnected *instances* (or occurrences) are hidden behind the summary node. The nodes are linked by edges corresponding to the pointer links binding them in the memory and are labelled by the selectors (like `next`, `previous`, `left`, `right`, etc.) to which the appropriate pointer links correspond.

When verifying a program within pattern-based verification, one iteratively computes the set of all ASGs reachable at each program line. Particular program statements are performed on ASGs in three phases: (1) The given ASG is first (partially) *materialised* if the manipulation would cause a pointer variable point to a summary node (e.g., via a statement like `x = y->next`), which must not happen as the target of the pointer variable in such a case would not be well defined. That is why one concrete occurrence of the appropriate pattern hidden behind the summary node is explicitly instantiated and then used. (2) The statement is performed on concrete memory nodes as usual. (3) The resulting graph is *summarised*, i.e. searched for repeated adjacent occurrence of patterns that are then merged into a single summary node.

The idea of pattern-based verification has first appeared in [12] in a semi-automated framework whose user had to supply not only the input configurations and the program to be verified, but also the patterns to be used for abstraction. In the work, linear structures extended with possibly additional pointers to some shared nodes (like singly-linked or doubly-linked lists with head/tail pointers) were considered. The approach was able to handle only a single pattern which restricted its applicability to dynamic structures with a simple, fixed inner structure. In [4, 8, 7, 3], several extensions were introduced which made pattern-based verification over extended linear structures fully automated (by providing an automated discovery of patterns) and enhanced its generality (through a more general definition of a pattern and a possibility to work with more than one pattern).

In the rest of the section, we will say a bit more on the principles of automated pattern-based verification on extended linear structures in order to be able to contrast them with the tree case presented in Section 3.

2.1 The Main Principles of Linear Pattern-Based Verification

Memory patterns considered in [4] for extended linear structures are defined as a 5-tuple $P = (N^P, e^P, x^P, S^P, E^P)$ where N^P is a set of nodes, $e^P \in N^P$ is an entry node, x^P is an exit node ($e^P \neq x^P$), $S^P \subset N^P$ is a set of the so-called *shared* nodes, and $E^P \subseteq N^P \times Sel \times N^P$ is a set of edges of the memory pattern (labelled by pointer selectors). The entry and exit nodes delimit the linear skeleton of the pattern and represent the main connection points of instances of the pattern in concrete memory graphs to their surroundings. The shared nodes are shared among all instances of the patterns (like the head element of a list pointed to by head pointers) and play a somewhat similar role as global variables in programs with recursive procedures. The remaining nodes are internal to the pattern and thus also to its instances.

An *automatic discovery of patterns* is activated before every summarisation attempt (in [4], this holds only till some pattern is found since [4] is still restricted to a single pattern; this restriction is relaxed in [3]). The discovery of patterns consists in exploring all nodes of encountered shape graphs and searching for subgraphs that can be delimited by an entry and exit node (and perhaps shared nodes) and that appears at least twice in the given shape graph (before and after its original occurrence). For illustration, the pattern detected when working with singly-linked lists (with data abstracted away) is $P_1 = (\{e, x\}, e, x, \{\}, \{(e, \mathtt{next}, x)\})$. The pattern discovered in doubly-linked lists with data stored in separately allocated nodes and with tail pointers is $P_2 = (\{e, x, t, d\}, e, x, \{t\}, \{(e, \mathtt{next}, x), (x, \mathtt{prev}, e), (e, \mathtt{data}, d), (e, \mathtt{tail}, t), (x, \mathtt{tail}, t)\}$.

Summarisation consists in searching a given shape graph for subgraphs isomorphic with known patterns. If at least two adjacent instances of some patterns are found (linked via the exit/entry nodes), all nodes belonging to these instances are replaced with a single summary node (apart from the exit node of the last instance). Some further restrictions apply ensuring correctness and reversibility of the summarisation—e.g., no program variable can point to any node of any of the detected instances of the pattern.

We then distinguish a complete and a partial *materialisation*. In a complete materialisation, the summary node is replaced by two new instances of the pattern, while in a partial materialisation, one new instance of the pattern is inserted into the graph (followed by the summary node). The former case corresponds to having just two instances of the pattern hidden behind the summary node whereas the latter to having three or more. Both of these variants must always be explored as we do not record the exact number of summarised instances. In the case of a partial materialisation, we further distinguish a forward and a backward materialisation depending on the mutual positioning of the preserved summary node and the materialised instance of the pattern.

3 Pattern-Based Verification on Trees

The main contribution of this paper is an extension of pattern-based verification to programs manipulating *tree structures*. This extension implies changes in all parts of the method and also a need for some optimisations as a serious state space explosion problem appears. Due to space restrictions, all the changes are only briefly outlined in the section, a detailed description together with formal definitions can be found in [6].

