Permission-Based Verification of Red-Black Trees and Their Merging

Lukas Armborst   Marieke Huisman
Formal Methods and Tools
University of Twente
Enschede, The Netherlands
Email: {l.armborst, m.huisman}@utwente.nl

***Abstract*—This paper presents a verification case study, focussing on red-black trees. In particular, we verify a parallel algorithm for merging red-black trees, which uses lists as intermediate representations and which an industrial partner uses to efficiently manage tables of IP addresses. To verify the algorithm, we use the tool VERCORS, which uses permission-based separation logic as its logical foundation. Thus, we first needed a suitable specification of the data structure, using that logic. This specification relies on the magic wand operator (a.k.a. separating implication), which is a connective often neglected when discussing separation logic. This paper describes that specification, as well as the verification of the parallel algorithm. It is an interesting case connecting the more academic endeavour of verifying a data structure with the practical one of verifying industrial code.**

***Index Terms*—software verification, formal methods, trees**

I. Introduction

Deductive verification techniques and tools have matured over the last years, making them more applicable for industrial use (e.g. [1]). Nevertheless, it still requires considerable effort to verify a program, and it is still ongoing work to verify even quite simple or academic examples, such as basic data structures (e.g. [2]). In this paper, we combine the industrial and the academic, based on the desire of our industrial partner NLnet Labs to verify a parallel merge algorithm for red-black trees. We therefore first do the more academic task of formally verifying the data structure of red-black trees. Due to the concurrent nature of the merging algorithm, we need to verify the data structure already in an appropriate logic, even if it does not use concurrency itself. Hence, we provide a formalisation of the tree structure in permission-based separation logic. We also define the appropriate pre- and post-conditions for operations like *insert* and *delete* that allow a deductive verifier (in our case the VERCORS tool) to formally prove that the implementation correctly adheres to those specifications. The verification of *delete* is particularly interesting, as it uses the *magic wand* (a.k.a. *separating implication*) operator, which is still not supported by all tools, despite clearly being a useful connective. Afterwards, we use the (now proven to be correct) data structure to verify the parallel merge algorithm from NLnet Labs. This algorithm uses lists as intermediate representations during the merging, and uses batch processing together with a producer-consumer pattern to organise the concurrent access to the lists. The verification of the latter can again be of academic interest and is applicable in other use cases, too.

A. Contribution

In this paper, we present a case study, which

- highlights the applicability of deductive verification and associated tools to industrial use cases;
- presents knowledge and insights that can be reused in future case studies and that can guide future research and tool development;
- provides a verified Java implementation of red-black trees and the merging algorithm by NLnet Labs;
- showcases the usefulness of the magic wand operator, potentially incentivising the maintainers of other tools to support it in the future.

B. Outline

The remaining paper is structured as follows: the next section explains the background, such as permission-based separation logic, the VERCORS tool and red-black trees. Afterwards, Section III details how the red-black trees are formalised in VERCORS. Then, Section IV explains the parallel merging algorithm and how it was verified in VERCORS. Section V discusses the findings of this case study as well as related work, before Section VI concludes the paper, also providing an outlook on future work. While the following sections include code snippets, you can find the full code base at [3].

II. Background

We use the VERCORS tool to verify the red-black trees and their merging. To better understand how we do that, this section provides some background knowledge: first we give a brief introduction into the logic underlying VERCORS, namely permission-based separation logic, and in particular the *magic wand* connector (Section II-B). Then we explain the VERCORS tool (Section II-C), and finally the data structure of red-black trees (Section II-D).

A. Permission-based separation logic

Permission-based separation logic is an extension of Hoare logic [4]. In Hoare logic, the behaviour of a statement (or combination of statements) is described via *pre-conditions* and

post-conditions: the pre-condition defines the state that the program needs to be in, in order to execute the given statement correctly; the post-condition characterises the state that the program is sure to be in after the statement is executed. The program state is described via a formula in first-order logic, characterising the values of the program variables. Variables can be local *stack variables* with a restricted scope (e.g. arguments of a method), or global *heap variables*. To manage access to shared memory, *separation logic* [5] extends Hoare logic to describe heap values and how they are accessed. *Permission-based separation logic* [6] extends this for multi-threaded systems by allowing multiple threads to access the same heap location in a safe way, meaning they can only access the same heap location simultaneously if none of them writes to it. This is coordinated by explicitly managing access permissions to heap locations: the logic is extended to contain permission predicates, and a formula can only refer to a heap value if it also contains permission for that location. This is called *self-framing*. We use the style of Implicit Dynamic Frames [7] for using access permissions. In particular, we have a predicate $\mathsf{Perm}(x, p)$, which allows access to the location of the heap variable x. The value p is a fraction from the interval $(0, 1]$, where 1 represents write access, while any value between 0 and 1 only allows read access. At any time, the sum of all permissions for x from all threads must not exceed 1, meaning either one thread has write access, or multiple threads can have read access.

We use the *separating conjunction* $A * B$ from separation logic to combine access permissions: $\mathsf{Perm}(x, 1) * \mathsf{Perm}(y, 1)$ means that we have access to both x and y, and they are in disjoint parts of the heap, i.e. they cannot be aliases for the same location. For simplicity, we also use that symbol to connect access permissions to the logical part of the formula, and between logical formula elements. In that case, it has the meaning (and precedence) of a logical *and*: $\mathsf{Perm}(x, 1) * \mathsf{Perm}(y, 1) * x = y * x > 0$.

We can group access permissions into *resource predicates* (or *predicates* for short), for instance combining access to all fields of an object into a single predicate: $\mathsf{resource}\ \mathtt{my_pred}(o) = \mathsf{Perm}(o.\mathit{field}_1, 1) * \mathsf{Perm}(o.\mathit{field}_2, 1)$. This grouping improves readability and facilitates modularity. Also, the bodies of predicates can refer to other predicates, and be recursive. The latter allows us to define access permissions for entire recursive data structures like the trees in this paper (see Section III). We use the notion of *iso-recursive* predicate (for more information on the notion of iso- and equi-recursiveness, see [8]). This means that having a predicate like `my_pred` is not automatically equal to having the contained access permissions; instead, the user has to explicitly *unfold* the predicate to replace an instance of the predicate with the respective predicate's body (thereby making the corresponding locations accessible). Inversely, *folding* a predicate requires that all the access permissions of the predicate's body are currently held (e.g. write permissions for $o.\mathit{field}_1$ and $o.\mathit{field}_2$), and removes them from the current context, replacing them with an instance of the predicate (i.e. $\mathtt{my_pred}(o)$). While this slightly increases the effort for the user, it gives more control, guiding a verifier to the proper unrolling of recursive predicates. Besides access permissions, a predicate's body can contain logical formulae. These formulae must be true in the current context *before* the predicate is folded, and conversely they are assumed to hold after it is unfolded (without proving that they really do). The predicate must be self-framed, i.e. it must contain access permissions for all locations that the logical formulae refer to.

B. Magic wands

While the separating conjunction allows us to split the heap into disjoint parts and reason about them independently, the *magic wand* does the converse, allowing us to merge disjoint parts of the heap and reason about them as a whole. Reynolds called this binary connective *"separating implication"* in his initial paper on separation logic [5]. But nowadays, it is more often referred to as the "magic wand" (e.g. [9], [10]), or "wand" for short, so we will also use those terms. A wand $A \mathrel{-\!\!*} B$ encodes the possibility of transforming the left-hand side, A, into the right-hand side, B. But it does not contain A itself, it only stores all the permissions and assertions that are necessary to exchange a *given* A for a B. Note that this transformation consumes both the given left-hand side and the wand, leaving only the right-hand side, i.e. $A * (A \mathrel{-\!\!*} B)$ only entails B. This is similar to the *linear implication* in linear logic (see e.g. [11]). However, it is different from the boolean implication, where $p \rightarrow q$ can transform a given p into a q, but retains the original parts: $p \wedge (p \rightarrow q)$ entails $p \wedge (p \rightarrow q) \wedge q$.

