Int J Softw Tools Technol Transfer (2015) 17:729–744
DOI 10.1007/s10009-013-0293-y

Implementation-level verification of algorithms with KeY

Daniel Bruns · Wojciech Mostowski · Mattias Ulbrich

Published online: 17 November 2013

Abstract We give an account on the authors' experience and results from the software verification competition held at the Formal Methods 2012 conference. Competitions like this are meant to provide a benchmark for verification systems. It consisted of three algorithms which the authors have implemented in Java, specified with the Java Modeling Language, and verified using the KeY system. Building on our solutions, we argue that verification systems which target implementations in real-world programming languages better have powerful abstraction capabilities. Regarding the KeY tool, we explain features which, driven by the competition, have been freshly implemented to accommodate for these demands.

Keywords Formal verification · Benchmark · Java Modeling Language · Theorem prover

1 Introduction

This paper is a thorough experience report of the members of the KeY team from the verification competition which was held on August 30th and 31st 2012 during the Formal Methods conference in Paris. The competition was organised by Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan with the aims *"to bring together those interested in formal verification [… and] to evaluate the usability of logic-based program verification tools in a controlled experiment that could be easily repeated by others."* The competition consisted of three algorithmic problems stated in natural language or pseudo-code, which were to be implemented in a programming language of the competitors' choice, specified, and formally verified with a tool also of the competitors' choice. Each challenge was to be solved in a limited amount of time and represented a different general category of algorithms. The problems were as follows:

1. The *Longest Repeated Substring* challenge (Sect. 3) deals with structural analysis of a flat data structure (array of characters/integers),
2. the *Prefix Sum* challenge (Sect. 4) also works with arrays, but performs non-trivial arithmetic in-situ operations on the array,
3. the *Deletion in a Tree* challenge (Sect. 5) is about inspecting and modifying a non-cyclic linked data structure of binary search trees.

Two of the authors of this paper, Daniel Bruns and Wojciech Mostowski, participated in the competition as a team, using the Java Modeling Language (JML) [16] to specify Java implementations of the challenges and verify them with the KeY tool [3]. Mattias Ulbrich contributed to the post-competition work to further develop our solutions and tool support. The full results can be found at http://formal.iti.kit.edu/~bruns/VerifyThis/. All of the authors of this paper are experienced users and have been active developers of the KeY system for several years. Moreover, all of them participated in similar events previously [5,10,15], always with the combination of Java, JML, and KeY.

D. Bruns · M. Ulbrich
Karlsruhe Institute of Technology, 76127 Karlsruhe, Germany
e-mail: bruns@kit.edu

M. Ulbrich
e-mail: ulbrich@kit.edu

W. Mostowski (✉)
University of Twente, 7500 AE Enschede, The Netherlands
e-mail: w.mostowski@utwente.nl

The challenges of this competition turned out to be very demanding. Only the first challenge could be properly addressed during the time of the competition; the other challenges were dealt with off-line after the event. The post-competition effort involved working on the challenges themselves as well as improving KeY. The tool turned out to run short of certain features indispensable for the challenges. In this respect, such a competition advances tools such as some of the new features which have been inspired by this competition and which the authors have implemented will provide support for verification beyond the competition challenges.

The challenges had been selected to be language- and paradigm-agnostic, not requiring any particular feature like object-orientation. This choice allowed many different approaches to participate in the competition, but, on the other hand, led to a selection of problems concentrating on *algorithmic* rather than *programming-language* issues of programs.

The KeY tool verifies programs written in Java and does not abstract away from the code. Hence, verification of algorithms with KeY requires one to deal with two problems at a time: (1) the difficulty to verify a non-trivial algorithm and (2) the implementation issues of the real-world programming language Java (which includes dynamic method lookup, framing, checking the absence of runtime exceptions, etc.). Despite the improvements we introduced, KeY does not seem to be overly suitable for these algorithm-oriented programs as it is geared towards verification of Java programs (having reasoning support for various Java-specific features, in particular inheritance and heap data structures).

2 Verification with the KeY system

KeY [3] is a source-code-based verification system for sequential Java programs. Specifications can be given in an extension to the Java Modeling Language (JML), called JML*. At the core of KeY is a dedicated interactive theorem prover for first-order dynamic logic. Work on KeY started in 1998; it is being developed at Karlsruhe Institute of Technology; Chalmers University of Technology in Gothenburg, Sweden; Technical University of Darmstadt, Germany; and to a lesser degree at the University of Twente.

April 2013 marked the release of KeY 2.0, which introduced several features for abstraction, in particular the possibility to modularly verify recursive method implementations. KeY is free and open source software and can be downloaded from http://key-project.org/. This stable version, however, does not include some of the experimental features we implemented during the work on the competition challenges, in particular specification through model methods that we describe later in Sect. 5.

2.1 Java Modeling Language

The Java Modeling Language (JML) [16] is a popular and powerful specification language for Java programs based on the design by contract paradigm; the main concepts are class invariants and method contracts. JML integrates seamlessly into Java as it is embedded inside comments in Java source code and JML expressions extend Java expressions in a natural way. By now, JML has become the de-facto standard in formal specification of Java source code. Besides KeY, several other verification tools are built on JML; for an overview see [6].

Contracts in JML essentially consist of preconditions, indicated by the keyword `requires`, postconditions (`ensures`), and frame conditions (`assignable`). More than one of each of these specification constructs may be declared, in which case they are conjoined. In this paper, we are only concerned with *normally terminating* methods, i.e., terminating and doing so without throwing any exceptions. The termination itself is the default property for method contracts in JML; the absence of exceptions is denoted with the keyword `normal_behavior` at the beginning of a contract.

Class invariants describe a global state which must be preserved by all methods. In this paper, we understand invariants of a class `C` as being implicitly added to pre- and post-conditions in the contracts of every method declared in `C`. Methods can be exempt from invariants by marking them as `helper`.