3.1 Patterns and Their Discovery

While in the linear variant of pattern-based verification there was just one exit node in a memory pattern, for trees there must be more exit nodes to allow us to cope with the branching of the structure. So, now, a pattern is a tuple $P = (N^P, e^P, X^P, S^P, E^P)$ where $X^P \subset N^P$ is *the set of exit nodes* and other symbols keep their original meaning. The pattern of a binary tree would have two exit nodes, in case of ternary trees, there will be three exit nodes, etc.

The algorithm of discovering patterns is very similar to the linear case with the difference that it is needed to recognise and distinguish the *parent* nodes. While in the linear case, it was not necessary to distinguish predecessors and successors (and, e.g., in the case of doubly-linked lists, this would even lack sense as predecessors and successors are structurally indistinguishable), in trees, the role of a parent is much different than that of successors. Its identification is needed in order for the search of patterns to be restricted to the nodes below the parent only. The reason is illustrated in Fig. 1. For simplicity, the names of selectors are omitted. In Fig. 1(a), a pattern that we would like to be discovered in common binary trees with parent pointers is shown. Entry and exit nodes are named as e and x_1, x_2, respectively. A repeated materialisation of the pattern from a single summary node leads to the structure shown in Fig. 1(b). However, if we do not distinguish the role of parent nodes, in such trees, we can also detect the undesirable pattern shown in Fig. 1(c). If we tried to materialise this pattern repeatedly, we would obtain the structure shown in Fig. 1(d) which is not sensible as there would be two edges (marked with crosses) with the same selector leading out of a single node.

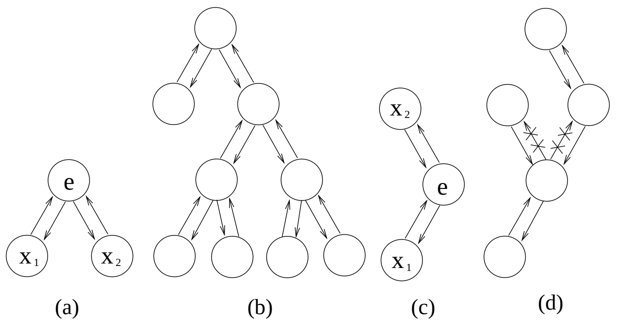

Fig. 1. Patterns in binary trees with parent pointers

3.2 Summarisation

A major change in summarisation in pattern-based verification on trees comes from the fact that trees of different sizes have different numbers of leaves. While summarisation of any (long enough) linear structure results in one and the same abstract shape graph with one exit node (the end of a list), summarisation of two trees of a different size, if done in the same way as for linear structures, results in different abstract structures that differ in the number of exit nodes (i.e. leaves), which are not summarised. These exit nodes together with the graphs rooted at them will be further called *tails*. Note that tails may really be subgraphs more complex than a single node: For example, they can have the form of a node with a self-loop, a node with an attached data node, a node pointed by a program variable, a beginning of a linear list attached to a tree, or generally an arbitrary subgraph that cannot be summarised.

To cope with the above, we propose a two-phase summarisation for trees. The first phase is equal to summarisation on linear structures (up to minor changes related to a re-definition of the pattern). The second phase consists in clustering the same tails into the so-called *multi-tails* which represent *two or more equal tails*. The heart of the second phase of the summarisation is the equivalence relation on the tails. Let us outline the main idea of the equivalence here—its formal definition is rather technical and can be found in [6]. Two tails are equivalent iff they are both linked to the same summary node in the same way (i.e. via the same edges in both directions), they are isomorphic, and no pointer variable points to any node of any of the tails (tails with a program variable pointing into them are always unique).

After the first phase of summarisation, the tails are partitioned according to the described equivalence. Unique tails are then left unchanged, while the others are treated in the following way. From every non-singleton set of the partition, one random representative is chosen and kept in the shape graph (just the type of the edges leading to it is changed to a special value specific for multi-tails), and the other tails are deleted. In this way, it is ensured that arbitrary sets of reachable trees of arbitrary forms will always summarise to a finite number of (finite) abstract shape graphs.

3.3 Materialisation

The changes in materialisation when using patterns over trees are the most complex and are related mainly to the multi-tail concept and to the increased number of exit nodes. Materialisation of a summary node with a left and a right multi-tail, which represents the simplest binary tree structure (and which we consider as an illustration example in the rest of the section), results in over 40 new shape graphs. Why such a high number?