As an example, A might represent the permissions for a partial list and B the permissions for the full list. The wand $A \mathrel{-\!\!*} B$ contains the permissions for the part of the list that is not in A, and also the knowledge how to combine the parts (e.g. that A is the tail of the list). Intuitively, the wand represents a predicate with a "hole" cut into it (*"B, but with A cut out"*). It allows us for instance to iterate over recursive data structures with recursive predicates: while the part that still needs iterating is usually a valid data structure due to the recursive nature, the part that we already iterated is not a valid data structure by itself (e.g. not a list ending in null). Therefore, defining the permissions for that part can be tricky. With the magic wand, separation logic provides an elegant solution for that.

Combining the wand $A \mathrel{-\!\!*} B$ with the pre-condition A to obtain B is called *applying* the wand. We *create* a wand by bundling the necessary permissions (e.g. the permissions for the remainder of the list) and replacing them with the wand, similar to folding a predicate. Thus, we can only create a wand if we hold the corresponding permissions and can prove in the current context that the necessary facts are true (e.g. the fact that the missing part is the tail, and not the front). Typically, you would start with a B and split it into an A and the corresponding wand, in order to work on them separately. After the separate work is done, you recombine the parts by applying the wand.

In a wand $A \mathrel{-\!\!*} B$, the two parts A and B can be more complex than just predicates, for example asserting additional information about the length: if the provided list has length k, then the joint list has length $k+n$ (semi-formal: $(perm(l_1) * len(l_1) = k) \mathrel{-\!\!*} (perm(l_2) * len(l_2) = k+n)$). Again, the necessary information (in this case the fact that the list portion stored in the wand has length n) has to be available in the current context when creating the wand. Note that both sides of the wand have to be self-framing expressions, so the right-hand side cannot contain for example $len(l_1) + n$, since the access permissions to l_1 are no longer (directly) available at this point, but are integrated into l_2.

Tool Support Even though the magic wand is an intrinsic part of the logic and a useful operator (as this case study shows), many verification tools do not support it. Blom et al. [9] provide a detailed analysis which tools support wands, or can simulate the functionality of them. Even though their work is several years old, not much has changed: jStar, SmallFoot and Chalice are no longer maintained, and therefore still lack support. Development of Verifast [12] still continues, but does not include wands. The Viper tool-suite [10] does support wands, as does VERCORS [13].

C. VERCORS

VERCORS [13] is an automatic verifier based on permission-based separation logic. It requires the user to provide annotations inside the code, and verifies that the program adheres to the specifications defined by those annotations. VERCORS can verify programs written (and annotated) in its own language PVL, as well as more common languages like Java, with the latter being what we use here. In the case of Java, annotations are provided as JML-style comments [14], such as `//@ fold my_pred(o)`. VERCORS parses those annotations along with the code, and translates the annotated program into the Silver language of the back-end verifier Viper [10]. It then invokes Viper on that Silver program, which in turn uses the Z3 solver [15] to reason about the code.

Some keywords of VERCORS, which are relevant for the code snippets below, are: a *method contract* is an annotation right above a method header, specifying the method's behaviour in terms of *pre-conditions* and *post-conditions*. The former are specified using the keyword requires, the latter using ensures. A post-condition can refer to the return value of the method via \result, and to values before the method's execution via \old (e.g. say ensures $x = \backslash$old(x) to specify that the method does not change the value of x). *Ghost code* are annotations that look similar to executable code, e.g. variable declarations and updates. This can be helpful to verify the program, for example to store intermediate values. In most cases, ghost code uses the keyword ghost. A particular type of ghost code are *ghost results*, which are additional return values of a method besides the "real" return value. They are defined in the method contract using yields. To use them, a method call is followed by then and a block of assignments $x = y$ that store the ghost result y in a local ghost variable x. Ghost code must not have side effects on the executable code, for instance it cannot store a ghost return value into a "real" variable.

D. Red-black trees

Red-black trees (following [16]) are a special type of binary search trees, whose additional constraints ensure a notion of balance, preventing the tree from degenerating into a linear structure and thus ensuring that the lookup time remains logarithmic. Like any binary search tree, a red-black tree consists of nodes that contain a key, according to which the tree is sorted, and up to two child nodes, referred to as *left* and *right*. These child nodes can have children themselves, thereby recursively spanning sub-trees that are again red-black trees. Nodes in a binary search tree are sorted such that all keys occurring in the left sub-tree are less than the key of the root node, while keys occurring in the right sub-tree are greater or equal. In addition to those properties, nodes in a red-black tree each have a colour that can be either red or black (see Figure 1). The maximal number of black nodes encountered on any path from the root to a leaf node is called the *black height* of that tree. For instance, in Figure 1a, the black height of the root node is 1, as each path from the root 5 to any leaf only has one black node. In contrast, Figure 1h has black height 2.

A valid red-black tree has to satisfy three important properties:

1) The left sub-tree and the right sub-tree are themselves valid red-black trees.
2) The two sub-trees have the same black height (the tree is *black balanced*).
3) The children of a red node are black.

Together, these properties ensure that the longest path from the root to a leaf is at most twice as long as the shortest path (alternating red and black nodes vs. having only black nodes, as in node 16 vs. node 3 in Figure 1h). This means that the tree is roughly balanced, and looking up a key takes logarithmic time (in the size of the tree).

We now describe the tree operations of *insert* and *delete* on a very abstract level. For more detail, see for example [17, Chapter 13]. Section III-B describes the annotations necessary to verify those operations. The operations need to maintain the properties listed above. This is accomplished by first inserting/removing the node, potentially causing a temporary violation of some properties, and then re-establishing the properties with a series of localised corrections. In particular, this can mean changing the colour of a node (e.g. Figure 1b to 1c), or rotating the tree (Figure 1f to 1g is a rotation to the left of the right sub-tree of 10, doing the reverse is a rotation to the right).

a) Insertion: A new node is initially inserted as a red leaf (see Figure 1a), thus maintaining the second property. However, if its parent is itself red (like node 18 in the example), this violates the third property; this is called a *double red*. Depending on the colour of the sibling of the new node, a specific series of re-colouring and rotation operations are performed, either resolving the double red or propagating it upwards (as in Figure 1a to 1b), while maintaining the

