Abstract Regular Tree Model Checking of Complex Dynamic Data Structures – Implementation Details

Adam Rogalewicz

FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic.
e-mail: rogalew@fit.vutbr.cz

Abstract. This article describes some implementation details used in our prototype tool for verification of programs manipulating dynamic data structures. This tool is based on the automata framework. We encode data structures into trees and sets of trees as finite tree automata. The program behaviour is encoded as a tree transducer. Then the abstract regular tree model checking technique can be applied to compute a set of all reachable configurations.

1 Introduction

Automated verification of programs manipulating dynamic linked data structures is currently a very live research area. This is partly due to the fact that programs manipulating pointers are often complex and tricky, and therefore methods for automatically analysing them are quite welcome, and also because automated verification of such programs is not easy. Programs manipulating dynamic linked data structures are typically infinite-state systems, their configurations have in general the form of unrestricted graphs (often called *shape graphs*), and the shape invariants of these graphs may be temporarily broken by the programs during destructive pointer updates.

In [5], we propose method based on technique *regular tree model checking* [9, 13, 6]. In regular tree model checking, configurations of the system being examined are encoded as trees over a suitable ranked alphabet, sets of configurations are described by tree automata, and transitions of the system are encoded as tree transducers. Then the set of all configurations reachable from an initial set of configurations is computed by repeatedly applying the tree transducers on the set of the so-far reached configurations (encoded as tree automata). In order to make the method terminate as often as possible and to fight the state explosion problem arising due to increasing sizes of the automata to be handled, various kinds of automatically refinable abstractions over automata are used. The technique combining regular tree model checking with automated abstractions is called *abstract regular tree model checking* [4], and it is successful on a lot of case studies.

In order to be able to apply ARTMC for verification of programs manipulating dynamic linked data structures, whose configurations (shape graphs) need not be tree-like, we propose an *original encoding of shape graphs based on tree automata*. We use trees to encode the *tree skeleton* of a shape graph. The edges of the shape graph that are not directly encoded in the tree skeleton are then represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (as, e.g., left up, right down, etc.) and the kind of nodes that can be visited on the way. Both the tree skeletons and the routing expressions are automatically discovered by our method. The idea of using routing expressions is inspired by PALE [12] and graph types [11] although there, they have a bit different form (see below) and are defined manually.

The complete description of this method can be found in [5]. This method was prototypely implemented in a tool based on MONA [10] GTA library, and a series of case studies were verified for safety properties. This article brings some more details about the prototype implementation – encoding of transducers as mona-automata (preliminary proposals are in [1]), implementation of finite-height abstraction, automatic creation of routing expressions, and initial partition of automata states.

2 MONA

Mona [10] is a tool which was designed for decision of WS1S and WS2S (Weak Second-order Theory of One or Two successors) formulas validity. WS1S is a fragment of arithmetic augmented with second-order quantification over finite sets of natural numbers. Mona is based on the relation between the logic and the finite (tree) automata theory. For each WS1S formula, there exists a corresponding finite automaton. For each WS2S formula there exists a corresponding tree automaton. Validity of a formula is equal to non-emptiness of the corresponding automata. The conjunction of formulas is related to the intersection of automata, the formulas disjunction to the automata union, the negation to the automata complementation. Mona receives a formula, converts it into the corresponding automaton and checks for emptiness.

The implementation of Mona contains two interesting libraries. The first one for finite automata and the second one for guided tree automata (GTA). Both are based on BDDs (binary decision diagrams) [7] with a strong emphasis to efficiency and complexity. Instead of alphabet symbols, binary codes are used. Bits in binary coded symbols are numbered from 0, and some bits can be undefined. Example of BDD can be seen on figure 1. In this BDD, we distinguish alphabet symbols using the first 4 bits. Note, that not all bits must be used to distinguish the output. The input node is called BDD-root. There can be defined several BDD-roots – e.g. one for each automaton state. Leaves of the BDD define the value associated with concrete BDD-root, and the symbol of alphabet. This definition is much more efficient than the classical sequential definition of automata rules.