JML also provides auxiliary annotations such as loop invariants. Loop invariants are marked by the keyword `maintaining`. Since termination is required for all of the challenges discussed in this paper, we also use `decreasing` clauses, which are followed by an integer *variant* expression, which must, for each succeeding loop iteration, evaluate to a strictly smaller value than in the preceding iteration. Additionally, as opposed to standard JML, for loops changing any data on the heap KeY requires one to state corresponding framing conditions with the `assignable` clause. The rationale behind this is that invariant specifications for loops are treated as the same kind of abstraction as contracts are for methods and hence loops also need to have their framing conditions stated. We explain the particular framing conditions as implemented in the KeY logic in Sect. 2.2 after the following paragraphs on JML expressions and abstraction.

Expressions in JML can be almost any side-effect free Java expression. This includes all built-in numerical and boolean operators as well as calls to pure methods. On top of that, JML contains a boolean implication operator `==>` and quantified expressions. Universal quantification is

expressed in the general shape (`\forall T x; guard; body`), where `T` is the type over which the expression ranges, `x` is a variable of that type, and `guard` and `body` are boolean expressions. Existential quantification works analogously. Syntactically similar to quantified expressions, JML offers numerical comprehension expressions `\sum` and `\product`.

In a post-condition of a contract it is often necessary to refer to data on the heap in the state when the method started its execution. This is called a pre-state and in JML can be accessed with the `\old` keyword. For instance, the following expression states that the value of a field `x` has been increased: `x > \old(x)`. This unary operator may be applied to any expression. Method parameters are always evaluated in the pre-state. Finally, in a post-condition to a non-`void` method, the expression `\result` refers to the return value.

Abstraction in JML is provided through *model fields* [7] and *model methods*. While similar in appearance to fields and methods in Java, those are specification-only elements, which may use the full arsenal of JML expressions, including some KeY-specific enhancements such as a built-in abstract data type of sequences or sets. Model fields are loosely coupled to the concrete system state through `represents` clauses, which are understood as logical axioms. Model methods offer a substantially more powerful abstraction mechanism compared with model fields in that (a) they also include properties for the defined axioms, i.e., lemmas, and (b) they allow parametrised queries rather than parameterless observations. For instance, model methods can be used to query the nth element in a list, where n is an arbitrary expression possibly referring to other concrete or abstract expressions.

None of the existing verification tools properly implement model methods as defined in JML. Applying model methods to solve the third challenge is one of the contributions of this paper which goes beyond the competition scope. That is, the third challenge is the first non-trivial verification case study using the newly implemented model methods mechanism in KeY. We shall go into details of model methods in Sect. 5 devoted to the third challenge.

Specification-only program elements, such as model methods, allow the use of *Abstract Data Types* (ADTs). These are indispensable when reasoning about concrete data structures which are object-based, e.g., linked lists or trees. While concrete implementations carry some overhead (i.e., references), we are usually only interested in properties on the payload data. ADTs are primitive in the Java sense; two instances containing the same data are always identical. JML* provides the two ADTs `\seq` (finite sequences, i.e., tuples, of any type) and `\locset` (finite sets of memory locations; see below).

2.2 Modular verification using dynamic frames

KeY supports *modular verification* based on the design by contract paradigm. This means that single methods are being verified against their contract alone assuming any possible environment. This feature ensures (preservation of) correctness in proofs when some other parts of the implementation are unknown or have been changed. It is essential that contracts do not only contain pre- and post-condition pairs (to describe what an implementation must achieve), but also *frame conditions* (to describe what must be *preserved*). A frame is a set of heap locations which a method may at most write to.

In many situations, in particular when changes are local to one object, it is sufficient to enumerate the locations of a frame. This is known as a static frame. In the presence of (object-based) dynamic data structures, however, there is a need to give a frame for the data structure as a whole, e.g., one that includes also locations reachable from the given execution context. In the *dynamic frames* approach [13,26], frames can be given in an abstract, possibly recursive, definition. For instance, the frame of a linked list would be the union of the head node's locations (for its local data) and the frame of the tail (if it is not `null`). The set of heap locations that a given program accesses is commonly called a *footprint*. In JML* the handling of frames is delegated to a dedicated primitive type of location sets denoted with `\locset`. The frames built by combining location sets are typically stored in the definitions of model fields or model methods that can in turn be referred to in `assignable` clauses.

A typical specification pattern with dynamic frames are separation invariants. For a linked list, for instance, an invariant is that the locations of a node are not included in the footprint of its successor; this implies acyclicity, in particular. Tree data structures usually have the additional property that the footprints of a node's children must be pairwise disjoint. Such a property appears in the challenge in Sect. 5.

An approach serving the same goals as dynamic frames is *Separation Logic* with abstraction [20], where separation properties are inherent in formulae. Specifications in separation logic are usually more concise—there is no need to explicitly give dependencies.

2.3 Using the KeY prover

The core of KeY is made up of an interactive theorem prover for first-order Java dynamic logic (JavaDL) [11], which can be seen as a generalisation of Hoare logic. Construction of proofs in KeY corresponds to *symbolic execution*. That is, for every possible execution branch a stepwise transformation of the program leads to a set of constraints describing the corresponding final program state, which can be then evaluated against the stated properties using classical first-order reason-

ing. The symbolic execution method is an improvement [14] over a more commonly used *verification condition generation* (VCG) technique, in which programs and properties are collectively transformed using weakest precondition calculus to one big proof obligation formula which is then discharged using a general purpose theorem prover. The usually large size of the resulting formula is often a bottle-neck in VCG based approaches. In addition, symbolic execution provides more feedback since formulae are more human-readable and allow debugging of the program.

Overall, KeY is able to handle a rich subset of sequential Java programs. In particular, it is one of the few formal verification tool which consider static initialisation. Other supported features include `Strings` (including string pool), enhanced `for` loops, and Java Card atomic transactions [17].