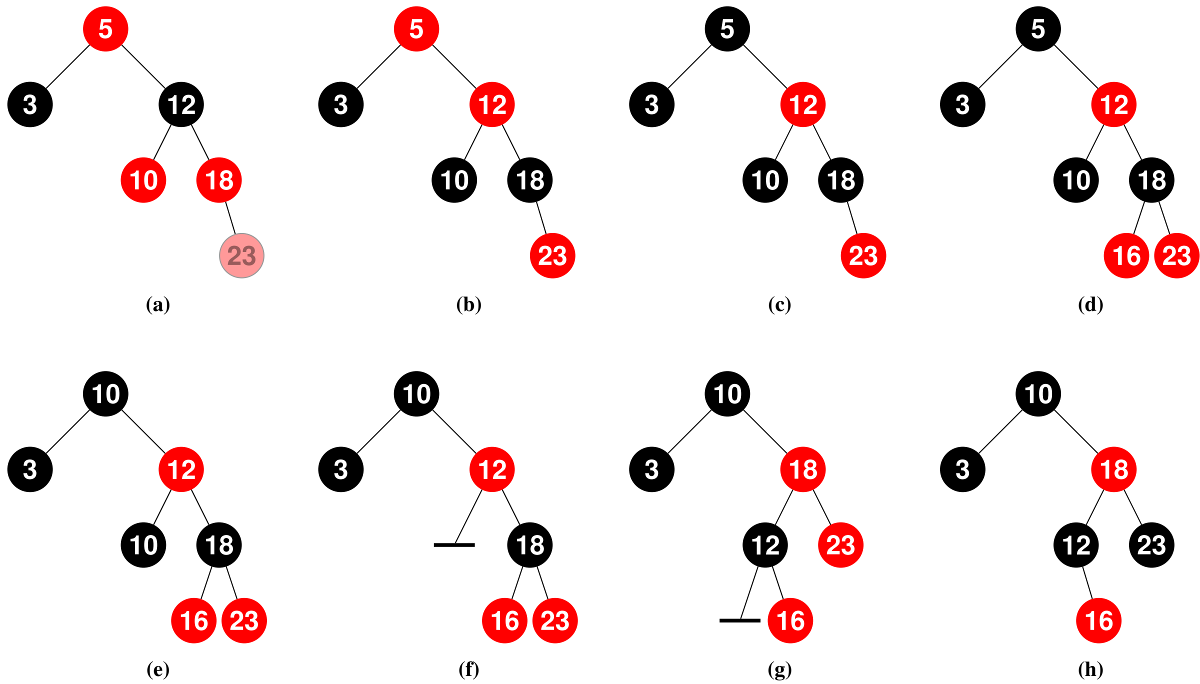


Figure 1: (1a) 23 is added as a new node into a previously valid red-black tree, creating a double red at 18-23. (1b) Changing colour on 10, 12 and 18 propagates the double red upwards to 5-12. (1c) The issue is resolved by colouring the root black; red-black tree is valid. (1d) Adding 16 does not require any fixing afterwards. (1e) To delete the internal node 5 (the root), the data from the successor node 10 is copied into that (root) node. (1f) The successor node 10 can be deleted, but leaves a black marker behind. (1g) Rotating the sub-tree at 12 to the left makes 18 the new root of that sub-tree. (1h) Re-assigning the black marker to 23 results in a valid red-black tree.

first two properties. If the double red reaches the root node, we can change the colour of the root to black (Figure 1b to 1c), thereby re-establishing Property 3 without upsetting the balance of black nodes.

b) Deletion: To delete an internal node with two children, we use a helper method *getMin* to find the successor node, which is the smallest (i.e. left-most) node in the right sub-tree, and copy its data into the current node (Figure 1d to 1e). Afterwards, we delete that successor node. This reduces the problem of deleting an internal node to the one of deleting a node with at most one child, which is straightforward structurally. Unfortunately, if the deleted node was black, the deletion breaks Property 2. To alleviate that, the black marker of the deleted node is kept in place (Figure 1f, Figures 2a and 2d). Again, depending on the colour of the immediate surrounding, a specific sequence of re-colouring and rotation operations is performed if necessary, basically "moving around" the extraneous marker. If it is assigned to a red node, that node turns black and the problem is resolved (Figure 1g to 1h, Figure 2d); assigning it to a black node temporarily makes that node *double black* (Figure 2b). As with the double red, the problem is either resolved locally, or propagated upwards. And again, if it reaches the root, the extraneous marker can be discarded without upsetting the balance (Figure 2c), resulting in a valid red-black tree.

Implementing *delete* in that way leads to a fourth property for valid red-black trees, that the implementation has to ensure:

4) There are no extra black markers or double black nodes in the tree.

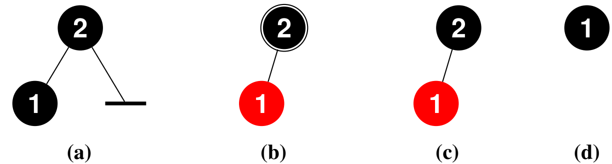


Figure 2: (2a) Deleting a black node left a black marker behind. (2b) The marker is propagated upwards, resulting in a double black root. (2c) At the root, any extra markers can be discarded; red-black tree is valid again. (2d) Deleting a node with one child (which has to be red due to black balancing) assigns the marker to the child, turning it black.

Note that the first three properties are more intrinsic to the data structure, while this fourth one is a remnant of the deletion algorithm (and while other versions of *delete* might not need it, it is commonly used). Nevertheless, for us they are all equally relevant.

III. Formalisation of Red-Black Trees

This section describes how the red-black tree structure is formalised (Section III-A) and how the operations *insert* and *delete* are verified (Section III-B). For the latter, a magic wand is used. While we have to skip many details here for brevity, the full code can be found at [3].

Note that none of these operations use concurrency, and thus would be far easier to verify in a sequential logic that does not require managing permissions. However, since we want to

```
1:  int key;
2:  Node left, right;
3:  boolean colour, dblack, dblackNull;
4:  /*@ resource tree_perm(Node node) = node != null
5:          ⇒ node_perm(node)
6:            * tree_perm(node.left)
7:            * tree_perm(node.right)
8:            * node.dblack ⇒ !node.colour;

9:       requires tree_perm(node);
10:     boolean noDoubleRed(Node node) = node != null
11:         ⇒ (!colour(node)
12:             || (!colour(node.left)
13:                 && !colour(node.right)))
14:           && noDoubleRed(node.left)
15:           && noDoubleRed(node.right); @*/
```

Figure 3: The fields of a Node, the tree_perm access predicate, and an example of a property of a valid red-black tree. For readability, the folding and unfolding commands for predicates were omitted.

use concurrency in the merging of Section IV, we need to use a logic supporting concurrency to formalise the trees, and thus also verify the operations below in the respective logic.

A. Tree structure

A tree consists of Nodes. Each Node contains an (integer) *key* for the sorting, two Node references *left* and *right*, and a boolean *colour*, where true means red and false means black. Additionally, it has two boolean fields *dblack* and *dblackNull*, one to indicate whether the node is double black (e.g. Node 2 in Figure 2b), and the other to indicate that the node was deleted, but left a black marker behind (e.g. in Figure 1f). Apart from the *key*, a node may contain additional data, which we omit here. We will use the abbreviation node_perm(*node*) to concisely refer to access permissions to all these fields combined.

We define a recursive predicate tree_perm to store access rights to the entire tree (see Figure 3, Lines 4-8): if the given node is null, there are no permissions to have. Otherwise, the predicate contains the permissions for the fields of *node* (expressed by node_perm(*node*)), and recursively the tree_perm predicate for the two sub-trees. Additionally, we add some sanity checks to the predicate, for instance that a double black node must actually be black (Line 8). Encoding such invariant properties in the predicate simplifies the verification.

With this tree structure, we can now encode the properties of a red-black tree as boolean functions, for example Property 3 as noDoubleRed (see Figure 3, Lines 9-15). The definitions for all properties can be found in the appendix, as well as the source code online [3]. We group the properties (along with the sortedness, which we omit here for brevity as it is a standard property commonly found in tree formalisations) together into a boolean function valid.

B. Tree operations

The tree operations *insert* and *delete* are implemented recursively. As mentioned in Section II-D, they are ultimately performed on leaves (or nodes with just one child), and can cause a violation of red-black tree properties. These are then repaired locally, propagating the problem potentially up to the root, where it can be resolved easily. Therefore, these methods consist of two parts: the public method insert is a wrapper that calls the recursive method insertRec, which performs the actual insertion and local corrections. After insertRec returns, insert performs any action on the root node that is necessary to resolve a potential double red (e.g. turning the root black). Likewise, delete is a wrapper for the recursive deleteRec, and performs additional actions on the root node (e.g. discarding extra markers, see Figure 2). insertRec and deleteRec both use rotate helper methods to rotate a (sub-)tree, and deleteRec uses getMin to find a successor node.

a) Insert: The recursive insertRec method (see Figure 4, Lines 1-12) requires access to the current (sub-)tree via tree_perm and that it is a valid red-black tree. It ensures tree_perm (i.e. returns the permissions), but only parts of valid: the properties sorted, noDBlack and blackBalanced hold, while noDoubleRed may be violated. However, it can only be violated in a specific way (expressed by dbRedAtTop, whose implementation we omit for brevity): the root of the sub-tree is red and *one* of its children is, too (it cannot be the root and *both* children), but there must not be any violation within the two sub-trees spanned by the children. For example in Figure 1b, the root 5 and its right child 12 are red, but the left child 1 must be black and there must be no instances of double red within either the left subtree (which is only node 3) or the right subtree (with root 12).