Like in the linear case of a partial materialisation, a new, concrete instance of a pattern is inserted into the place of the original summary node. However, as the pattern has multiple exit nodes, its materialised instance will not be followed by a single new summary node as in the linear case, but by several new summary nodes (two for binary trees) where each of them represents a subtree of the materialised node. Each of these new summary nodes can have various combinations of tails and multi-tails since it is needed to cover all possible irregularities of subtrees of the materialised node. For example, in the materialisation mentioned above, the new summary nodes may have two multi-tails (the subtrees rooted at them are again proper trees with at least two left and two right tails), or a left multi-tail and a right tail (the subtrees have two or more left tails, but only one right tail), or just a right multi-tail (the subtrees have at least two right tails and no left tails), etc. This is the main reason of the high number of the resulting shape graphs.

Moreover, analogously to a complete materialisation in the linear case, one has to also consider replacing the new summary nodes by a concrete instance of the pattern. Finally, we have to even consider a new possibility specific for the tree setting, namely, the possibility of completely omitting the new summary node: If there are only two instances of a pattern hidden behind a summary node, one of them takes the place of the summary node, one goes to one of the branches, and the other branch is left empty).

Let us note that not all combinations of tails and multi-tails arising as described above are admissible—e.g., if both new summary nodes possible in the discussed example are replaced with single concrete nodes, the condition that *one multi-tail represents two or more tails* will not hold. In the original abstract shape graph there was a left multi-tail, and so in all resulting graphs there must be at least one left multi-tail, or at least two left tails. Otherwise, some left successors are lost. The same holds for the right tails too.

As we already mentioned, materialisation of the simplest tree structure results in over 40 new shape graphs. In the case of one additional pointer pointing to one of the tails, the number gets almost 100, and in the case of ternary trees with one additional pointer, it becomes over 2000 structures resulting from one materialisation operation. This would make the verification infeasible, and so some optimisations are needed.

3.4 Optimisations of the Materialisation

The reason of the state explosion within the basic materialisation on trees mentioned above lays in a too precise representation of the various irregularities of trees. But, our experiments show that in practical programs, it is usually not necessary to distinguish between, e.g., a tree with one right leaf (i.e. a leaf that is the right son of its father) and a tree with two right leaves, which is, however, a distinction that we enforce by defining multi-tails as covering two or more tails.

If we change the understanding of a multi-tail from *two or more tails* to *one or more tails*, we achieve a significant reduction of the materialisation state space. A materialisation of a summary node with two multi-tails (the same case that we used as an illustrative example in Section 3.3) would then not result in shape graphs where the new summary nodes can have normal tails (e.g. a summary node with a left tail and a right multi-tail is fully covered with a more general summary node with two multi-tails since this summary node now represents a subtree with one or more left and one or more right tail). The number of the resulting shape graphs is reduced to 10.

How would the situation look like if we continue in the above direction and set the meaning of the multi-tail to *zero or more tails*? A materialisation of the summary node with two multi-tails that we considered above would result in only two structures. In the first resulting shape graph, the materialised summary node would be replaced by an instance of the pattern where all exit nodes are summary nodes with two multi-tails (an analogy to the partial materialisation), and in the second resulting graph, the materialised summary node would be replaced by a single tail (an analogy to the complete materialisation). There would not be needed any combinations of tails and multi-tails of summary nodes since a new summary node with two multi-tails would cover all possibilities of concrete subtrees (even the irregular ones like a tree reduced to a list and so on). Thanks to this reduction of the materialisation state space, the verification time is cut from hours or days down to seconds or minutes in the examples we consider in

Section 4. The optimisations cause a slight increase of imprecision of the abstraction, but in the practical examples we considered, it does not make any difference.

4 Experiments

After the optimisations mentioned in the previous section, the verification times become reasonable for most library procedures manipulating binary trees. The times in seconds (if not explicitly declared otherwise) that we obtained from an initial implementation of our method are shown in Table 1. The prototype was implemented in SWI Prolog, and the tests were run on a PC with an AMD Athlon 2GHz processor.

