Verifying Red-Black Trees*

Paolo Baldan[1], Andrea Corradini[2], Javier Esparza[3], Tobias Heindel[3], Barbara König[3], and Vitali Kozioura[3]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany

baldan@dsi.unive.it andrea@di.unipi.it
{esparza,heindets,koenigba,koziouvi}@fmi.uni-stuttgart.de

Abstract. We show how to verify the correctness of insertion of elements into red-black trees—a form of balanced search trees—using analysis techniques developed for graph rewriting. We first model red-black trees and operations on them using hypergraph rewriting. Then we use the tool AUGUR, which computes approximated unfoldings, in order to show that insertion preserves the property that there are no two consecutive red nodes in a tree, a requirement for red-black trees. Furthermore, we prove that the tree remains balanced by exploiting a type system that can be obtained as an instance of a general framework.

1 Introduction

In order to verify programs written in languages with dynamic memory allocation, such as C, it is important to find suitable abstractions for the dynamically evolving pointer structures on the heap. The same problem arises for object-oriented languages, for instance Java. Despite existing techniques such as alias and points-to analysis [20, 24] and shape analysis [19], this is still a major open problem. This paper proposes to use verification techniques based on graph rewriting. The basic idea is to represent the state of the heap by a graph and dynamic transformations of the pointer structure by graph rewriting rules. Compared to the approaches to shape analysis which represent these structures as models of a 3-valued logic we follow a more direct approach where pointer structures are represented as graphs, and graph morphisms can be used as a convenient abstraction mechanism. This allows us to exploit partial order semantics already developed for graph rewriting, as well as its close relation to Petri nets, which we use as abstractions (over-approximations) of the behavior of graph rewriting systems.

We demonstrate the effectiveness of this approach by modeling the insertion of elements into red-black trees and verifying (partial) correctness of the insertion operation. Red-black trees are binary search trees whose nodes are colored either

* Research partially supported by DFG project SANDS and EC RTN 2-2001-00346 SEGRAVIS.

black or red. Only inner nodes can be red, and the following two properties are satisfied: no red node has a red child, and the "black depth" is the same for all leaves. In order to re-establish these properties after a new element is inserted, it is necessary to perform some local transformations on the tree (called *rotations*), which have the effect of rebalancing it. After modeling rotations as graph rewriting rules, we use two different techniques to show that the two properties of red-black trees mentioned above still hold after an insertion. The first property is shown by automatically abstracting the graph rewriting system into a Petri net [2, 4] by means of the AUGUR tool. The second property is proved by resorting to a type-theoretical framework for graph rewriting, proposed in [9].

The rest of the paper is structured as follows. In Section 2 we introduce red-black trees and their representation as hypergraphs. In Section 3 we model insertion into red-black trees using graph rewriting. In Section 4 we show how to verify that insertion preserves the structural properties of red-black trees. Finally, in Section 5, we draw some conclusions.

2 Red-Black Trees

Red-black trees are a form of balanced search trees which can be easily implemented (see [11, 5]). They can also be seen as a variant of $(2, 4)$-trees.

Definition 1 (Red-black tree). *A red-black tree is a finite binary tree whose inner nodes are associated with keys. Keys are elements of a totally ordered set. A node can either be red or black. A red-black tree satisfies the following conditions:*

(S) *The tree is sorted, i.e., for every node v the maximal key in the left subtree is smaller than the key of v, and the minimal key in the right subtree is equal to or larger than the key of v.*
(RL) *The root and the leaves are black.*
(D) *All leaves have the same black depth, i.e., the number of black nodes on the path from the root is the same for all leaves.*
(R) *No path from the root to a leaf contains two consecutive red nodes.*

Due to these conditions the longest path from the root to a leaf is at most twice as long as the shortest one. The height of a red-black tree with n inner nodes is therefore $O(\log(n + 1))$, and thus we say that the tree is balanced.

Since we will model insertion into red-black trees by hypergraph rewriting, in the paper we always depict red-black trees as hypergraphs.

In the following, given a set A, let A^* denote the set of finite sequences of elements of A and for $s \in A^*$, let $|s|$ denote its length.

