Specification of Red-Black Trees: Showcasing Dynamic Frames, Model Fields and Sequences

Daniel Bruns

Complex data structures still pose a major challenge in specification and verification of object-oriented programs. Leino and Moskal have proposed a suite of benchmarks for verification tools, nicknamed "VACID-0" [1]. In contrast to similar papers, the tasks of VACID-0 do not only include verification in terms of an observable behavior but also of internal workings of algorithms and data structures. The arguably most difficult target are so-called red-black black trees (see, e.g., [2, Chap. 13]). In this contribution, we present an implementation and specification in Java/JML* (i.e., KeY's extension to JML with dynamic frames). It makes use of several new features: first and foremost the dynamic frame theory [3, 4], model fields, the sequence ADT, as well as some newly supported features from standard JML.

Red-black trees. A red-black tree is a kind of binary search tree with well-regarded performance properties, such as search, insertion, or deletion take $\mathcal{O}(\log n)$ time at worst. Besides a key/value pair, each node in the tree has a "color" which can either be red or black, thus giving that name. The following properties are invariants which guarantee that the tree is well-behaving: (i) The root is black. (ii) Every leaf[1] is black. (iii) If a node is red, then both its children are black. (iv) For each node, the left and right subtree equal in the number of black nodes. Algorithms for insertion and deletion are essentially the same as for basic search trees, but followed by a 'fixing operation,' which recolorizes nodes and performs elementary rotations.

Specifying with dynamic frames. The current development branch of KeY is based on the theory of dynamic frames, which is a decisive step towards modularity of proofs. In JML terms, a dynamic frame is a model field of a type 'set of locations,' which may be used more liberally than standard JML constructs like data groups. Typically, classes have a dynamic frame 'footprint' which contains its private fields. Then, showing disjointness of footprints is evidence for heap separation.

Due to their highly non-modular semantics, the concept of class invariants, which are to be respected throughout the whole program, has been discarded in this version. A more modular and fine-grained invariant control can be established through the addition of `boolean` model fields to pre- and post-conditions. Invariants may still be given in JML*; they are understood as `represents` clauses of an implicit model field `<inv>`, which is added to the pre- and postconditions of all non-`helper` methods in the class where it is declared. In JML*, the invariant of a particular object is referenced through the `\invariant_for` expression.

Specifying red-black trees. Our implementation of red-black trees consists of two classes `Node` and `Tree`. The former only contains its characteristic information in package-private fields; all tree-mutating methods are found in class `Tree`. The implementation has no `null` references; instead, it employs a special dummy node `NIL`. In order to allow a modular verification, `Tree` implements a `Map` interface which provides the necessary methods and specifications on an abstract level—mainly employing model fields and dynamic frames. By that, we achieve the requirement of VACID-0 that a given test harness is to be verified on interface specification alone.

As described above, the invariants which axiomatize red-black trees are concerned with single nodes (or subtrees, respectively), but those have to hold for all nodes in the tree. However, in an implementation

[1] Leaves in this context are always dummy nodes which do not hold a key.

```
/*@ invariant parent==NIL || parent.left==this || parent.right==this;
  @ invariant height == (this == NIL ? 0 :
  @     (left.height > right.height ? left.height : right.height)+1);
  @ invariant \invariant_for(left) && \invariant_for(right);
  @ accessible \inv : footprint, left.footprint, right.footprint
  @     \measured_by height; @*/
```

Figure 1: 'Low-level' invariants of the `Node` class.

of such complex algorithms, there are several private methods which temporarily violate those invariants while there are others which just establish them. We also need some 'low-level' invariants which provide more elemental properties, such as non-`null`-ness or that a node is always a child of its parent. In addition, invariants are recursive: a node's invariant is sufficient for its children's invariants. To this end, in JML* a termination witness may be given to represents clauses (as to loops and recursive methods). In our example this is given by the maximum height of a node (counted from the leaves upwards). The low-level invariants are shown in Fig. 1. To sum up, the dynamic frame methodology suits these requirements well since it provides a more fine-grained control over invariants.

On the higher level when regarding the whole tree, we need to keep track of the nodes in the tree as those may be added or removed. We have chosen to specify an abstraction which disregards the tree structure and simply contains the nodes in a sequence. Sequences are very handy since we can easily (i.e., without recursion) give specifications to assert the existence of a given element.

Conclusion. A highly challenging example like red-black trees makes an excellent showcase for the new capabilities of KeY, in particular dynamic frames and the increased coverage of JML features. Another goal was to investigate whether there is need for difficult model fields in specifications, e.g., reference-typed model fields or non-functional represents clauses. In a related work [5], we discuss the semantical issues with those and raise the supposition that those do not occur at all in practice. This case study supports that supposition.

Actual (feasible) verification using KeY is not yet possible, however. There two main reasons: Firstly, the new methodology needs to mature; there are still implementation effort necessary (in particular adaption of strategies to new rules).[2] Secondly, we still need to gain some experience from "simpler" examples such as double-linked lists in order to see whether our invariants are sufficient.

References

[1] K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, Edition 0. In *Tools and Experiments workshop at VSTTE 2010*, Edinburgh, UK, 2010.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects Computing*, 23(3):267–288, 2011.

[4] Benjamin Weiß. *Deductive Verification of Object-oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, January 2011.

[5] Bernhard Beckert and Daniel Bruns. Formal semantics of model fields in annotation-based specifications inspired by a generalization of Hilbert's ε terms. unpublished, 2011.

[2] Preliminary attempts to verify the harness have led to (incomplete) manual proofs with some 100 branches or some 1000 branches when attempting to let it run automatically.