In general, KeY has been designed as an interactive theorem prover. That means that the user is responsible for finding a proof, in particular, for providing values for quantifier instantiations. Despite that fact, KeY offers a high degree of automation. Many problems in program verification can be solved fully automatically. In addition, powerful SMT solvers, such as Z3 [8], can be plugged in to prove first-order logic subgoals more efficiently than KeY. This applies mostly to arithmetical problems such as systems of inequations.

While proofs for valid formulae can be found automatically, failed automated proof attempts are not always helpful. Such a situation may result from an inexhaustive proof search, but it is more likely that the formula to be proven is just not valid, meaning that the program does not satisfy the original specification. It is often not obvious for a human user to understand the kind of counter-examples provided through open proof goals produced by an automated proof engine. The main reason is that efficient automation relies on normal forms which tend not to be human-readable. In particular, it is not obvious which formulae are corollaries from the original assumptions and which ones derive from original proof goals. Humans, on the other hand, prefer to make frequent case distinctions and simplifications of terms to keep a focus on the original proof goals—possibly at the expense of being less efficient.[1]

This has led us to pursue a *semi-automated* proof style in which the user does not apply every proof step by hand, but chooses an automated strategy at certain points of interest in the proof. KeY also provides compound interaction steps, so-called *strategy macros*, which combine the application of several basic deduction steps to achieve a specific purpose. In the challenges presented in this paper, we have made frequent use of the following macros:

Propositional expansion (without splits) apply only non-splitting propositional rules,
Propositional expansion (with splits) apply only propositional rules,
Finish symbolic execution apply only rules for modal operators, i.e., execute Java programs symbolically,
Close provable goals automatically close all open goals for which this is possible, but do not apply any rule to goals which cannot be closed.

2.4 Related tools

Other state-of-the-art verification systems for real-world imperative and object-oriented programming languages like Java, C, and C# are mostly based on VCG, such as Spec# [1], Krakatoa [9], or VCC [23]. They typically combine several subsystems in a tool chain, involving general purpose theorem provers. With these systems, the user does not interact through proof search but through adding auxiliary annotations to the program. The VeriFast [12] system for verification of Java and C programs performs symbolic execution like KeY, but does not offer user interaction either.

The KIV verification system follows a similar approach as KeY by providing a dedicated interactive theorem prover with a calculus for dynamic logic [25]. However, while KIV is a general verification framework for various logics, the KeY system is specialised for the modular verification of programs implemented in the Java programming language. This enables KeY to be more optimised to this task and achieve a higher degree of automation.

3 Longest Repeated Substring

The Longest Repeated Substring problem (LRS; cf. [24]) occurs in string searching. It can be described as given an array of integers (or elements of any other alphabet) $\{a_0, \ldots, a_n\}$, find a sub-array $\{a_s, \ldots, a_{s+\ell-1}\}$ which appears at least twice and there is no repeated sub-array longer than ℓ. For example, in the array $a = \{3, 0, 3, 4, 3, 4, 1\}$, the only LRS is $\{3, 4\}$.

While LRS can be solved naïvely in $O(n^3)$, the more efficient algorithm to be used here first indexes all suffixes and sorts them lexicographically. Then, a longest common prefix (LCP) over neighbouring elements (w.r.t. this ordering) is a solution. We will first show how to verify an implementation of LCP and then return to LRS. LCP could be specified and verified on-site at the competition in the given time; LRS was verified off-line.

3.1 Longest common prefix

The LCP problem can be described as follows: given an array $\{a_0, \ldots, a_n\}$ and integers $x, y \in [0, n]$, find the maximum integer ℓ such that sub-arrays $\{a_x, \ldots, a_{x+\ell-1}\}$ and

[1] For an empirical analysis of user experience with KeY, see [2].

```
/*@ normal_behaviour
  @ requires 0 <= x && x < a.length;
  @ requires 0 <= y && y < a.length;
  @ requires x != y;
  @ ensures 0 <= \result;
  @ ensures \result <= a.length - x;
  @ ensures \result <= a.length - y;
  @ ensures (\forall int i; 0 <= i && i < \result;
  @                         a[x+i] == a[y+i] );
  @ ensures \result == a.length-x
  @      || \result == a.length-y
  @      || a[x+\result] != a[y+\result];
  @ strictly_pure @*/
static int lcp(int[] a, int x, int y) {
    int l = 0;
    /*@ maintaining 0 <= l && l+x <= a.length
      @             && l+y <= a.length && x != y;
      @ maintaining (\forall int z; 0 <= z && z < l;
      @                           a[x+z] == a[y+z] );
      @ decreasing a.length-l;
      @ assignable \strictly_nothing; @*/
    while (x+l < a.length && y+l < a.length
            && a[x+l] == a[y+l]) l++;
    return l;
}
```

Listing 1 Implementation of LCP with specification

$\{a_y, \ldots, a_{y+\ell-1}\}$ are equal ($\ell = 0$ means that there are no duplicates at all). This can be solved by simply iterating over the array once. Listing 1 shows an iterative implementation in Java and its specification in JML. In Ls. 8f., the contract states that there is a common sub-array of length ℓ, where ℓ is the result of `lcp()`. Lines 10ff. imply that ℓ is a maximal solution: the adjacent next cells $a_{x+\ell}$ and $a_{y+\ell}$ (if they exist) do not contain the same element. As we have explained in Sect. 2.1 above, per default, this contract already states that `lcp()` always terminates without throwing an exception.[2]

In order to verify the implementation with a loop for any number of iterations, we need to provide additional annotations to guide the prover. We have given two invariants, one in Ls. 16f. which abstracts from the pre-condition and one in Ls. 18f. which abstracts from the post-condition (i.e., there is a solution candidate up to the current loop iteration). Line 20 contains a variant which decreases towards zero as the loop index increases. Line 21 gives a frame condition for the loop. The keyword `\strictly_nothing` here says that the loop does not change the heap state in any way. This property is known as *strict purity*, as opposed to what otherwise is considered 'pure' in software verification—namely that there are possible changes to heap which are not observable.