Definition 2 (Hypergraph). *Let Λ be a fixed set of edge* labels*, where every label $l \in \Lambda$ is associated with an* arity *$ar(l) \in \mathbb{N}$.*

*A (Λ-)*hypergraph *(or simply* graph*) is a tuple $G = (V_G, E_G, c_G, l_G)$, where V_G is a set of* vertices *and E_G is a set of* hyperedges*. Each hyperedge is attached to a sequence of vertices, as expressed by the* connection function *$c_G \colon E_G \to V_G^*$,*

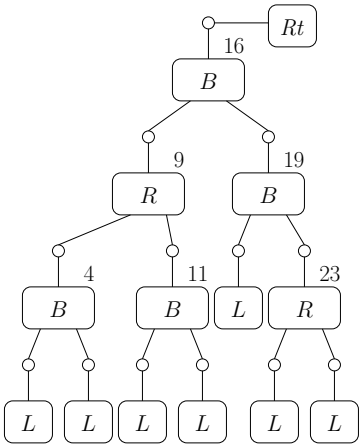

Fig. 1. An example of a red-black tree.

and it is labeled with an element of Λ via the labeling function $l_G: E_G \to \Lambda$. *For any hyperedge $e \in E_G$ it must hold that $ar(l_G(e)) = |c_G(e)|$, i.e., the number of nodes an hyperedge is attached to is determined by the arity of its label.*

Hypergraph morphisms $\varphi: G \to G'$ are defined, as usual, as structure preserving mappings (see also [18]).

A red-black tree is represented as a hypergraph where hyperedges correspond to the nodes of the tree. Inner nodes are represented by hyperedges of arity 3, i.e., they are connected to exactly three vertices, where the parent and the left and right children can be attached. They are labeled by either R or B depending on whether the node is red or black. Leaves are represented by unary hyperedges labeled L. Furthermore there is, for technical convenience, a single unary hyperedge labeled Rt, indicating the root node. Figure 1 depicts a red-black tree, where the keys are written next to the hyperedges. Note that, by definition of hypergraph, each hyperedge is connected to an *ordered* sequence of vertices. In our pictures, vertices are always arranged in such a way that the vertex above a hyperedge is its first vertex, whereas the remaining vertices are ordered counter-clockwise.

3 Insertion into Red-Black Trees using Graph Rewriting

We introduce now the concepts of graph rewriting rule and rewriting step, which will be used to model the insertion of a new node into a red-black tree.

Definition 3 (Graph rewriting rule). *A* graph rewriting rule *r is a tuple (L, R, α) where L and R are hypergraphs, called the* left-hand side *and* right-hand side *of the rule, while $\alpha : V_L \to V_R$ is an injective function.*

Intuitively, to apply a rule $r = (L, R, \alpha)$ to a hypergraph G one must find an occurrence of the left-hand side L in G, i.e., a hypergraph morphism $\varphi : L \to G$. The application of the rule first removes from G the image of the hyperedges of L, and then extends the resulting hypergraph by adding the new vertices in R (i.e., the vertices in $V_R - \alpha(V_L)$) and all the hyperedges of R, yielding a new hypergraph H. In this case we write $G \Rightarrow^r H$. Observe that, unlike hyperedges, vertices are never deleted: the vertices of G are not affected by the rewriting step. We refer to [2] for a discussion of this restriction with respect to more general definitions of graph rewriting. Notice, anyway, that the deletion of a vertex can be simulated in our framework by leaving it isolated in the resulting graph.

The insertion of a new node into a red-black tree is described by the hypergraph rewriting rules shown in Fig. 2 and Fig. 3. For the corresponding pseudocode, see for instance [11]. An interesting question, that we leave as a topic of future research, is whether and how graph rewriting rules can be synthesized automatically from (pseudo-)code.

In the following the mapping α of a rule is represented by numbering the nodes in the left-hand and right-hand sides: α maps a node in the left-hand-side to the node of the right-hand side with the same number. Furthermore keys are denoted by the letters y, z, u, v.

Rule [add-leaf] describes how a leaf is replaced by a new inner node labeled M and two leaves. The label M stands for "marker" and denotes a red node during the insertion phase. Rule [add-leaf] also consumes a "token", the 0-ary hyperedge *add*, that will be generated again when the insertion is completed: this mechanism prevents the concurrent insertion of nodes. We assume that the insertion of the new key y starts from the appropriate leaf, whose position must have been determined by a previous search on the tree. Although this is out of our focus, it is worth observing that this search could be realized by means of graph rewriting rules acting on attributed graphs [10].

The remaining rules describe the local transformations needed to ensure that the tree is converted into a red-black tree. If the marker has a black parent, it is converted into a red hyperedge and insertion terminates (rule [marker-black], this rule has two symmetric variants). If the marker is the root (rule [marker-root]), it is replaced by a black hyperedge; in this case the black depth of the tree increases by one. If the marker has a red parent, we distinguish several cases (notice that in this case the marker's grandparent (if any) must be black, because otherwise Condition (R) would be violated):

- If the red parent of the marker has a red sibling, we perform a flip and move the marker upwards (rule [flip], four variants). In this case the algorithm continues.
- If the red parent of the marker has a black sibling, and this sibling is not a leaf, we apply either rule [rotation] or rule [double-rotation]. Rule [rotation] is applied if the marker and its red parent are either both left children or both right children. In the two remaining cases rule [double-rotation] is applied. In all cases the algorithm terminates.