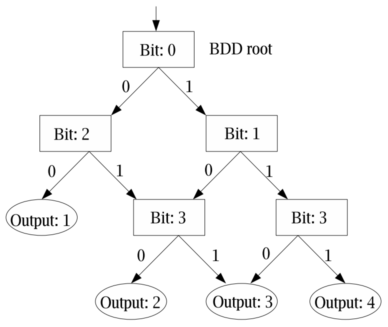

Fig. 1. An example of a binary decision diagram (BDD)

Loops is BDDs are forbidden, but a sharing between BDD paths is required. BDD nodes with an equal behaviour are automatically merged. Therefore if equal rules are defined from two automata states, this states will be manipulated by the same BDD-root. We use the GTA library to manipulate bottom-up tree automata. The class of GTA contains whole class of bottom-up tree automata. Bottom-up tree automaton [8] is a generalisation of finite automaton. It is a finite state machine, which starts with several heads (one at each leaf), and reads the tree from leaves to the root.

Remark: General tree automata can accept n-ary tree (with n-ary branching), but the GTA library supports only binary ones (with the branching factor two). This is not a huge restriction, because we can introduce an encoding of n-ary trees into binary ones.

2.1 Transduction

The GTA library does not provide transducers. However, as we proposed here, it allows us to encode structure preserving transducers as automata.

The idea is to define symbols of alphabet only at even bits. Regular automata then use just even bits in BDDs. Transducers are encoded as normal automata, where the input is encoded in even bits, and the output in odd bits. The application of transducer is done in following way:

1. Intersection between automaton and transducer
2. Projection of all even bits (the result of this step contains only odd bits)
3. Remapping of odd bits to even ($1^{st} \rightarrow 0^{th}, 3^{rd} \rightarrow 2^{nd}, \ldots$)

The result is an automaton, where each accepted tree is reachable (using the transducer) from some tree accepted by the original automaton. The reverse application of a transducer is done by same 3 steps, but in a different order – (1) remapping (even bits to odd), (2) intersection with the transducer, (3) projection on even bits.

2.2 Finite Height Abstraction

The first proposed abstraction method to accelerate the fixpoint computation is an *abstraction based on languages of trees of a finite height*. It defines two states equivalent, if their languages up to the given height n are equivalent. There is just a finite number of languages of height n, therefore this abstraction is finitary. A refinement is done by an increase of the height n. The abstraction works as follows:

1. First, we create two initial classes of states – final ones, and non-final ones.
2. Two states q_1, and q_2 are equal, if $\forall v \in Alphabet \wedge \forall [v(i,j) \rightarrow q_1] : \exists [v(eq(i), eq(j)) \rightarrow q_2]$ and via versa (change q_1 and q_2), where i, j are states, $eq(x)$ is a state in equal class as x (classes are created in step 1, or 3), and $[x(a,b) \rightarrow c]$ is an automaton rule.
3. New classes are created according to the result of the previous step.
4. Steps 2, and 3 are iterated n-times, where "n" is the bound of the abstraction.

The implementation of steps 2, and 3 is done as follows: For each automaton state "s", we compute the following matrix of the size $n \times n$ (n is actual number of classes). A field $(class_a, class_b)$ of this matrix associated to classes $class_a$ and $class_b$ contains a set of alphabet symbols $\{x | a \in class_a, b \in class_b, x(a,b) \rightarrow s\}$. Only states with exactly equal matrices can stay in the same class.

2.3 Predicate Based Abstraction

The second proposed abstraction is an *abstraction based on predicate languages*. Let $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

In our implementation, one automaton defines a set of predicates – each state can be chosen as a final state of the automaton. This allow us to check intersections with all predicates defined by one automaton as follows: $\mathcal{A}$ is an automaton on which the abstraction is applied, $\mathcal{P}$ is an automaton describing predicates, $init(A)$ is an initial state of the automaton A, $States(A)$ is number of states in the automaton A.