Provided these loop annotations, the implementation can be verified against its specification without any user interaction in 1,800 proof steps, taking 8 s of computation time.[3] In particular, KeY is able to find all quantifier instantiations. The solution found on-site is similar in size, but, due to a different variable ordering, required one quantifier instantiation to be applied by hand.

[2] In this case, we also need to prove that no runtime exceptions, such as `NullPointerException` or `IndexOutOfBoundsException`, are raised.

Table 1 Suffixes of an example array, unsorted (left) and sorted in ascending lexicographic ordering (right)

0	{3, 0, 3, 4, 3, 4, 1}	1	{0, 3, 4, 3, 4, 1}
1	{0, 3, 4, 3, 4, 1}	6	{1}
2	{3, 4, 3, 4, 1}	0	{3, 0, 3, 4, 3, 4, 1}
3	{4, 3, 4, 1}	4	{3, 4, 1}
4	{3, 4, 1}	2	{3, 4, 3, 4, 1}
5	{4, 1}	5	{4, 1}
6	{1}	3	{4, 3, 4, 1}

3.2 Suffix array

The time-optimal algorithm for solving LRS does not traverse the original array a, but indices to suffix sub-arrays. These indices are sorted on a lexicographic ordering of the suffixes. This already ensures that solution candidates are neighbours in this so-called suffix array. In Table 1, on the left-hand side, the suffixes for the above example array are shown. The number on the left is the pointer to the position where the respective sub-array starts. On the right-hand side, these suffixes are sorted lexicographically ascending. The *suffix array* to a contains the pointers in that order; in this example it is $\{1, 6, 0, 4, 2, 5, 3\}$.

In the Java implementation, the original array and its suffix array are both contained in one class `SuffixArray` (see Listing 2) . The invariant in Ls. 6ff. states that `suffixes` is a permutation of pointers into `a`. Sortedness is specified in the last invariant (Ls. 10ff.): for any two neighbouring suffixes, there is an index j such that the j-th entry is larger in the first suffix array (or j equals the length of the second suffix array), and all entries until j are equal.

3.3 Proving LRS

As explained above, provided a sorted suffix array, it is sufficient to find an LCP in neighbouring entries. Listing 3 shows a Java implementation, where `sa` is a field of type `SuffixArray`. It computes pointers s and t to two occurrences of an LRS and the length ℓ of the LRS, which are stored in fields for technical reasons since in Java there is at most one return value. For solving the LRS problem, it would be sufficient to return just one pointer, but then we

[3] For all proofs, KeY was running on standard desktop computers in single-processor mode.

```
public final class SuffixArray {
    final /*@ spec_public @*/ int[] a;
    final /*@ spec_public @*/ int[] suffixes;

/*@ invariant a.length == suffixes.length;
  @ invariant
  @  (\forall int i; 0 <= i && i < a.length;
  @    (\exists int j; 0 <= j && j < a.length;
  @      suffixes[j] == i));
  @ invariant (\forall int i; 0 < i && i < a.length;
  @  (\exists int j;
  @    0 <= j && j < a.length-suffixes[i];
  @     ((j < a.length - suffixes[i-1]
  @       && a[suffixes[i]+j] > a[suffixes[i-1]+j])
  @      || j == a.length-suffixes[i-1]) &&
  @      (\forall int k; 0 <= k && k < j;
  @        a[suffixes[i]+k] == a[suffixes[i-1]+k]));
  @*/
}
```

Listing 2 Invariants of a suffix array

would have to existentially quantify over the second occurrence. A second optimisation is to update these fields only at the end of the method, after the loop has finished. Since the loop is then executed in one single heap state, it is easier to verify.

Specifying `doLRS()` is straightforward (see Listing 4): we only assume the class invariant of `SuffixArray` in L. 1 and that the array length is non-trivial. The post-condition in Ls. 8f. states that the sub-arrays starting at s and t, respectively, are equal for the first ℓ elements. We also prove that these point to different sub-arrays, or there is no repetition at all (L. 10). Lines 11ff. state that the solution candidate is indeed maximal: there are no indices i and k such that they point to a repeated substring of length $\ell + 1$.

Proving correctness of the LRS implementation requires some user interaction. The difficult part here is to prove maximality of the computed solution. Completeness properties like this one are the most challenging problems in software verification. It is not obvious to see that maximality follows from the fact that the suffix array is sorted and it suffices to only check neighbours. A further complication is that the specification of LRS as displayed in Listing 4 refers to the

```
public void doLRS() {
    int s = 0; int t = 0; int l = 0;
    for (int x=1; x < sa.a.length; x++) {
        int length = LCP.lcp(sa.a, sa.suffixes[x],
                             sa.suffixes[x-1]);
        if (length > l) {
            s = sa.suffixes[x];
            t = sa.suffixes[x-1];
            l = length;
    }   }
    this.s = s; this.t = t; this.l = l;
}
```

Listing 3 The algorithm for solving the LRS problem

```
requires \invariant_for(sa);
requires sa.a.length >= 2;
ensures 0 <= s && s < sa.a.length;
ensures 0 <= t && t < sa.a.length;
ensures 0 <= l && l < sa.a.length;
ensures s+l <= sa.a.length;
ensures t+l <= sa.a.length;
ensures (\forall int j; 0 <= j && j < l;
                          sa.a[s+j] == sa.a[t+j]);
ensures s != t || l == 0;
ensures !(\exists int i,k; 0 <= i && i < k
            && k < sa.a.length-l;
            (\forall int j; 0 <= j && j <= l;
                sa.a[k+j] == sa.a[i+j]));
```