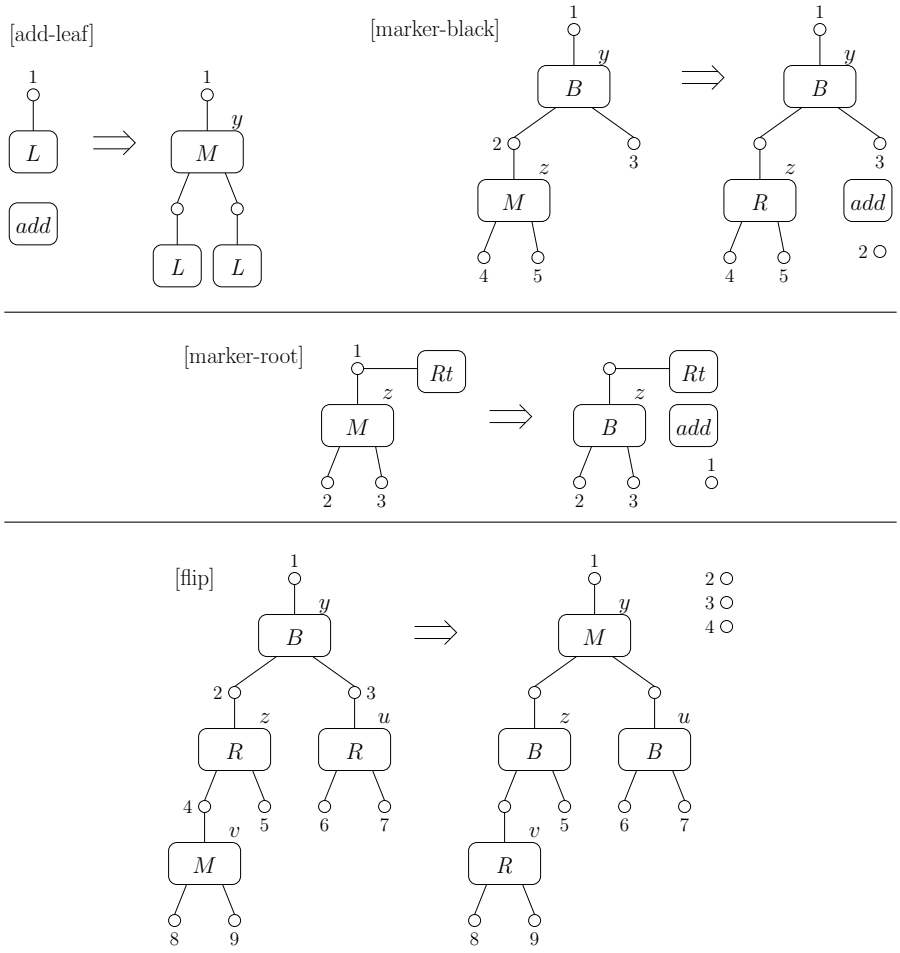

Fig. 2. Graph rewriting rules (insertion of an element into a red-black tree), part I.

– If the red parent of the marker has a black sibling, but this sibling is a leaf, we proceed similarly to the previous case. There are four more rules, obtained from those of Fig. 3 by replacing the node with key u by a leaf.

One can see fairly easily that all the transformations expressed by the above rules preserve the sortedness Condition (S) in Definition 1. Moreover, for any given finite tree, the insertion procedure started by rule [add-leaf] surely terminates, generating again the token *add*. The formal verification of these two properties goes beyond the goals of this paper: we shall only sketch in the conclusion how this could be done by exploiting the available theory of confluence and termination of graph rewriting systems.

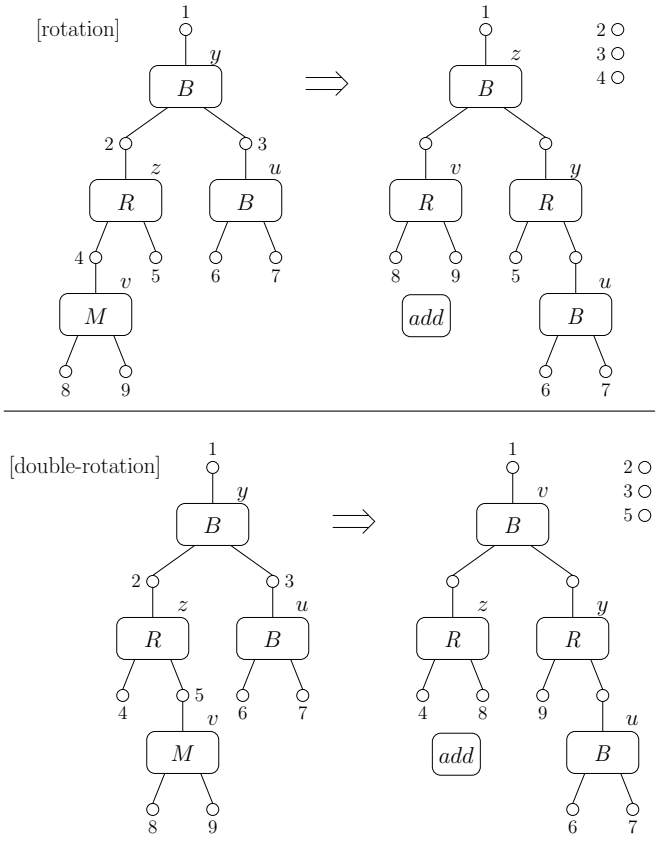

Fig. 3. Graph rewriting rules (insertion of an element into a red-black tree), part II.