Additionally, we prove two post-conditions to facilitate the verification of insertRec on higher tree levels: first, the black height of the tree remains unchanged (Line 7 of Figure 4), meaning the parent node and higher levels remain black balanced. Second, the colour of the root node of the sub-tree either did not change, or the root changed from black to red. In the latter case, it cannot make use of the exceptionally allowed double red (i.e. the children then have to be black, Line 9ff). To understand why, consider Node 12 in Figure 1b: it was black before, so the parent could be red (and in fact, it is). If we allowed 12 to turn red and have a red child (via the exception allowed in dbRedAtTop), then we would have a triple red (5, 12, and the child of 12), which the local corrections would not be able to deal with. Luckily, our implementation never turns a node with red children red, and providing VerCors with that knowledge allows the verification of the method.

Inside the body of the recursive method, no annotations are required, except folding and unfolding the tree_perm predicate where needed. However, the method does make use of helper methods to rotate the tree, which are described below. We omit the public method insert, the entry point to the

```
1:  /*@ requires tree_perm(node) * valid(node);
2:      ensures \result!= null * tree_perm(\result);
3:      ensures sorted(\result) * noDBlack(\result)
4:          * blackBalanced(\result);
5:      ensures noDoubleRed(\result)
6:          || dbRedAtTop(\result);
7:      ensures \old(height(node)) = height(\result);
8:      ensures (\old(colour(node)) = colour(\result))
9:          ||  (colour(\result)
10:              && !colour(\result.left)
11:              && !colour(\result.right)); @*/
12: Node insertRec(Node node, int key);

13: /*@ requires node!= null * tree_perm(node)
14:         * valid(node);
15:     ensures tree_perm(\result)* sorted(\result);
16:     ensures noDoubleRed(\result);
17:     ensures blackBalanced(\result)
18:         * (noDBlack(\result)
19:           || dblackAtTop(\result));
20:     ensures \old(height(node)) = height(\result);
21:     ensures !\old(colour(node))
22:         ⇒ !colour(\result); @*/
23: Node deleteRec(Node node, int key);

24: /*@ yields boolean resColour;
25:     yields int resHeight;
26:     yields bag⟨int⟩ resBag;
27:     requires tree_perm(node)* valid(node);
28:     ensures tree_perm(\result)* valid(\result);
29:     ensures (tree_perm(\result)* valid(\result)
30:             * subtreeFitsHole(\result,
31:                 resColour,resHeight,resBag))
32:         −∗ (tree_perm(node)* valid(node)
33:             * subtreeFitsHole(node,
34:                 \old(colour(node)),
35:                 \old(height(node)),
36:                 \old(toBag(node))));
37:     ensures resHeight = height(\result);
38:     ensures resColour = colour(\result);
39:     ensures resBag = toBag(\result); @*/
40: Node getMin(Node node){
41:     Node res;
42:     if (node = null || node.left = null) {
43:         res = node;
44:         /*@ ghost resColour = colour(node);
45:              ghost resHeight = height(node);
46:              ghost resBag = toBag(node);
47:             create {...} @*/
48:     } else {
49:         res = getMin(node.left)
50:           /*@ then {resColour=resColour;
51:                      resHeight=resHeight;
52:                      resBag=resBag;} @*/;
53:         //@ create {...}
54:     }
55:     return res;
56: }
```

Figure 4: Specifications for *insert*, *delete* and *getMin*. For readability, the folding and unfolding commands for predicates were omitted.

procedure, because it is simple and its verification straight-forward.

b) getMin: As described in Section II-D, to delete an internal node `node`, we find the successor node `succ` (the smallest node in the right sub-tree) via *getMin* and copy its data into `node`, and then call `deleteRec` to remove `succ`. For the copying, we need to have simultaneous access to both `node` and `succ`, meaning the permissions of the successor have to be temporarily extracted from the `tree_perm` of the sub-tree at `node.right`. To guarantee that we can merge those permissions back into the overall tree later, we use a magic wand.

The recursive method `getMin` (see Figure 4, Lines 24-56) descends the left tree until reaching the left-most node, and returns that node. It requires the `tree_perm` access predicate and the knowledge that the given (sub-)tree is valid. It ensures the same for the returned node and its sub-tree (Line 27f). Note that this means that the caller has full control of that sub-tree, and could for instance remove a black node from it. This would change the black height, and integrating the adapted tree back into the remainder of the original tree would disturb the overall black balance. We have to prevent that, to avoid breaking the red-black properties. Therefore, in addition to the successor node, `getMin` returns its colour, black height and the set of keys in that sub-tree as ghost return values *`resColour`*, *`resHeight`* and *`resBag`*, respectively (see Lines 24ff and 37ff of Figure 4).

In order to integrate the successor node `succ` (and its sub-tree) back into the original tree at `node`, we require that these values have not changed. This means any tree that should fill the hole in the original tree has to have the black height *`resHeight`*, contain the keys stored in *`resBag`* and its root must have the colour *`resColour`*. Note that these restrictions are stricter than necessary, for example the keys technically only have to be in the right interval to ensure that the tree remains sorted, and need not necessarily be the exact same keys as the initial sub-tree. However, in our use case, we only read data from the sub-tree and do not change it, so the simpler restriction of exactly matching the old values is used, rather than deducing the proper interval of keys. The property of a node having the right colour, black height and keys is encoded in the function `subtreeFitsHole(node, colour, height, keys)` (the implementation of which is straight-forward and omitted here).

The magic wand on Line 29-36 of Figure 4 uses the constraints of `subtreeFitsHole` and specifies the merging of the successor node back into the main tree: if the user provides the `tree_perm` access predicate for the sub-tree spanned by the successor node, asserts that this sub-tree is a valid red-black tree, and that it fits into the hole in the outer tree, then the wand provides the `tree_perm` predicate for the outer tree, and guarantees that it is valid and corresponds to the original outer tree. The latter is necessary due to the recursive nature of `getMin`: each call guarantees to the level above that the respective part of the tree still fits. In the remainder of the text, we may refer to the com-

bination of `tree_perm(node)` with `valid(node)` and `subtreeFitsHole` as `cond(node)` to help readability.

When creating the wand, there are two possibilities: if the given root *node* has no left child, then *node* is itself the smallest node and will be returned (Figure 4, Line 43-47). In that case, creating the wand (Line 47) is trivial, as \result and *node* are equal. In the recursive case of *node* having a left child, creating the wand is more difficult: the recursive call `getMin(node.left)` (Line 49) will ensure a wand as described above, but for *node.left* (i.e. `cond(res) −∗ cond(node.left)`). Since the transformation of `cond(res)` into `cond(node)` is not so simple now, we have to provide a proof script that describes how to do the transformation (Line 53). While we omit the script and all its details for brevity, the general idea consists of two steps: first, we apply the lower-level wand to exchange the permissions for *res* into those for *node.left*. Then, having a proper sub-tree at *node.left* again, we can easily recombine it with the right sub-tree and the permissions for the *node* itself into a proper `tree_perm(node)`.

For example in Figure 1d, `getMin(12)` calls `getMin(10)`. This is then the trivial case (Line 43-47), and ensures the wand `cond(10)−∗cond(10)`. Afterwards, `getMin(12)` needs to guarantee `cond(10)−∗cond(12)`. This is done by first using the lower-level wand to exchange `cond(10)` for `cond(10)` (admittedly not doing much), and then combining it with knowledge and permissions for the sub-tree at 18 and the node 12 itself to create `cond(12)`. Here, 12 is the right child of 5; if it were the left child, then `getMin(5)` would guarantee `cond(10)−∗cond(5)` by taking the wand `cond(10)−∗cond(12)` guaranteed by the lower-level call `getMin(12)`, applying it, and using knowledge and permissions of 5 and its other child to build `cond(5)`.

c) Delete: As mentioned above, the main functionality is a recursive function `deleteRec` (for its specification, see Figure 4, Lines 13-23). In the method's body (not shown), we differentiate two cases: if the node to be deleted has at most one child, then the executable code is a bit intricate to take care of potential double black scenarios, but the additional effort for verification is minimal, only requiring folding and unfolding the `tree_perm` predicate. However, if the node has two children (e.g. node 5 in Figure 1d), we have to use `getMin` on the right child (here: node 12) to find the successor (node 10), and copy its data over into the node that shall be deleted (in our case, that data is just the key). Afterwards, we have to merge the permissions for the successor sub-tree back into the original tree by applying the wand that `getMin` returned (`cond(10)−∗cond(12)`). Then, we call `deleteRec` to remove the successor node. We used ghost variables to store the colour, black height and set of keys of the right sub-tree before calling `getMin`, in order to know what values to expect from the wand's `subtreeFitsHole`, and to apply the wand correctly.