Listing 4 Contract for LRS

original array while the implementation iterates through a suffix array. This means that a suitable loop invariant is not just an abstraction from the method's post-condition. We use the following loop invariant, which states that there is no suffix indexed by w up to the current loop index x such that $w - 1$ and w point to a repeated substring of length $\ell + 1$:

```
!(\exists int w; 0 < w && w < x
    && sa.suffixes[w-1] < sa.a.length-l
    && sa.suffixes[w]   < sa.a.length-l;
      (\forall int j; 0 <= j && j <= l;
           sa.a[sa.suffixes[w-1]+j]
        == sa.a[sa.suffixes[w]  +j] ));
```

Interaction was both required in proving that the invariant is preserved and that the post-condition follows from it. The preservation proof uses the already proven contract for `lcp()` (see Sect. 3.1 above) and required six quantifier instantiations, which were more or less obvious. A case distinction on whether `suffixes[x]` already points to the end of the array was also helpful.

Proving the use case, i.e., that the post-condition follows from the loop invariant in a terminal loop state, required even more guidance. First of all, we need the fact that the suffix array describes a permutation of the original array, and in particular that it is invertible. This is contained in the class invariant of `SuffixArray` (see Listing 2), the definition of which we unfold in the proof. Since we now have pointers into the suffix array, we can make use of the following lemma: if there are pointers i and j into a suffix array such that they point to a repeated substring of length k, then the same substring also appears at the position indexed by $i + 1$. A technical issue is that our specification language JML does not support the concept of lemmas. Therefore, we formulate it as a contract to a static boolean (model) method which always returns `true`, as seen in Listing 5. In the proof, we make a case distinction over the result of this method being true. One case can be closed instantly using the implementation of the method. On the other branch, we can use the lemma to prove that there are LRS of length $\ell + 1$. Apart from these case distinctions and occasional (mostly trivial)

```
/*@ public normal_behaviour
  @ requires \invariant_for(sa);
  @ requires 0 <= i && i < j && j < sa.a.length &&
  @   sa.suffixes[i] + k <= sa.a.length &&
  @   sa.suffixes[j] + k <= sa.a.length &&
  @   (\forall int t; 0 <=t && t < k;
  @       sa.a[sa.suffixes[i]+t]
  @     == sa.a[sa.suffixes[j]+t] );
  @ ensures sa.suffixes[i+1] + k <= sa.a.length &&
  @   (\forall int t; 0 <=t && t < k;
  @       sa.a[sa.suffixes[i]+t]
  @     == sa.a[sa.suffixes[i+1]+t] ); @*/
public static boolean neighbourMax(SuffixArray sa,
  int i, int j, int k) { return true; }
```

Listing 5 This lemma states that longest repeated substrings can be found among neighbours in the suffix array

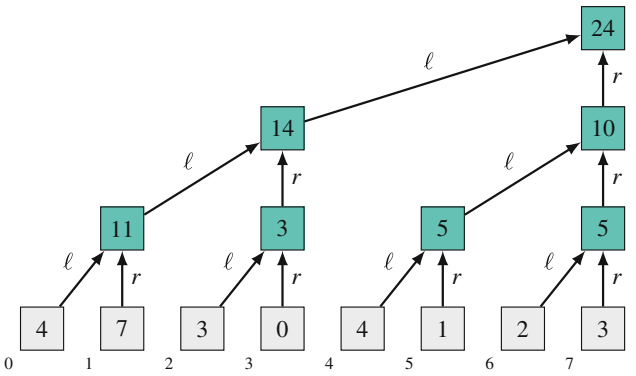

Fig. 1 An example for upsweep. If viewed in colour, the *green* cells are updated with the contents of their *left* and *right* children

quantifier instantiations, the prover ran automatically, eventually closing the proof in about 26,000 proof steps.

The introduced lemma captures the fact that a longest repeating substring will occur between neighbouring entries in the suffix array: the substrings to which the indexes in the suffix array point are lexicographically sorted; if two entries share a common prefix, this must hence be shared by all entries between these two. The longest common prefix must be between neighbours.

For the proof of the lemma, further propositions were provided as additional lemmas (also as method contracts) which captured intermediate properties like the fact that lexicographic order is transitive and reflexive.

Leaving out the maximality property would simplify the proof drastically: it would not require the above lemma or even the inversion property and could be found without user interaction in around 12 s involving 2,000 proof steps.

4 Prefix sum

In the second challenge, we worked on an intricate algorithm to compute prefix sums [4]: Given an array of integers $\{a_0, \ldots, a_{2^n}\}$, compute an array $\{s_0, \ldots, s_{2^n}\}$ such that $s_i = \sum_{j=0}^{i-1} a_j$ (every cell s_i contains the sum over a_j up to $i-1$). For example, the array $\{4, 7, 3, 0, 4, 1, 2, 3\}$ has prefix sums $\{0, 4, 11, 14, 14, 18, 19, 21\}$. While this property is quite simple, the target is an efficient divide-and-conquer algorithm, which has various applications to sorting and computer graphics.

At the competition, there was a choice between an iterative and a recursive implementation. We decided to take on a recursive implementation, because it reflects more precisely the real algorithm (which is parallel) than an iterative implementation; in addition, recursion allows us to reason more modularly about subproofs, and specifications are more concise since there is no need for annotations such as loop invariants. All verification results have been obtained off-line after the competition.

4.1 Problem description

The prefix sums algorithm works in three phases: In the first one, called *upsweep*, sums over subarrays are calculated bottom-up. Upsweep recurses for the left and the right half subarray, then these sub-results are added up and written to the right-most cell of the current subarray. Figure 1 displays upsweep graphically for the above example; the resulting array is $\{4, 11, 3, 14, 4, 5, 2, 24\}$. In the second phase, the right-most cell is set to zero. In the third phase, called *downsweep*, the respective full prefix sum is propagated to each cell. It runs top-down, beginning with the full array: For every subarray, the value of its right-most cell is copied to the right-most cell of the left half subarray (`left` index in the implementation), at the same time, the sum of values from these two cells is written to the right-most cell.