Note that modeling insertion into red-black trees using graph rewriting rules is very natural. Similar diagrams can be found in most text books introducing red-black trees. Usually no marker is used, a red node takes its place instead. However, this would lead to "inconsistent" intermediate states, produced during the insertion procedure, which *do* contain two consecutive red hyperedges, violating Condition (R). We avoid this by using a specific marker, which is furthermore useful for indicating the position in the tree where operations have to be performed.

4 Verifying Red-Black Trees

In the following we describe two static analysis techniques developed for the verification of graph transformation systems: approximated unfolding and type systems. Approximated unfolding is a fully automatic technique, based on a partial order semantics of graph transformation systems. Here it is used to show that

no tree generated by the rewriting rules for insertion has two consecutive red nodes (Condition (R)). The property that red-black trees remain balanced (Condition (D)) is checked using a suitable type system, which is a simple instance of a general framework [9]. We assume that the preservation of Conditions (S) and (RL) has already been proved, as well as the fact that the result of the insertion procedure is again a tree.

As the rest of the paper concentrates only on structural properties of red-black trees, we neglect keys in the following.

4.1 Approximated Unfolding

Approximated unfolding was proposed in [2, 4] for the verification of infinite state systems, modelled as graph transformation systems. It is based on the unfolding construction which, applied to a graph transformation system, produces a static structure fully describing the concurrent behavior of the system, including all possible rewriting steps and their mutual dependencies, as well as all reachable states [17, 3].

The unfolding is infinite for any non-trivial graph transformation system. The mentioned papers propose an algorithm for constructing finite structures which can be seen as over-approximations of the full unfolding, where interesting classes of properties of the original system can be studied and verified. The structures used for approximation are so-called *Petri graphs*, consisting of Petri nets the places of which are hyperedges.

The outcome of the algorithm is determined by the chosen level of accuracy: essentially one can require the approximation to be exact up to a certain causal depth k, thus obtaining the so-called *k-covering* $\mathcal{C}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$.

The covering $\mathcal{C}^k(\mathcal{G})$ over-approximates the behavior of $\mathcal{G}$ in the sense that every computation in $\mathcal{G}$ is mapped to a valid computation in $\mathcal{C}^k(\mathcal{G})$ and every hypergraph reachable from the start hypergraph can be mapped homomorphically to (the graphical component of) $\mathcal{C}^k(\mathcal{G})$ (and its image is reachable in the Petri graph). Therefore, given a property over hypergraphs reflected by hypergraph morphisms, if it holds for all hypergraphs reachable in the covering $\mathcal{C}^k(\mathcal{G})$ then it also holds for all reachable hypergraphs in $\mathcal{G}$. Important properties of this kind are the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). These structural properties can be specified using regular expressions or by a monadic second-order logic on graphs that can be combined with a temporal logic [4].

The technique described above has been implemented as part of the AUGUR tool.[4] It takes as input a graph transformation system encoded in GTXL (Graph Transformation eXchange Language, an XML standard for graph transformation systems) and outputs the Petri graph in GXL (Graph eXchange Language). Next, several tools integrated with AUGUR can be used for verifying the desired properties over the resulting Petri graph.

[4] See `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`.

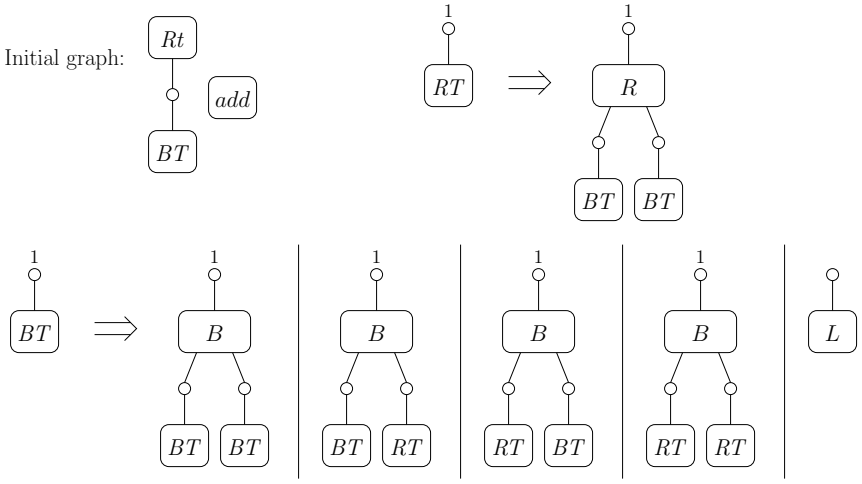

Fig. 4. A context-free grammar for generating red-black trees.