The procedures mentioned in Table 1 are the following: `search` performs a random search in a tree (as the data contents is abstracted). The `deleteAll(*)` procedures delete all nodes of a tree, for which `deleteAll` exploits the parent pointers, while `deleteAll*` does not—in its case, the deletion consists in a repeated deletion of the leftmost leaf (repeatedly searched starting from the root). The `insertLast` and `deleteLast` procedures insert and delete a random leaf, respectively. The `tree2list` procedure converts a tree to the linear list via the postfix traversal. The `insert` and `delete` procedures insert and delete random node into/from a tree, respectively. Finally, `DSWtraversal` performs the Deutsch-Schorr-Waite tree traversal, namely the Lindstrom variant.

procedure	Binary trees	Binary trees with parents
search	1.22	2.26
deleteAll	-	1.80
deleteAll*	4.26	6.58
insertLast	1.42	2.64
deleteLast	8.00	12.30
tree2list	52.40	77.44
insert	6.00	12.30
delete	459.2	840.7
DSWtraversal	2.6h	5.1h

Table 1. Verification times for some procedures manipulating trees (in seconds if not stated otherwise)

In all the cases, we automatically verified that no null dereference and no memory leakage can occur. Moreover, using the generated reachable abstract shape graphs, we were able to manually verify various other safety properties (such as shape invariance, etc.). Let us note that the time needed to verify the `delete` procedure was higher due to the nodes manipulations used when a node with both children is deleted. The rightmost leaf of the left subtree has to be found, exchanged with the node which should be deleted, and then the node can be deleted as a leaf. The verification of `DSWtraversal` needed even more time because of its vast abstract state space caused by its use of four program variables pointing to the tree with a large number of combinations of positions of these variables in the tree.

5 Conclusion

In this paper, we briefly introduced an extension of the pattern-based verification method from programs over extended linear data structures to programs over tree structures, including tree structures with parent pointers and perhaps some other additional pointers (like root pointers, pointers to separate data nodes, etc.). We have especially discussed the state explosion in the materialisation (i.e. concretisation) step, which occurs in this setting due to having to deal with all various irregularities of trees, and ways how to deal with this problem.

The verification times that we obtain from our early prototype implementation are one to two orders of magnitude higher than when handling linear structures. They are still not yet fully satisfactory, but they are several orders of magnitude better than when using a straightforward extension of the principles of pattern-based verification from linear structures to trees. Other existing tools capable of handling programs over trees can sometimes provide better verification times, but are often less general (like the grammar-based shape analysis [9]) or less automated (like PALE based on the WSkS logic and tree automata [10] or TVLA based on first-order predicate logic with transitive closure [11]).

The remaining efficiency problem of the analysis is that it still preserves relatively a lot of information about the various irregularities that may arise in trees. If pattern-based verification is to achieve similarly nice results on trees as on extended linear structures, some further optimisations are still needed, perhaps in the form of some sort of a counterexample-guided abstraction refinement loop allowing one to drop more information about the structure and then reclaim it on demand.

Interestingly, the feature of our analysis of keeping a relatively precise information about the structures handled could become an advantage if the analysis was applied to a dynamic data structure with a complicated internal scheme (e.g. trees whose nodes have an attached linked substructure of a fixed form such as a circular list with four nodes, etc.). Such structures could be handled by our analysis with no additional need of a manual intervention or a dramatic increase of the verification time.

References

1. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. *Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking.* In *Proc. of TACAS'05, LNCS* 3440. Springer, 2005.
2. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. *Abstract Regular Tree Model Checking of Complex Dynamic Data Structures.* In *Proc. of SAS'06, LNCS* 4134. Springer, 2006.
3. M. Češka, P. Erlebach, T. Vojnar *Generalized Multi-Pattern-Based Verification of Programs with Linear Linked Structures.* Accepted to *Formal Aspect of Computing*, Springer, 2006.
4. M. Češka, P. Erlebach, T. Vojnar *Pattern-Based Verification of Programs with Extended Linear Linked Data Structures.* In *Proc. of AVOCS'05, ENTCS*, 145:113–130, 2006.
5. D. Distefano, P.W. O'Hearn, H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
6. P. Erlebach. *Automated Formal Verification of Programs Working with Dynamic Data Structures.* PhD. thesis, Brno University of Technology, Czech Republic, to appear during 2007.
7. P. Erlebach *Automated Pattern-Based Verification for Tree Structures.* In *Proc. of the PhD Student Workshop MEMICS'06*, 2006.
8. P. Erlebach *Towards a Systematic Framework for Automatic Pattern-Based Verification of Dynamic Data Structures.* In *Proc. of the PhD Student Workshop MEMICS'05*, 2005.
9. O. Lee, H. Yang, and K. Yi. *Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis.* In *Proc. of ESOP'05, LNCS* 3444. Springer, 2005.
10. A. Møller, M. Schwartzbach. *The Pointer Assertion Logic Engine.* In *PLDI '01*, 2001.
11. S. Sagiv, T.W. Reps, R. Wilhelm. *Parametric Shape Analysis via 3-Valued Logic. TOPLAS*, 24(3), 2002.
12. T. Yavuz-Kahveci. *Specification and Automated Verification of Concurrent Software Systems.* PhD. thesis, Computer Science Department of University of California, Santa Barbara, CA, USA, 2004.