In the end, we were only able to prove a side-condition of upsweep, that on each recursive call, the right-most cell contains the sum over all cells in range. In Sect. 4.4 we explain a stronger specification for upsweep (which talks about *all* cells), which would be needed to prove correctness of the complete algorithm. Proving it with KeY, however, does not seem to be possible at the moment. This is mostly due to frequent occurrences of the exponential function and reasoning by induction is not well supported in KeY.

4.2 Specification approach

To simplify the implementation of upsweep, we factored out common arithmetic patterns into separate helper methods. In particular, we use methods `div2` and `even` to encapsulate integer division and modulo operations. This allows us to attach contracts to them. For instance, the precondition for `div2(x)` requires that `x` is a positive even number. From

```
public void upsweep(int left, int right) {
    int space = right - left;
    if (space > 1) {
        upsweep(left - div2(space), left);
        upsweep(right - div2(space), right);
    }
    a[right] = a[left] + a[right];
}
```

Listing 6 Recursive implementation of upsweep

this, it not only follows that the result is always defined, but also that it is strictly less that `x`, which we need to prove termination of the algorithm. The method `isPow2(x)` contains a recursive definition of when a positive integer `x` is a power of 2. In the contract, we have a sufficient and necessary condition given using a product comprehension, but also a more practically helpful condition which states that a power of 2 is either 1 or it is even and half of it is again a power of 2. We also use an additional lemma which states that the sum of two even numbers is even and the sum of an even and an odd number is odd.

Upsweep (see implementation in Listing 6) computes sums for prefixes of the array bottom-up. On each iteration, beginning with a two-element subarray, `upsweep(l, r)` writes the sum $\sum_{i=2\ell-r+1}^{r} a_i$ into a_r. In particular, after the upsweep phase has completed, the right-most cell contains the sum over the complete array. ℓ and r point to the right-most cell in the left and the right subarray, respectively; then $2\ell - r + 1$ points to the left-most cell.

4.3 Proving upsweep

We prove that the right-most cell contains the above given sum (see JML specification in Listing 7). This property is slightly simpler than the general case where we need to show that every cell contains a sum over a more complicated range (see Sect. 4.4 below). The two main challenges in verification are (i) proving the post-condition, which involves reasoning about sums and (ii) proving the pre-conditions of recursive calls. The principal idea of proving (i) is simply showing that the sum of the sums temporarily stored in a_ℓ and a_r is the sum of all cells in range in the pre-state; this requires a proof by induction over the length of the array. The more intricate issue here is framing. We need to show that the two recursive calls are independent of each other.[4] Lines 12ff in Listing 7 give a frame condition for upsweep under which only odd positions between $2\ell - r + 1$ (which is computed by `leftMost`) and r may be written. This means that the recursive calls may only write to $a_{2\ell-r+1}, \ldots, a_\ell$ and $a_{\ell+1}, \ldots, a_r$, respectively. We could have written that property also as a regular post-condition in the contract, but using frames with KeY's built-in data type of location sets is favoured since it adds more structure to the proof.

```
requires left < right;
requires right < a.length;
requires leftMost(left, right) >= 0;
requires isPow2(right - left);
requires !even(right);
requires !even(left) || right - left == 1;
ensures a[right] ==
    (\sum int i;
        leftMost(left, right) <= i && i < right+1;
        \old(a[i]));
measured_by right - left + a.length + 3;
assignable \infinite_union(int k;
    !even(k) && leftMost(left, right) <= k &&
        k <= right; \singleton(a[k]));
```

Listing 7 Specification of `upsweep`. The operator `\infinite_union` denotes the union over a range of sets in JML*

Since our goal is to show total correctness, i.e., including termination, we have to give *variants* (termination witnesses) to recursive methods. A variant is an integer expression which must evaluate to a non-negative number in any legal pre-state, and for recursive calls it must be strictly less than that in the caller-side scope. The variant for upsweep (given with `measured_by`) can be seen in L. 11 of the contract. The "+3" part is a technical requirement in order for other recursive methods to be called; it fixes a strict ordering of called methods and ensures that the call graph does not contain cycles (see also Sect. 6 for a discussion). For the precondition of the recursive call on the left half, it amounts to show that $r - \ell > \lfloor \frac{r-\ell}{2} \rfloor$, which is true for all r with $r > \ell + 1$.

However, to prove this we need to make case distinctions on whether $r - \ell$ is a multiple of 2. In order to have this property established for all recursive calls, we assume that $r - \ell$ is a power of 2 in the precondition. This, in turn, requires us to reason about evenness and powers of 2. Side-proofs of this kind actually did contribute a lot to the overall proof size.

We have proven upsweep (including termination) in a semi-automated proof style: we have let the automated strategy run for most of the proof steps, but stopped before applying any method contracts. Some proof cuts were made by hand to simplify formulae, e.g., the user had to indicate where the above-mentioned lemma about sums of even numbers was needed to conclude the proof. In total, the proof took about 100,000 steps, of which less than 100 were applied by hand; time consumed by automated rule application was around 18 min. Correctness proofs for the auxiliary queries and lemmas mentioned above could be found without any user interaction; altogether they took 6,000 proof steps applied in under 6 min.

[4] This is in fact essential to this algorithm being parallelisable.