In order to show with AUGUR that insertion in a red-black tree does not violate Condition (R), we provide as input to the tool a modified version of the rules shown in Figs. 2 and 3, as well as rules for generating all possible red-black trees. The context-free rules for generating trees are shown in Fig. 4, together with the initial graph: they use the non-terminals BT and RT, and generate all finite trees satisfying Conditions (RL) and (R), but possibly not Condition (D) (i.e., they are not balanced). Moreover, the rules modeling insertion are obtained from those of the previous section as described next.

First, since every possible red-black tree is generated by the rules of Fig. 4, it is sufficient to show that Condition (R) holds again after a single insertion; thus in the modified rules, the token *add* is never generated again. Second, in order to speed-up the verification, it is convenient to "freeze" the part of the tree traversed during insertion. This is obtained by changing all labels Rt, B, R and L appearing in the right-hand side of rules to labels Rtx, Bx, Rx and Lx, respectively, which do not appear in any rule's left-hand side (see Fig. 5). This transformation is safe, because the hyperedges with x-marked labels do not interfere with the current insertion, and no further insertion is possible by the previous point.

The third modification is necessary because the current implementation of the approximated unfolding suffers from the restriction that a rule cannot have two hyperedges with the same label in the left-hand side, but rules [flip], [rotation] and [double-rotation] do not satisfy this restriction. Therefore the offending rules are converted into an equivalent set of rules which use some new labels and satisfy this restriction. The way the new rules work can be grasped from Fig. 6. If the first three rules can be applied in sequence, then we identified an occurrence of the left-hand side of [double-rotation], and therefore the corresponding right-hand side is generated (modified according to the previous two points). If instead

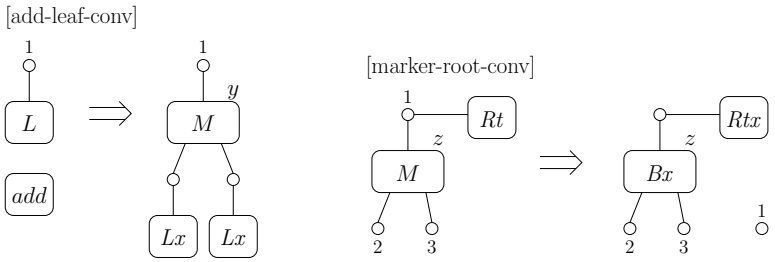

Fig. 5. Rules of the converted system, part I.

after the first two rules the left-hand side of the fourth rule is found, then we generate the right-hand side of a [flip]. It can be shown that the converted rules are equivalent to the original ones, in the sense that if G and G' are graphs containing only labels of the original graph rewriting system, then G can be rewritten to G' in the original system if and only if G can be rewritten to G' in the converted system, possibly in more steps. Furthermore, all hyperedges labeled by a label introduced in the converted system will eventually be deleted.

Applying AUGUR to the graph rewriting system just described and asking for the 0-th approximation we get a Petri graph $\mathcal{C}^0$ with 125 hyperedges, 72 vertices and 46 transitions, which is too large to be depicted here. In order to show that the property under consideration holds, we want to check that no reachable graph contains a path corresponding to the regular expression $(R+Rx)(R+Rx)$. The tools integrated into AUGUR can convert this regular expression into a set of markings such that a path of this kind exists in the approximation if and only if the corresponding markings are reachable in $\mathcal{C}^0$. However, in this case the set of markings is empty, meaning that the hypergraph underlying the Petri graph does not contain two consecutive red edges. In other words, using only the structural properties of the covering $\mathcal{C}^0$ (without taking into account its behavior) we can infer the desired property.

4.2 A Type System

In [9] a general framework for typing graph rewriting systems has been presented which will be instantiated in the following in order to analyze red-black trees. Type systems of this kind can be used to check structural invariants and are related to type systems for process calculi [12].