1. Create matrix *block* of the size $States(\mathcal{A}) \times States(\mathcal{P})$, and fill it by 0's.
2. Set $block[init(\mathcal{A}), init(\mathcal{P})] = 1$
3. Create a stack of tuples, and push $(init(\mathcal{A}), init(\mathcal{P}))$ into this stack.
4. Pop tuple (a, b) from the stack
5. $\forall i \in \mathcal{A}\ \forall j \in \mathcal{P} : block[i, j] = 1$ do:
   - If $s \in Alphabet : s(a, i) \rightarrow x \wedge s(b, j) \rightarrow y \wedge block[x, y] = 0$, set $block[x, y] = 1$ and push (x, y) into the stack.
   - If $s \in Alphabet : s(i, a) \rightarrow x \wedge s(j, b) \rightarrow y \wedge block[x, y] = 0$, set $block[x, y] = 1$ and push (x, y) into the stack.
6. If stack is not empty, then continue by step 4.
7. Two states i, and j are equal, if fields block[i], and block[j] are equal.

3 Encoding of Memory Configurations

To be able to use abstract regular tree model checking [4], we need to encode program configurations as trees, and its sets as tree automata. The problem is, that dynamic data structures are in general oriented graphs (called shape graphs). Therefore we proposed an encoding of the graphs into the trees [5]. The main idea is, that a tree is used just as a backbone, and edges between nodes are defined as expressions over directions in this backbone, and over values of nodes.

Let $\mathcal{V}$ be a set of pointer variables, $\mathcal{M}$ a set of markers, $\mathcal{D}$ a finite set of data values, and $\mathcal{R}$ a set of so-called pointer descriptors. Then the tree encoding of a shape graph has the following form: At the top of the tree is located a node containing a program location (a line of the program). The second node from the top contains undefined pointer variable – it is a symbol from the alphabet $2^{\mathcal{V}}$. The third node contains null pointers – also a symbol from alphabet $2^{\mathcal{V}}$. Under this node is located the top node of the heap contents – the top memory node. Each memory node (top or inner) can have n sons – inner memory nodes. The memory node contains symbol of the alphabet $2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\perp, \top\})^k$. This symbol encodes pointer variables pointing to this node, markers set in this node, the data value of this node, and n so-called pointer descriptors – each for one next pointer beginning from this memory node. This descriptors are symbolic names – references to routing expressions. "$\perp$" is a special pointer descriptor denoting *null* pointer, and "$\top$" denotes an undefined pointer.

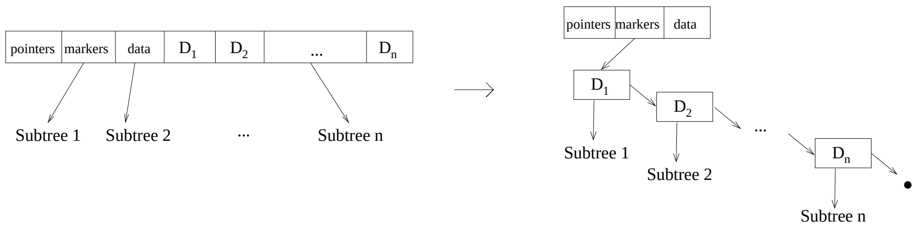

Fig. 2. Splitting memory nodes in Mona into data and next pointer nodes

Routing expressions are regular expressions over the directions in tree, and determine the destination of the pointers. There is a finite number of routing expressions which are updated during the computation. Each routing expression is paired with a marker – only a marked node can be a destination of the pointer (this decreases the non-determinism of routing expressions).

As the library supports binary trees only (we need n-ary ones), we *split each memory node* labelled with $\Sigma_2 = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\perp, \top\})^k$ in the above definition of a tree memory encoding into a data node labelled with $2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D}$ and a series of k next pointer nodes, each labelled with $\mathcal{R} \cup \{\perp, \top\}$—cf. Fig. 2.

3.1 Routing Expressions

In our prototype implementation, we encode routing expressions as a transducers moving token (selected bit in BDD representation) from a source node to a destination one. The destination node must contain the marker paired with the routing expression.