After deleting the node, the higher levels of the tree may have to perform recovery actions to remove an extra black marker in their sub-trees (see Figures 1e-1h). Note that the recursive call to `deleteRec` ensured that if there is a double black at all, it is on the root node of the sub-tree (Figure 4, Lines 18f). In the example, node 12 calls `deleteRec` on node 10, after which the extra black marker is in the place of that node (Figure 1f). We use a helper method `fixDBlackLeft`, or the symmetric `fixDBlackRight`, to fix the double black on the respective child (potentially propagating the marker upwards). Afterwards (Figure 1h), the extra marker would only be allowed on node 18 (the new root of this sub-tree), but in this case the issue was resolved entirely. Again, the executable code of these helper methods is somewhat intricate, performing rotations and colour changes, while the annotation effort is merely folding and unfolding the necessary `tree_perm` predicates. However, in the `deleteRec` method itself, we needed a bit more ghost code around the recursive call to ensure the sortedness of the resulting tree. In particular, we needed to explicitly assert that the set of keys after deletion is a subset of the keys before the deletion. So in summary, the annotations required to verify `deleteRec` are folding and unfolding the necessary predicates, caching some values in ghost variables before `getMin`, applying the wand after `getMin`, and asserting the subset relation after the recursive `deleteRec`.

d) Rotate: Rotating a tree (e.g. Figure 1f to 1g or vice versa) is simple in terms of the executable code, but more difficult in terms of verification. Again, we need to explicitly assert some subset relation, in order to ensure sortedness. However, the main difficulty is that the method is called on non-valid trees: either by `insert` when there is a double red, or by `delete` (indirectly, via `fixDBlackLeft/Right`) with a double black and a potential imbalance of black nodes. This requires a careful analysis of the property violations when calling `rotate`, for example which nodes exactly have the double black, and how this is transformed by the rotation, i.e. which node has the double black afterwards. It also requires several case distinctions, such as different places for the potential double red. An excerpt of the specification can be found in the annex, for more details please refer to the full code at [3]. Additionally, the order of operations in the calling context has a big impact: do you first rotate the balanced tree, thus create an imbalanced tree, and then move the double black marker to restore balance (as shown in Figure 1); or do you move the marker first, thus creating an imbalanced tree, and rotate it to restore balance? Initially, we used the former variant as depicted in the figure, but we found the latter case easier for defining `rotate`'s pre- and post-conditions. However, we had full control over the source code and could change this order in the executable code. When verifying externally provided code, or trying to automatically infer the specification from the code, you may not have the possibility to change to code for an easier verification, making the specifications more complex.

IV. Parallel Merging Algorithm

We use the formalisation of red-black trees described above to verify a parallel algorithm working on red-black trees. This

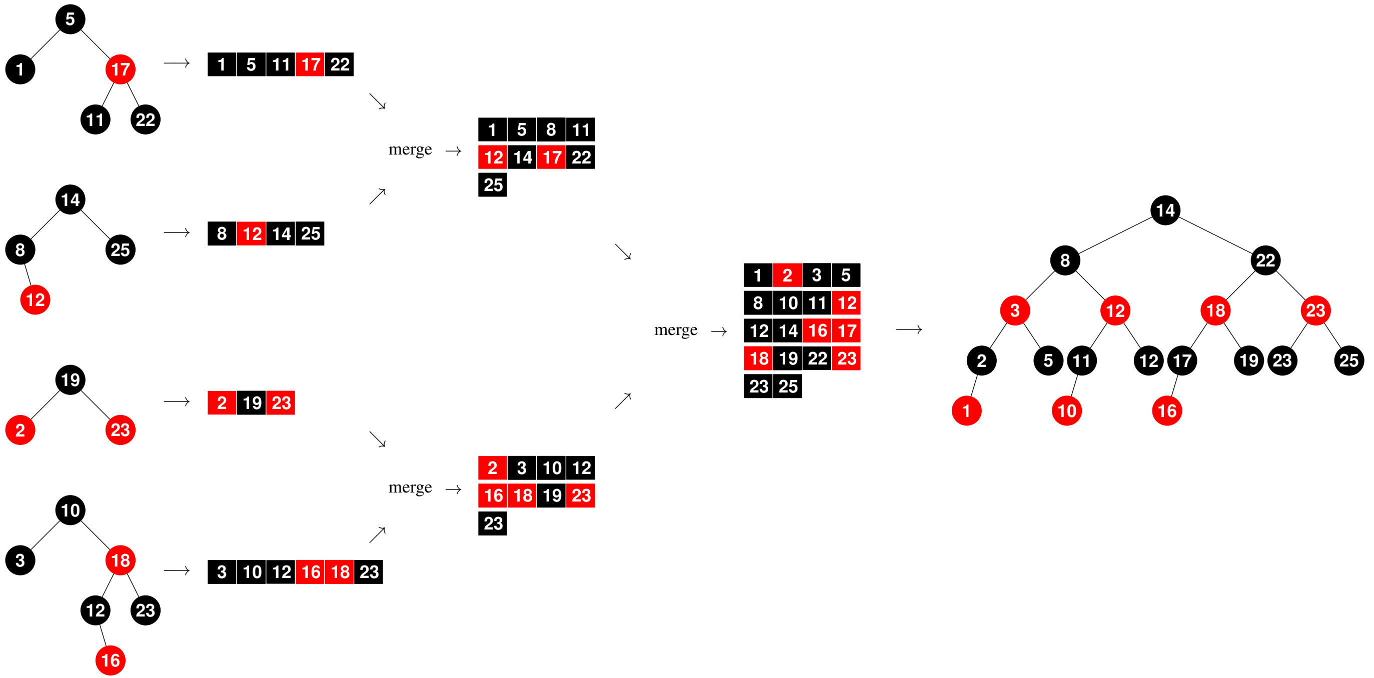


Figure 5: Scheme for merging red-black trees, using a chunk size of 4 for the lists between merger threads

justifies the overhead of using permission-based separation logic compared to a simpler sequential logic. The algorithm takes an array of red-black trees, and merges them into a single tree. The general concept of the algorithm is taken from the industrial application by NLnet Labs [18] that inspired this case study. They parallelise the loading of a large file of IP addresses by having multiple threads loading parts of the file concurrently into one red-black tree per thread. Afterwards, these separate red-black trees need to be merged into one tree, representing the entire file. The algorithm does not merge the trees directly; instead, it uses list representations of the given trees and then concurrently merges these lists into larger and larger lists (so while it reuses the `Nodes` from the trees, it is in essence a list-merging algorithm, not a tree-merging algorithm). Finally one single list remains, which contains all nodes from the given trees, and which is then transformed back into a valid red-black tree. Figure 5 depicts this concept, merging the tree from 1h with three other trees. In Figure 5, each "merge" represents a separate merger thread.