Some intuition. Loosely speaking, a type system for a graph rewriting system is a mapping that associates to a graph G another graph T, the (graph) type of G. We say that it satisfies the *subject reduction* property if whenever G is rewritten to G' and G has type T, then G' also has type T. In order to prove that insertion preserves Condition (D) (all leaves have the same black depth), it suffices to design a type system and a condition (P) over graph types such that:

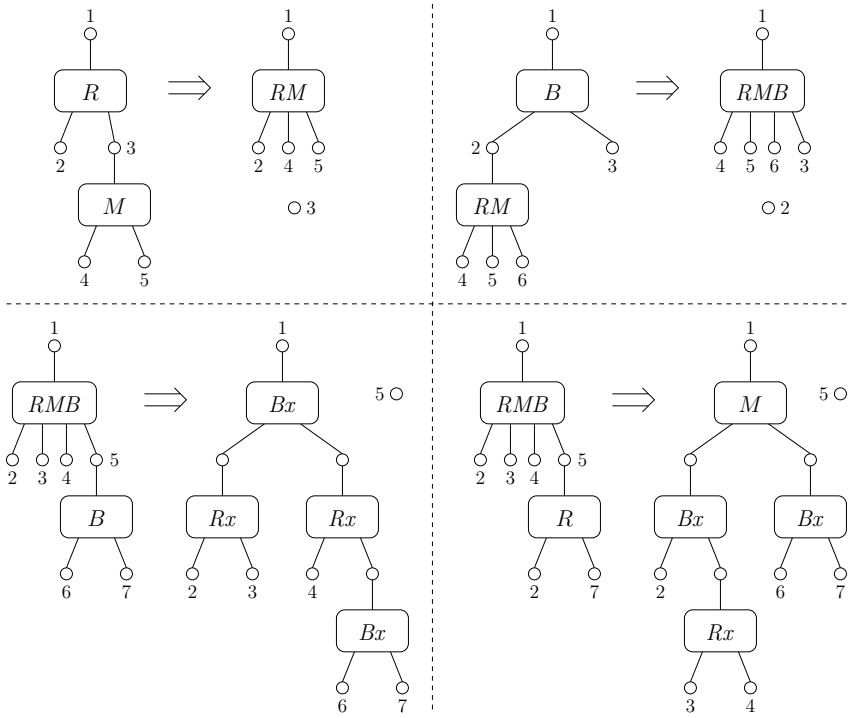

Fig. 6. Rules of the converted system, part II.

(1) the type system has the subject reduction property with respect to the rules for insertion;
(2) a graph satisfies Condition (D) if and only if its type satisfies Condition (P).

To see why, let G be any tree satisfying Condition (D), and let G' be the result of performing an insertion into G. By (1), G and G' can be assigned the same type. By (2), this type satisfies Condition (P) and, by (2) again, G' satisfies Condition (D).

Intuitively, our type system assigns to a graph G the graph T obtained by (a) removing all red hyperedges, directly linking their parents to their children, and (b) merging all black hyperedges having the same distance from the root. It is easy to see that G satisfies Condition (D) if and only if no leaf of G is merged to an inner hyperedge of T. This is Condition (P).

The technical setting. In the following we consider graphs G with a distinguished sequence of *external vertices* $\chi_G \in V_G^*$, possibly with repetition. Graphically, we identify the i-th vertex in the sequence by writing the number i close to the corresponding node. The length of χ_G is called the *arity* of G. Rewriting rules of the form (L, R, α) can now be seen as pairs of graphs with the same arity, where χ_L is an arbitrary but fixed linearisation of V_L, and $\alpha(v) = v'$ if and only

if v, v' appear in the same position of χ_L, χ_R. In the following, all operations and morphisms are expected to preserve external vertices, i.e., for a morphism $\varphi: G \to G'$ we demand $\varphi(\chi_G) = \chi_{G'}$. Technically, a type system associates to G not only the graph T, but also all the graphs T' such that there is a graph morphism $T \to T'$ (the type T can be seen as a subtype of each such T'). All these graphs are the *graph types* of G, and T is the *strongest graph type*. We consider type systems in which the strongest graph type is obtained by first applying a local transformation which replaces every hyperedge e by a graph having the same arity as e. In a second phase a global closure operator is applied which usually "folds" the graph obtained after the first step. In [9] it is shown that under some mild conditions the subject reduction property holds whenever we can show the following local property for every rewriting rule (L, R, α):

> **(Local subject reduction)** Let T_L, T_R be the strongest graph types for L and R. Then there is a morphism $\varphi: T_R \to T_L$.

The type system. We describe the local and global step of our type system. They correspond to the algorithmic steps (a) and (b) described above. We consider here a graph rewriting system modeling a single insertion into a tree, consisting basically of the rules of Fig. 2 and Fig. 3, where in the right-hand sides the token *add* is removed and labels are of the variant marked by x (see the first two modifications described in Section 4.1).