Automatic Creation of Routing Expression After execution of a command $x.next = y$, it is necessary to update pointer expression assigned to this command. This routing expression must cover all newly added combinations of sources and destinations. The source nodes contain the pointer variable x, and the destination nodes variable y (position of x and y is unique in each tree). We need to extract the relation between this two pointer variables. For this extraction, we use a special transducer. This transducer transforms the input automaton to another one with a restricted alphabet (only 4 symbols) in the following way: Node without x and $y \rightarrow 00$, node containing $x \rightarrow 10$ node containing $y \rightarrow 01$ node containing both x and $y \rightarrow 11$.

After applying this transducer, we received an automaton, where the relation between x and y is preserved. (remark: we work with a set of trees *leadsto* we obtained a set of relations). We call this automaton a shape automaton. In addition, this automaton contains also information about a position of these variables in the tree. But we need to extract just the relation between x and y. Therefore we modified this shape automaton in the following way: Everything below and above the shortest path between x and y is substituted by 00^* (an unbounded number of reading of the symbol 00). This can not be done by a transduction – a special procedure is used. This procedure go through the automaton and search for states under, and over the shortest path. After applying of this procedure, we receive an automaton, where x can be placed almost everywhere (not all positions allows to place y according to relation) in the tree, and y is placed at the position related to x.

Now, we can create a routing expression (transducer) according to this shape. The transducer is created by the intersection of the modified shape with a 1-state template transducer. This template transducer has four types of rules starting from the 3^{rd} bit of BDD. The first 2 bits are used according to the shape: 00 – copying rule, 10 – check for presence of the token end delete it, 01 – check that the token is not set, and set it, 11 – copy node, and check for token (next pointer points to the current node).

After the intersection, we project out first 2 bits, and remap bits to get an ordinary transducer ($2 \rightarrow 0, 3 \rightarrow 1, 4 \rightarrow 2, \ldots$). Now we have the desired routing expression.

Cover Check The creation of routing expressions is due to a set of projections an expensive process. In most of the cases, it is not necessary to run it, because the actual routing expression is sufficient. Therefore we introduce a cover check procedure to decide whether to run the creation, or not.

The cover check is a simple procedure. At the beginning we have a set of trees, where in each tree, the source node is marked by the variable x, and the destination one contains the variable y, and a marker (paired with the tested routing expression). Then we apply the following steps:

1. Set the token to the node containing the variable x.
2. Delete markers paired with the tested routing expression from nodes without the variable y
3. Apply the routing expression.
4. Apply the routing expression in a reverse way (as the reverse transducer) on the result from the previous step.
5. Check, whether the original set obtained by the step 2 is included in the result of the step 4.

If there is some tree with the combination of a source and a destination not covered by the routing expression, this tree will be lost after the step 3. Then this tree is not included in the result obtained after the backward transition (step 4).

3.2 Initial partition

As it was described in section 3, the tree encoding consists from different types of nodes – memory nodes, next pointer nodes, etc. To ensure, that abstraction does not collaps nodes with different meanings (and degrade the memory encoding), we use an initial partition of automata states. At the beginning, all states are supposed to have an undefined type. The types will be assigned during the initial partition.

First, we separate initial, final, and rejecting (no tree is accepted from a rejecting state) states into separate classes. Now, we separate states accepting undefined pointers definition – states $\{x | val(x, initial) \rightarrow final\}$, where val is an arbitrary symbol of alphabet, $initial$ is an initial state, and $final$ is a final state. The symbol val represents a program location (number of line in program). The states accepting null pointer definition are separated after undefined ones – states $\{x | val(x, initial) \rightarrow undef\}$, where val is a symbol beginning with 00 (in binary encoding) of alphabet, $initial$ is an initial state, and $undef$ is an *undefined* state. After undefined states, we distinguish states accepting top memory nodes – $\{x|11.^*(i,j) \rightarrow x\}$, where $11.^*$ is an alphabet symbol beginning with 11 (in binary representation), and i, j are not-yet-sorted states.