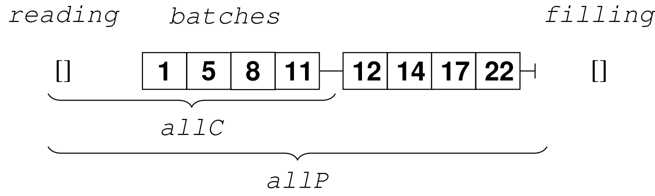
Note that the output of the lower-level mergers serves as input for the higher-level mergers, creating a producer-consumer pattern for the intermediate lists. This means that, to avoid race conditions, the merger threads have to acquire a lock for those lists before reading or writing. This could cause the threads to frequently block each other. To alleviate that, these intermediate lists are split into chunks of a fixed size (Figures 5 and 6 use a size of 4). The producer first writes his entries to a local chunk, and only when this chunk is full, it acquires the lock and submits the whole chunk at once. Likewise, the consumer reads an entire chunk of nodes at a time, and then processes that chunk locally, without the need to acquire the lock again until the chunk is fully processed and a new chunk is needed. Effectively, this turns the intermediate lists into lists of lists. That means there are three classes to look at: `NodeList`, representing a list of `Nodes`; `ListList`, representing a list of `NodeLists`; and `Merger`, defining the behaviour of the merger threads and the overall algorithm. In the following subsections, we examine them each in turn, with particular interest to the producer-consumer pattern for the `ListList` and the merging algorithm itself. While we have to skip many details here for brevity, the full code can be found at [3].

A. *NodeList*

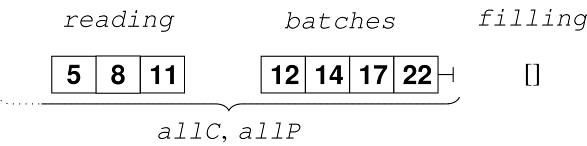
`NodeList` is a sorted linked list of `Nodes`. A recursive predicate `list_perm` represents the access permissions of the list. The method `append` adds a `Node` at the end of the list. Due to the sortedness of the list, the key of the new node is required to be greater than all keys already in the list. The method `extend` adds an entire `NodeList` to the end of another `NodeList`. Again, the new keys must be greater than the keys already in the list. The method `fromTree` uses those two methods to turn a red-black tree into a `NodeList`, by recursively turning the sub-trees and then appending and extending the results. All three methods need little annotation overhead to verify, mostly folding and unfolding.

We use VerCors' internal sequence data type to represent the red-black trees and lists for verification purposes. For example to verify `fromTree`, we use the seq<Node> representations of the input tree and of the output list to ensure that the resulting list contains exactly the nodes from the tree. Note that the sequence representations only store references to the nodes and do not actually have access permissions to

(a) State of the `ListListQueue` at some point during the execution



(b) Adding node 22. *filling* is full and added to *batches*. The producer can update *allP* accordingly, but not *allC*.



(c) To read a node, the consumer first loads the first batch into *reading*, and also synchronises *allC* with *allP*. Then, it returns the first node from *reading*, which is 1. Both *allC* and *allP* still contain that node after it was dequeued (indicated by the dotted brace).

Figure 6: Producing and consuming nodes in a `ListListQueue`

```
1:  NodeList reading;
2:  ListList batches;
3:  NodeList filling;
4:  /*@ ghost seq<Node> allP;
5:      ghost seq<Node> allC;
6:      ghost int readHead;
7:      ghost int batchHead; @*/
8:  /*@ resource producer() = Perm(filling,1)
9:        * NodeList.list_perm(filling)
10:       * Perm(allP,1/2)
11:       * sorted(filling);
12:     resource consumer() = Perm(reading,1)
13:       * NodeList.list_perm(reading)
14:       * Perm(allC,1/2) * Perm(readHead,1)
15:       * isInfix(reading, allC, readHead);
16:     resource lock_invariant() = Perm(batches,1)
17:       * ListList.list_perm(batches)
18:       * Perm(allP,1/2) * Perm(allC,1/2)
19:       * Perm(batchHead,1)
20:       * isPrefix(allC, allP) * sorted(allP)
21:       * isInfix(batches,allP,batchHead); @*/
22: /*@ requires producer();
23:     requires node_perm(node);
24:     ensures producer();
25:     ensures allP+filling
26:             = \old(allP+filling)
27:               + seq<Node>{node}; @*/
28: void append(Node node);
29: /*@ requires consumer();
30:     ensures consumer();
31:     ensures node_perm(\result);
32:     ensures \result = allC[\old(readHead)];
33:     ensures readHead = \old(readHead)+1; @*/
34: Node getNext();
```

Figure 7: The relevant fields of the `ListListQueue` class, its three predicates `producer`, `consumer` and `lock_invariant`, and the methods `append` and `getNext` with (parts of) their specification. For readability, the unfolding commands for predicates were omitted.

the nodes' fields. That way, we can compare the sequence representations (by comparing memory addresses), while the "real" data structures retain full control over the data. Using sequences makes the verification easier, because many features are natively supported by VERCORS, for instance we can assert without additional lemmas that the tail of a sorted sequence is sorted.

B. ListList

`ListList` is a sorted linked list of `NodeLists`. It is similar to a `NodeList`, and thus also has a recursive predicate `list_perm` containing the access permissions for all contained `Nodes`. An `append` method adds a `NodeList` at the end of the `ListList`. As with appending to a `NodeList`, new keys must be greater than the existing keys to ensure sortedness.

A `ListListQueue` class contains the handling of the concurrency via the producer-consumer pattern and the two local chunks (see Figure 7): *reading* is the local chunk that the consumer reads from via the `getNext` method, *filling* the local chunk that the producer writes to with `append`, and *batches* are all chunks in between, which the producer has filled and the consumer still has to read. Figure 6 depicts this, using one of the lists from Figure 5 as an example. Access permissions for these (and all other fields) are distributed over three predicates: `producer` and `consumer` for the respective threads, and a `lock_invariant` for shared elements that can only be accessed when a thread holds the lock. In particular, the producer and the consumer have full access to their respective chunk (Figure 7, Lines 8f and 12f), while *batches* is completely controlled by the lock (i.e. no thread has any access without holding the lock, Lines 16f).

However, this has a major downside: because the *batches* are only accessible in critical sections, method contracts (e.g. for `append`) cannot access this `ListList`. Therefore, the contract cannot directly guarantee correct behaviour, such as the fact that a `Node` added by the producing thread is really stored properly. To circumvent that, ghost variables are used: a seq<Node> called *allP* contains (references to) all `Nodes` that were ever added to the *batches*. Permission to that sequence is shared between the lock and the `producer` predicate (each holding half of it, Lines 10 and 18). That way, the producer has enough permission to use it in method contracts, for instance to ensure that new nodes are added correctly (Figure 7, Lines 25ff), while the lock has enough

permission to ensure that *allP* and *batches* remain in sync (Line 21, see below). Whenever the producing thread acquires the lock, the two halves add up to full access rights, and the thread can update *batches* and *allP* simultaneously (see Figure 6b).

Similarly, a seq<Node> called *allC* is shared between the lock and the consumer predicate (Lines 14 and 18), to ensure in the contract of getNext that the consumer only reads nodes that were written to the *batches*. Note that the producer has no access to that sequence and cannot update it when adding new nodes to the *batches*, so *allC* can get out of sync (see Figure 6b). However, the lock can ensure that *allC* is a *prefix* of *allP* (see Figure 7, Line 20), and whenever the consumer acquires the lock, it synchronises them again (Figure 6c).

We also use *allP* and *allC* in the verification of the merging algorithm (see the section below): as their names suggest, they contain all nodes that were ever added, and do not remove nodes when the consumer reads them. Therefore, when the producer finishes, *allP* contains exactly all the nodes that were ever written by the producer, and likewise for the consumer and *allC*. This helps us to keep track of the nodes and to ensure that the final tree contains exactly the nodes from the input trees. As a consequence of containing *all* nodes, the nodes in *reading* and in *batches* constitute *infixes* of *allP*. Integers *readHead* and *batchHead* store the index within *allP* where *reading* and *batches* begin, respectively (see Figure 7, Lines 15 and 21).

The specifications of append and getNext have to implement this pattern. When the producer adds a node, it appends it to *filling*. If *filling* is full after that, the producer acquires the lock and appends *filling* to *batches*, updating *allP* in the process (the step from Figure 6a to Figure 6b). Thus, *allP+filling* represents all nodes ever written by the producer, and the append method guarantees that the given node is added to that (see Figure 7, Lines 25ff). To read a node, the consumer checks the *reading* list: if it is empty, the consumer obtains the lock and dequeues a batch from *batches*, loading it into the *reading* list. In either case, a node can now be read from *reading*. While holding the lock, *allC* is also updated, to be again in sync with *allP* (the step from Figure 6b to Figure 6c).