Local step: We replace every hyperedge modeling a black node by two binary edges and every leaf by a unary edge indicated by a black rectangle (see Fig. 7). Furthermore markers and red hyperedges are removed and all their vertices are collapsed (this corresponds to step (a) above). A hyperedge labeled *Rt* indicating the root is typed with a binary edge in order to have a black node "in reserve" whenever the black depth of a tree grows.[5]

Global step/Closure: In the global step we collapse all branching black paths into one (step (b) above) as follows: Whenever there are two binary edges with the same source vertex or two unary edges with the same vertex, they are merged. This process may have to be repeated.

Alternatively this closure operation can also be characterised by means of a universal property.

If we apply the process described above to the red-black tree E in Fig. 1 we obtain the graph type T_E depicted on the right-hand side of Fig. 8 (the intermediate graph obtained after the local step is shown on the left-hand side).

Proving that insertions preserve Condition (D). According to the scheme shown at the beginning of the section, we have to prove that (1) the type system has the subject reduction property and (2) find a condition (P) such that Condition (D) holds for a graph iff Condition (P) holds for its (strongest) type.

[5] Observe that the type system makes a distinction between *Rt* and *Rtx* since in the case of *Rt* an extra black edge is inserted, which is not done in the case of *Rtx*. This makes it possible to establish the subject reduction property for rule [marker-root].

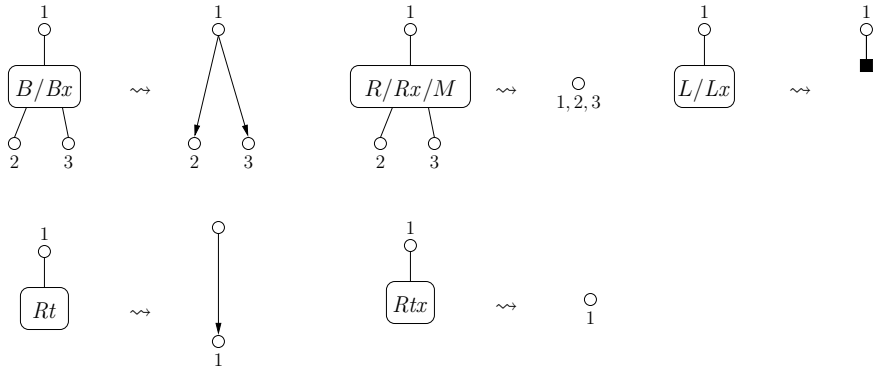

Fig. 7. Local step.

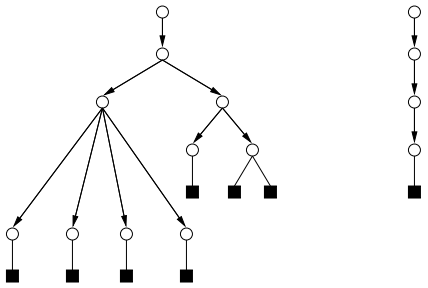

Fig. 8. Construction of a graph type (after the local/global step).

For (1), it is straightforward to show that the components of the type framework, especially the operators used in the local and global step, satisfy the conditions identified in [9], and thus it suffices to prove the local subject reduction property. This is quite easy for most of the rules, we only show the property for the rules [add-leaf], [marker-root] and [double-rotation] (see Fig. 9).

As for (2), recall that (P) should intuitively be: no leaf of a graph is merged with an inner node in the graph type. The next proposition formalizes this fact.

Proposition 1. *Let E be a tree satisfying all conditions of a red-black tree with the possible exception of Condition (D). Furthermore let T_E be its strongest type. Then all leaves in E have the same black depth if and only if T_E satisfies the following condition:*

> *(P) No unary edge (representing a leaf) is attached to a vertex which is also the source vertex of a binary edge (representing a black edge).*

Furthermore (P) satisfies the conditions specified in [9], specifically it is reflected by morphisms.

Hence we have shown that only balanced red-black trees are reachable from a balanced red-black tree by the rewriting rules in Figs. 2 and 3.

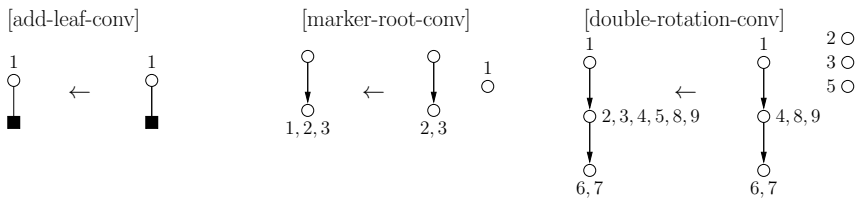

Fig. 9. Checking the local subject reduction property.

5 Conclusion and Related Work

We have shown how to model insertion into red-black trees using graph transformations and we have demonstrated how analysis and verification techniques based on graph transformation can be successfully used to verify the (partial) correctness of insertion. The first technique (approximated unfolding) is fully automatic and especially well-suited for showing that no reachable graph contains certain forbidden graph patterns. Other types of invariants can be more conveniently checked by using the second technique, a type system which is an instance of a general framework.