4.4 Upsweep revisited

We are now going to take an outlook at the more general postcondition where we state the value of each cell. Above, we have already seen that the right-most cell a_r contains a sum over $2(r-\ell)$ values. In the general case, this can be described through a function $f : \mathbb{N} \to \mathbb{N}$ such that each cell $a_{2\ell-r+1+i}$ holds the sum over $f(i)$ elements. The general version of the postcondition thus is the following:

```
ensures (\forall int k; 0 <= k && k < 2*(right-left);
    a[k + leftMost(left, right)] == (\sum int i;
        k - f(k) + 1 <= i && i < k + 1;
            \old(a[i + leftMost(left, right)])));
```

A closed form definition (cf. [19]) of this function f is $f(i) = 2^{\min\{k\in\mathbb{N} | i \bmod 2^{k+1}=2^k-1\}}$. It can be easily implemented using a functional programming style. Writing down a contract is not trivial because the minimum is not a total function in general. Therefore, the idea is to provide an implementation of the minimum function and show that it terminates for all non-negative integer parameters. However, proving intricate specifications like these, involving recursive predicates, turns out to be still out of reach for a tool like KeY which targets imperative implementations.

5 Deletion in a tree

The third challenge is concerned with linked data structures, concretely binary search trees. The execution of the challenge program results in a modified tree structure with the minimal element removed from the tree. It is particularly important that an *iterative* algorithm was chosen instead of a recursive one. More concretely, the given verification task is the following:
Given: a pointer `t` *to the root of a non-empty binary search tree (not necessarily balanced). Verify that the procedure in Listing 8 removes the node with the minimal key from the tree.*

```
final class Tree {
  Tree left, right; int data;

  static Tree deleteMin (Tree t) {
    Tree tt, p2, p;

    p = t.left;
    if (p == null) { t = t.right; } else {
      p2 = t; tt = p.left;
      while (tt != null) {
        p2 = p; p = tt; tt = p.left;
      }
      p2.left = p.right;
    }
    return t;
  }
}
```

Listing 8 Implementation to delete the minimum element in a tree

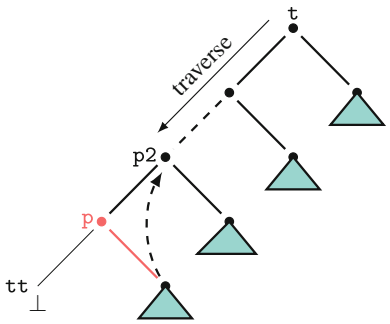

Fig. 2 Deletion of the minimal node in the binary search tree

After removal, the data structure should again be a binary search tree.

Figure 2 depicts this algorithm schematically and shows the intended result: It traverses the tree through `left` references and removes the left-most node (which contains the minimum element). The main difficulty of this challenge, as it turned out, is to maintain the information about which part of the input tree `t` actually changed. In other words, the precise frame of the changed heap locations by the one line of code that rearranges only two nodes (L. 13 in Listing 8) is very difficult to be given in terms of the initial tree `t` only. In the end, the challenge was successfully solved after the competition and required modifications of the KeY system and a non-trivial interactive proof. The following sections explain our solution in detail.

5.1 Abstracting from concrete data structures

The typical approach to address the verification of programs operating on linked data structures with KeY is to

1. define a footprint of the structure in terms of sets of locations involved in building up the structure and provide disjointness properties over this footprint to ensure that the structure is acyclic.
2. Using an abstract data type, define and link to the appropriate data elements a flat, abstract representation of the structure. Any changes to the actual data structure are reflected in the abstract representation, and any required properties, like sortedness, can be reasoned about in terms of this representation rather than the concrete data structure.

Both the footprint and the abstract representation can be declared and defined as either a model or a ghost field in JML*, and then properties can be defined in terms of a class invariant or representation clauses over these fields. For simpler verification tasks, the particular choice of whether to use ghost or model fields for these specifications is more a mat-

ter of specification style, rather than any particular reasoning capability. Specifications based entirely on model fields provide a solid separation between the specification and the code, but make proofs more difficult. Ghost fields 'intrude' the verified program by extending its state and require additional book-keeping from the specifier, but make proofs somewhat easier in practice.

However, both model and ghost fields as currently implemented in the stable version of the KeY system turned out to be a bottleneck in solving this verification challenge. The essence of a clean solution to the specification of the `while`-loop with loop invariants and then verification of the top-level specification of the `deleteMin()` method is to maintain the information about the locality of structural tree changes 'under' the node `p2` with respect to the rest of the tree, see again Fig. 2. Ideally, any properties about the tree expressed with any of the two specification methods should be parametric with respect to this *bordering* node which clearly divides the tree structure into the part which does not change (everything above the node), and the part which may still change (anything below the node). The specifications expressed with model or ghost fields would need to be defined in relation to this node, which is not fixed until the loop iteration is completed. This means that model fields should be parametric to become fully fledged abstract predicates rather than simply abstract 'prepositions'. This has led us to build our specification on the more flexible concept of *model methods*.

Model methods are a way of declaring fully functional abstract predicates in JML*. Like other specification elements, model methods are declared in designated comments and consist of

- a method signature with input parameters and the result, all of which can be of any valid Java or abstract/logical type;
- an *optional* body of the method, which gives the actual definition of the predicate;[5] a model method without a body is truly abstract and has only properties stated with pre- and postconditions;
- a precondition declaring the conditions under which the predicate is well defined and a postcondition that declares any additional properties of the predicate which follow from its definition under the assumed preconditions;
- an `accessible` clause, which declares the set of locations which the predicate at most depends on;
- a `measured_by` clause, which declares a variant for recursive calls.

[5] Currently, we only support single return statements for the body, i.e., definitions can be only given with direct formulae, rather than proper Java programs that involve, e.g., loops or similar constructs.

Moreover, all model methods are by definition strictly pure—any modification of the heap is strictly forbidden. Otherwise, there are no real limitations on expressions appearing in both the model method definition and model method specification, in particular, mutual references to other model methods are allowed. This raises an additional question of termination conditions for mutually recursive calls, something which we have not fully addressed yet; this is discussed shortly in Sect. 5.3.