So append guarantees that *allP + filling* are exactly those nodes added by the producer, in the order they were added; and getNext guarantees that the node which the consumer read is the next in line at *allC* (see Figure 7, Line 32). Together with the lock's invariant that *allC* is a prefix of *allP*, this guarantees that the consumer reads exactly the nodes written by the producer, in exactly that order.

C. Merger

The Merger uses the functionality described above to merge multiple red-black trees into one, according to Figure 5. The main method mergeTrees takes an array of n Trees and sets up an array of $2{\cdot}n{-}1$ ListListQueues. The first n queues are initialised by n concurrent threads via NodeList.fromTree to contain list representations of the given trees (see left side of Figure 5). After all trees are converted, $n-1$ merger threads are started. Threads and their forking and joining are verified based on [19]. Each merger thread is the consumer of two input ListListQueues and the producer of one output ListListQueue. It merges the input lists by reading a node from each, and writing the one with the smaller key to the output list. The thread maintains the loop invariant that the output list is a sorted combination of the nodes read so far from the two input lists (using *allP* and *allC*). This ensures that the thread creates a merging of the input lists. Due to transitivity, we can thereby verify that the final list contains the nodes from the initial lists, and thus from the given trees. After all merger threads are done, we turn the final list into a balanced tree and apply the necessary colouring to be a valid red-black tree. This tree has now been proved to contain the nodes from the given trees.

V. Discussion

With nearly 4000 lines in total (ca. 600 lines of executable code, 2400 lines of annotations for the verifier, the rest comments or blank), this case study has a considerable size, and uses some advanced verification concepts like magic wands. Nevertheless, verification only took ca. 5 minutes on an Intel Core i7-9750 CPU with 16GB of RAM, using VerCors Version 1.3.0. This highlights how formal verification becomes more and more applicable to real-world scenarios, and not just academic toy examples. However, it also highlights the effort still required by the user to verify a program, with a sizeable overhead of annotations required. Overall, it took one PhD student several hundred hours to obtain a verified implementation, working on it nearly full-time for several months. Admittedly, this also includes familiarising with the original code by NLnet Labs, reimplementing it in Java, and getting to know separation logic and the VerCors tool. While this makes it difficult to pinpoint the exact effort spent on the verification itself, a rough estimation still yields a number of person-hours in the medium three-digit range. This strengthens our resolve to work towards a higher level of automation, for example automatically generating fold and unfold statements like in [20]. Indeed, the tree formalisation alone has already nearly 200 fold and unfold statements, and would thus benefit significantly from such an automation. While the user will still have to do the majority of the intellectual work, such automation techniques can lessen the time spent on doing (and debugging) the grunt work.

The project also emphasises the usefulness of the magic wand operator, and might encourage more tools to support them (cf. "Tool support" in Section II-B). Without it, formalising a recursive data structure such as these trees is more complicated. In fact, an initial draft of the project [21] used dedicated predicates mimicking the behaviour of a wand by encoding a "hole" in the tree where the recursion stops. Managing that hole and ensuring that the respective sub-tree can be re-combined with the outer tree took significant effort, and replacing the custom encoding with magic wands simpli-

fied the verification code considerably, and thus increased the maintainability of the verification. However, the verification time was not affected, indicating that the verifier internally treats magic wands similar to the custom encoding.

Iteratively tweaking and improving the verification like this, even after the code already verifies successfully in some manner, contributed to the large amount of time spent on the verification. However, this also means that we consider the case study to be in a good shape, with most of the improvements that we could think of already included. Nevertheless, there are some things we might do differently in the future. Most notably, the verified code ultimately deviates significantly from the original code by NLnet Labs. This is partly because first attempts at this project were made a few years ago (see [21]), and the support of VerCors for C code was not as good then as it is now, causing the decision to re-implement the trees in Java. We built on top of that, thereby continuing the re-implementation. While there are still parts of C that VerCors does not fully support, this has improved in recent years, and a more direct approach to the verification has become more feasible. Another significant difference is that in the original code, the tree nodes are directly traversed in-order, instead of converting the tree into a separate `NodeList` data structure. Unfortunately, managing the access permissions for such a traversal is not straight-forward, so we decided to use a more explicit transformation in our version. In hindsight, being closer to the original code might have warranted more research on this, and justified a more intricate verification. It also means that our version of the merging algorithm is mostly decoupled from the red-black trees, simply merging lists. Using the original approach of trees doubling as lists would mean that the algorithm is actually merging trees, at least in the first step (afterwards it is still lists of lists).

We think that a generalisation of the way we verified the producer-consumer pattern in the merger is applicable to various other use-cases of a similar pattern: to have three predicates, one for the producer, one for the consumer and one for the lock; and to use ghost variables like *`allP`* that shadow program variables whose permissions are out of scope. Sharing the access rights for those ghost variables with the lock allows one side to ensure that they are in sync with their "real" counterpart, and the other side to specify pre- and post-conditions that (indirectly) refer to the out-of-scope variables. We already considered using this approach in some other smaller case studies.

Likewise, other contexts might reuse the way that we use the wand, for instance when iterating other recursive data structures: both left- and right-hand side of the wand being a pair of an access predicate and a boolean function for sanity checks, combined with ghost variables to store the appropriate values to re-do the sanity checks later. While the idea is not entirely new and resembles the wand e.g. in [9], a lack of tool support for magic wands also means a lack of example usages, so having a "real-world" usage like ours can be useful to other potential users of magic wands.

A. Related work

Initial work on this project was done in the master thesis of Nguyen [21]. However, this only contained the tree formalisation, not the merging algorithm. Also, it was missing the *delete* operation, and as mentioned above, the *getMin* operation did not use a wand. Instead, the entire formalisation used a custom predicate to account for potential holes in the tree (even though the trees only have holes in a few places in the code). We also improved upon that initial version in various other, smaller ways.

Peña [22] describes the verification of the red-black tree operations using the tool (and programming language) Dafny. While Dafny has support for separation logic, he does not explicitly mention access permissions, and in particular does not use a magic wand. Additionally, he focusses on the sub-type of *left-leaning* red-black trees, which simplifies the verification. Our approach does not have that constraint.

There have been case studies about verifying other data structures in separation logic: for example, Da Rocha Pinto et al. [23] verify a form of B-tree using concurrent separation logic, and actually found a bug in the published algorithm for B-trees that they used. Lammich [24] uses a priority queue as test case for a refinement framework in Isabelle based on separation logic, but without concurrency. Krishna et al. [25] verify templates in Iris/Coq and use the resulting annotations to guide the user in annotating any implementations of those templates (the link between the template and the implementation is still manual). Again, they use data structures like B-trees as case studies to evaluate their approach. These verified data structures can complement the red-black trees from this case study to form a library of correct data structures (see Section VI-A). Note that red-black trees correspond to 2-3-4 trees, a special form of B-trees (see [22]). However, the verification work above does not directly match ours, as Da Rocha Pinto et al. focus on dealing with multiple threads accessing the tree concurrently, while our `tree_perm` predicate ensures that this does not happen, and Krishna et al. focus on verifying the templates. Neither investigates a merge algorithm.

Blom et al. [9] verify the tree delete problem using wands in a very similar way. They do not address red-black trees, and the complexity that they bring to the operation. In fact, the tree delete problem is merely an illustrating example, and their focus is on transforming specifications involving magic wands and other complex constructs into simpler specifications, which other tools without support for those constructs can also verify.

Note that the producer-consumer pattern on the `ListList` is comparable to an asynchronous channel, via which one thread sends nodes (or lists of nodes) to the other. There are publications on verifying channel communications. While those relating to protocol verification are not relevant here, the work of Bell et al. [26] goes into a similar direction, linking received values to sent values by storing a history of sent and received values and comparing them after the threads are joined. However, the relation is only made explicit at the end,

and not during the communication, as it is in our lock invariant.