More generally, we are convinced that a single analysis method can not solve all problems, and thus a mix of several techniques is a promising method for the verification of pointer structures. For example, there are other relevant properties related to insertion into red-black trees that we did not address formally (as this was beyond the goal of the paper), but that we could handle using other available techniques. For example, it is quite obvious that termination of insertion can be proved easily by defining a suitable reduction ordering. More interestingly, let us sketch how the available theory of confluence for graph rewriting systems could be used to prove that insertion into a red-black tree preserves sortedness (Condition (S)).

Let us consider the system consisting of the rules of Figs. 2 and 3; since keys are relevant for this discussion, we assume that they are represented as unary hyperedges connected to the B, R or M hyperedge through a fourth node. The technique is based on the well-known fact that a binary tree is sorted (i.e., it satisfies Condition (S)) if and only if the in-order traversal returns its keys in sorted order. The in-order traversal can be modeled by graph rewriting rules that, starting from the root, destroy the tree while collecting all the keys in a linked list. Then the preservation of sortedness can be reduced to the proof that the system containing the rules for insertion and for in-order traversal is confluent: together with termination, intuitively this means that at whatever stage we stop the insertion, the in-order traversal returns the keys in the same order, and thus the tree remains sorted if it was so at the beginning. Pragmatically, confluence can be proved by resorting to a *critical pair lemma* for graph rewriting [13], and automated support for critical pair analysis is provided, e.g., by the AGG tool.[6]

[6] See `http://tfs.cs.tu-berlin.de/agg/critical_pairs.html`.

Research concerned with the verification of graph transformation systems is fairly recent. While some research groups [22, 6] pursue the idea of translating graph transformation systems into the input language of a model checker, others attempt to develop new specialized methods for graph rewriting. Work from our side goes in this latter direction, as well as [15, 14, 16]. Most of the work so far is concerned with verifying finite-state systems, whereas we have shown in this paper how to analyze an infinite-state system where elements can be inserted into red-black trees of arbitrary size.

In [23] red-black trees are checked using the structural analysis technique implemented in Alloy. Originally written in Java, the program is translated to Alloy's input language and is then analyzed (by a further translation to SAT). This requires bounds on the number of generated objects, on the maximum depth of the call stack and on the number of times a loop can be executed. In [21], the Moped model-checker is used, which allows to remove the last two bounds, but not the first. As a consequence, both techniques will find bugs if they appear for trees with a few nodes (around 5-7 in [23, 21]), but are not able to completely verify red-black trees of arbitrary size as we have done in this paper.

In [8] and [1] shape types and shapes are introduced as certain graph languages. Both papers propose algorithms that analyze each rule and check whether (and how) it may change the shape of a graph. In order to describe shapes the former uses context-free grammars whereas the latter uses more expressive graph reduction systems, that are able to express properties such as balancedness. In principle this technique could be used to show invariants of red-black trees, but the choice of graph reduction systems for shapes is non-trivial.

Furthermore insertion into red-black trees has been analyzed using the pointer assertion logic PALE and the tool MONA [7].

References

1. Adam Bakewell, Detlef Plump, and Colin Runciman. Checking the shape safety of pointer manipulations. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *Proc. of RelMiCS '03*, volume 3051 of *LNCS*, pages 48–61. Springer, 2003.
2. Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.
3. Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proc. of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer, 1999.
4. Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT'02*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001. Second Edition.
6. Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, volume 2884 of *LNCS*, pages 261–275. Springer, 2003.

7. Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. of ESOP '00*, pages 119–134. Springer-Verlag, 2000. LNCS 1782.
8. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. of POPL '97*, pages 27–39. ACM, 1997.
9. Barbara König. A general framework for types in graph rewriting. *Acta Informatica*, to appear.
10. Michael Löwe, Martin Korff, and Annika Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley, 1993.
11. Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer, 1984.
12. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
13. Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–214. John Wiley, 1993.
14. Arend Rensink. Model checking graph grammars. In *Proc. of AVOCS '03 (Workshop on Automated Verification of Critical Systems)*, 2003.
15. Arend Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, volume 2986 of *LNCS*, pages 401–415. Springer, 2004.
16. Arend Rensink. State space abstraction using shape graphs. In *Proc. of AVIS '04*, ENTCS, 2004. to appear.
17. Leila Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
18. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.
19. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.
20. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. of POPL '96*. ACM, 1996.
21. Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jMoped: A Java Bytecode Checker Based on Moped. In *Proc. of TACAS 2005*, volume 3440 of *LNCS*, pages 541–545. Springer, 2005.
22. Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Proc. of GT-VMT'02*, volume 72 of *ENTCS*. Elsevier, 2002.
23. Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*, pages 505–520. Springer, 2003.
24. Robert Paul Wilson. *Efficient, Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, 1997.