Now, only inner memory nodes, and next pointer nodes are unseparated. The searching for states accepting memory nodes is done in equal way as searching for top memory nodes. Only symbols from alphabet beginning with 10 are used – $\{x|10.^*(i,j) \rightarrow x\}$. The rest of the states is declared as states accepting next pointer nodes.

This initial partition can be improved by one of the following two possibilities. (1) The states accepting next pointer nodes are separated into several classes with respect to the *level* of next pointer node – next pointer nodes of different levels can not be collapsed, and the data structure shape is preserved. (2) Each state accepting a next pointer node can be put into separate class. This exclude collapsing on states accepting next pointer nodes. Experimental results showed, that best results are obtained when collapsing on next pointer nodes is forbidden.

4 Experimental Results

This section is a short review of experimental results described in [5]. We have performed several experiments with singly-linked lists (SLL), doubly-linked lists (DLL), trees, lists of lists, and trees with linked leaves.

Table 1 contains verification times for the experiments mentioned above (the "+ test" in the name of an experiment means that some shape invariants were checked). We give the best result obtained using the three mentioned abstraction schemas and say for which abstraction schema the result was obtained. The note "restricted" accompanying the abstraction method means that the abstraction was applied at the loop points only. The experiments were performed on a 64bit Xeon 3,2 GHz with 3 GB of memory. The column $|Q|$ gives information about the size of the biggest encountered automaton, and N_{ref} gives the number of refinements.

Table 1. Results of experimenting with the prototype implementation of the presented method

Example	Time	Abstraction method	$\|Q\|$	N_{ref}
SLL-creation + test	1s	predicates, restricted	25	0
SLL-reverse + test	5s	predicates	52	0
DLL-delete + test	6s	finite height	100	0
DLL-insert + test	10s	neighbour, restricted	106	0
DLL-reverse + test	8s	predicates	54	0
DLL-insertsort	2s	predicates	51	0
Inserting into trees + test	29s	predicates, restricted	65	0
Linking leaves in trees + test	49s	predicates	75	2
Inserting into a list of lists + test	6s	predicates, restricted	55	0
Deutsch-Schorr-Waite tree traversal	57s	predicates	126	0

5 Conclusion

In [5], we have proposed a new, fully automated method for verification of programs manipulating complex dynamic linked data structures. The method is based on the framework of ARTMC. This method was fully implemented in the prototype tool, based on mona GTA library [10]. During the implementation, it was necessary to create a lot of non-trivial functions to manipulate shape graphs encoded as trees. Some of the implementation details are described in this article.

In the future, we would like to optimise the performance of our Mona-based prototype tool, e.g., by exploiting the concept of *guided tree automata* that are suggested as very helpful in many situations by the authors of Mona [2] and that we have not used yet. Also, there can be introduced more detailed abstraction methods designed specially for this encoding of shape graphs. This special purpose abstraction methods can improve the effi ciency of the ARTMC.

Our present method is designed for verifi cation of *safety properties*, but we would like to introduce also method for verifi cation of liveness properties. One possible way is to extend a method proposed by Bouajjani et al in [3] to structures with more than one selector. This extension brings a lot of new problems to be solved.

Acknowledgement. This work was supported in part by the Czech Grant Agency within projects 102/05/H050, 102/04/0780, 102/03/D211, and by the Czech-French project Barrande 2-06-27.

References

1. A.Rogalewicz. Towards Applying Mona in Abstract Regular Tree Model Checking. In *Proc. of Student EEICT'05*. FIT VUT, 2006.
2. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for Guided Tree Automata. In *Proc. of WIA'96*, volume 1260 of *LNCS*. Springer, 1997.
3. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, LNCS. Springer, 2006.
4. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infi nity'05*, 2005.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, LNCS. Springer, 2006.
6. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
7. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005.
   URL: `http://www.grappa.univ-lille3.fr/tata`.
9. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. *Theoretical Computer Science*, 256(1–2), 2001.
10. N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.
11. N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.
12. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.
13. P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.