VI. Conclusion

In this paper, we have described a formalisation of the red-black tree data structure in permission-based separation logic. We used permission-based separation logic as a formal framework, because we ultimately wanted to verify a concurrent algorithm: merging an array of red-black trees into one big red-black tree. The merging uses a list of lists of nodes as an intermediate representation, which was verified using a producer-consumer pattern. We also verified the correctness of basic tree operations, such as adding and deleting nodes, using the separating implication (i.e. *magic wand*). While these operations (and the wand) were not strictly necessary for the merging algorithm, we consider them essential for a data structure, and any formalisation should be robust enough to support them. We therefore verified both the merging algorithm and the operations using the VerCors tool. The verification required significant overhead from the user in terms of annotations for the verifier, but ultimately, VerCors was able to prove the correctness in reasonable time. This highlights that the deductive verification of industrial use cases is feasible, but more research and tool support is needed to reduce the overhead and thereby improve the practicality of using verification in industrial software development.

A. Future work

We are currently planning to improve the support for magic wands in VerCors. While VerCors fully supports magic wands, and we could verify this case study without issues, the back-end solver Viper offers even larger support, for example regarding the kind of expressions supported inside the wand expression. Also, the relation of nearly four-to-one of lines of annotations versus executable code highlights the need to automatically generate annotations, to ease the overhead for the user.

One could also think of other (ideally concurrent) use cases for red-black trees. Alternatively, verifying other, similar data structures (e.g. priority queues, B-trees) may be of interest, to form a library of verified data structures. Ideally, there is another industrial usage of the respective data structure to again have the combination of academic and industrial use case that characterised this case study.

Acknowledgments

This work is supported by the NWO VICI 639.023.710 Mercedes project.

References

[1] M. Huisman and R. E. Monti, "On the industrial application of critical software verification with vercors," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2020, pp. 273–292. [Online]. Available: https://doi.org/10.1007/978-3-030-61467-6_18

[2] C. Dross and J. Kanig, "Recursive data structures in spark," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Springer International Publishing, 2020, pp. 178–189.

[3] L. Armborst, "Permission-based verification of red-black trees and their merging - code," Jan 2021. [Online]. Available: https://data.4tu.nl/articles/software/_/13611578

[4] C. Hoare, "An axiomatic basis for computer programming," vol. 12, no. 10, pp. 576–580, 1969.

[5] J. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*. IEEE Computer Society, 2002, pp. 55–74.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *POPL*, 2005, pp. 259–270.

[7] J. Smans, B. Jacobs, and F. Piessens, "Implicit Dynamic Frames," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, pp. 2:1–2:58, 2012. [Online]. Available: http://doi.acm.org/10.1145/2160910.2160911

[8] A. J. Summers and S. Drossopoulou, "A formal semantics for isorecursive and equirecursive state abstractions," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Springer Berlin Heidelberg, 2013, pp. 129–153.

[9] S. Blom and M. Huisman, "Witnessing the elimination of magic wands," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, pp. 757–781, Nov 2015. [Online]. Available: https://doi.org/10.1007/s10009-015-0372-3

[10] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2016, pp. 41–62.

[11] J. Harland and M. Winikoff, "Agent negotiation as proof search in linear logic," in *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, ser. AAMAS '02. Association for Computing Machinery, 2002, p. 938–939.

[12] J. Smans, B. Jacobs, and F. Piessens, "Verifast for Java: A tutorial," in *Aliasing in Object-Oriented Programming*, ser. LNCS, D. Clarke, T. Wrigstad, and J. Noble, Eds., vol. 7850. Springer, 2013.

[13] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors Tool Set: Verification of Parallel and Concurrent Software," in *iFM*, ser. Lecture Notes in Computer Science, vol. 10510. Springer, 2017, pp. 102–110. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-66845-1_7

[14] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin, *JML Reference Manual*, Feb. 2007, dept. of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[15] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, C. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[16] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, 1978, pp. 8–21.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[18] NLnet Labs, "Red-black tree merging algorithm." [Online]. Available: https://github.com/NLnetLabs/nsd/blob/cbcf0fd/pzl/pzl4nsd.c

[19] A. Amighi, C. Haack, M. Huisman, and C. Hurlin, "Permission-based separation logic for multithreaded Java programs," *LMCS*, vol. 11, no. 1, 2015.

[20] F. Vogels, B. Jacobs, F. Piessens, and J. Smans, "Annotation inference for separation logic based verifiers," in *Formal Techniques for Distributed Systems*, R. Bruni and J. Dingel, Eds. Springer Berlin Heidelberg, 2011, pp. 319–333.

[21] ir. H.M. Nguyen, "Formal verification of a red-black tree data structure," Master's thesis, March 2019. [Online]. Available: http://essay.utwente.nl/77569/

[22] R. Peña, "An assertional proof of red–black trees using Dafny," *Journal of Automated Reasoning*, 2019.

[23] P. Da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse, "A simple abstraction for complex concurrent indexes (extended version)," 01 2011.

[24] P. Lammich, "Refinement based verification of imperative data structures," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. Association for Computing Machinery, 2016, p. 27–36.

[25] S. Krishna, N. Patel, D. Shasha, and T. Wies, "Verifying concurrent search structure templates," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. Association for Computing Machinery, 2020, p. 181–196.

[26] C. J. Bell, A. W. Appel, and D. Walker, “Concurrent separation logic for pipelined parallelization,” in *Static Analysis*, R. Cousot and M. Martel, Eds. Springer Berlin Heidelberg, 2010, pp. 151–166.

APPENDIX

Figures 8 and 9 provide more implementation details, in addition to what Section III provides. For the full implementation, see [3].

```
1:  /*@   requires tree_perm(node);
2:      boolean noDBlack(Node node) = node != null
3:          ⇒ !node.dblack
4:            && noDBlack(node.left)
5:            && noDBlack(node.right);

6:        requires node_perm(node);
7:      int getBlacks(Node node) = node = null ? 1
8:          : node.dblack ? 2
9:              : node.colour ? 0
10:                 : 1;

11:       requires tree_perm(node);
12:     int blackHeight(Node node) = node = null ? 1
13:         : node.dblackNull ? 2
14:             : getBlacks(node)
15:               + max(blackHeight(node.left),
16:                     blackHeight(node.right));

17:       requires tree_perm(node);
18:     boolean blackBalanced(Node node)
19:         = node = null ? true
20:           : blackHeight(node.left)
21:               = blackHeight(node.right)
22:             && blackBalanced(node.left)
23:             && blackBalanced(node.right); @*/
```

Figure 8: The definitions of the remaining red-black properties: Property 4 as `noDBlack` and Property 2 as `blackBalanced`, based on `blackHeight`. Again, the unfolding commands for predicates were omitted.

```
1:  /*@ requires node != null * tree_perm(node);
2:      requires node.left != null;
3:      // regarding double-red:
4:      //   - new connections must not create double-red
5:      requires !getColorLeft(node)
6:               || !getColorRight(node);
7:      //   - allow only specific cases of double-red
8:      requires noDoubleRed(node)
9:               || dbRedAtLeft(node) || ...;
10:     // regarding double-black: only allow one specific case
11:     requires noDBlack(node)
12:              || dblackAtRight(node);
13:     ensures \result != null * tree_perm(\result);
14:     // preservation properties
15:     ensures \old(blackHeight(node))
16:              = blackHeight(\result);
17:     ...
18:     // fixed double-red in most cases
19:     ensures \old(dbRedAtLeft(node))
20:              ? dbRedAtRight(\result)
21:              : noDoubleRed(\result);
22:     ensures \old(noDBlack(node))
23:              ⇒ noDBlack(\result); @*/
24: Node rotateRight(Node node);
```

Figure 9: The `rotateRight` method rotates a given tree to the right. Its specification requires a lot of case distinctions, this is only an excerpt. Again, we omitted folding and unfolding of predicates for readability.