5.2 Specification approach

Partitioning the tree The properties of the binary search tree structure which we need to specify are divided into two categories. The first category has to do with general properties about binary trees and defines the well-formedness criteria for trees as well as give the abstract flat representation of the tree to reason about its contents.

The second category of properties maintains the information about the current state of traversal. These properties in principle partition both the footprint and the abstract representation of the initial tree `t` between the upper part above some node `u` currently being visited and the lower part below node `u` that may still be changed by the deletion operation. The key idea is to maintain the fact that the above-footprint is strictly disjoint from the below-footprint, so that it can be ensured that changes below the current node will not affect the already visited structure above. The specifications of these partitioning predicates also ensure that inclusion of one additional left node during the traversal of the tree maintain all the required properties. Figure 3 illustrates this idea, and the following paragraphs discuss the concrete specifications.

The general structure of the binary search tree First we declare a model method which defines the overall footprint (method `fp()`) of our binary search tree and another one (method `treeInv()`) that gives the well-formedness condition to ensure that the structure does not contain cycles, i.e., that the strict footprint of the parent (i.e., all node

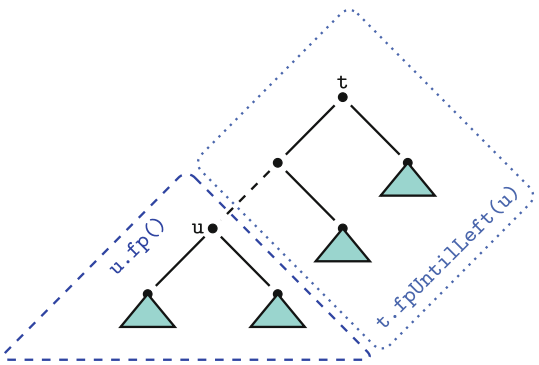

Fig. 3 Binary tree partitioning

```
ghost instance int height;

model_behavior
  requires treeInv();
  accessible fp();
  measured_by height;
  helper model \locset fp() { return \set_union(
    this.*, \set_union(
      left == null? \empty: left.fp(),
      right == null? \empty: right.fp())); }

model_behavior
  accessible fp();
  measured_by height;
  helper model boolean treeInv() { return
    height >= 0 &&
    (left != right || left==null || right==null) &&
    (left==null || (\disjoint(this.*, left.fp())
      && left.treeInv() && height > left.height)) &&
    (right==null || (\disjoint(this.*, right.fp())
      && right.treeInv() && height > right.height))
    && (left==null || right==null ||
      \disjoint(left.fp(), right.fp())); }

model_behavior
  requires treeInv();
  accessible fp();
  measured_by height;
  helper model \seq treeRep() { return \seq_concat(
    left==null ? \seq_empty: left.treeRep(),
    \seq_concat( \seq_singleton(this),
      right==null ? \seq_empty: right.treeRep())); }
```

Listing 9 The tree's structure defined through model methods

fields) is recursively disjoint from the footprints of the leaves, which are moreover mutually disjoint themselves. Both of these methods make use of the built-in `\locset` data type for location sets. Moreover, the well-formedness condition also states an invariant for the ghost field `height` tracking the height of the tree below the current node. This field is globally used in all specifications as a recursion measure. The property of the `height` field is purposely an over-approximation so that later when the nodes are rearranged during the deletion no updates to `height` are necessary.[6] Finally, we define an abstract representation of the tree contents (method `treeRep()`) in terms of object sequences, which use the built-in logical data type `\seq`. Since the tree is sorted, the most convenient way to organise the representation sequence is the inorder traversal of tree elements: the representation of left sub-tree is placed in the sequence first, then the actual node, and then the right representation. These practically generic binary search tree specifications given with model methods for the `Tree` class are given in Listing 9.

Two points of interest in these specifications are the following: The method `treeInv()` that essentially defines

[6] This would not have been the case if we had used a ghost field to track the *exact* size of the tree by precisely adding sub-tree sizes.

```
model_behavior
  requires treeInvUntilLeft(u) && u.treeInv();
  accessible fpUntilLeft(u);
  measured_by height;
  helper model \locset fpUntilLeft(Tree u) {
    return (this == u ? \empty : \set_union(
      this.*, \set_union(
        (left==null) ? \empty : left.fpUntilLeft(u),
        (right==null) ? \empty : right.fp()))); }

model_behavior
  accessible fpUntilLeft(u);
  measured_by height;
  helper model boolean treeInvUntilLeft(Tree u) {
    return this == u || (height >= 0 &&
      (left!=right || left==null || right==null) &&
      (right == null || (height > right.height &&
        right.treeInv() &&
        \disjoint(this.*, right.fp()))) &&
      (left == null || (height > left.height &&
        left.treeInvUntilLeft(u) &&
        \disjoint(this.*, left.fpUntilLeft(u)))) &&
      (\disjoint(left.fpUntilLeft(u), right.fp())
        || left == null || right == null)); }
```

Listing 10 Describing disjointness of footprints above and below node `u`

the acyclicity property is guarding the definitions of all other model methods so far. Then, even though the problem description talks about binary search trees, i.e., ones with a sorted structure, the corresponding property is purposely left out. The representation method `treeRep()` defines only a flat sequence of tree nodes with arbitrary ordering. Our top-level specification will state that this representation loses only the first element as a result of calling `deleteMin()`. That is, a more general property is established, namely that the remaining elements preserve their order and in particular they remain sorted when the initial tree is also sorted.

Invariant tree properties As already explained, to specify and prove correct the `deleteMin()` method it is crucial to keep the tree structure above the currently visited node separate from the structure still to be searched, as shown in Fig. 3. Thus, we define a similar set of model methods as above, but this time all of them are parametrised with a limiting node `u` until which the given property is checked. We start with model methods `fpUntilLeft(u)` and `treeInvUntilLeft(u)` as shown in Listing 10.

These methods define the footprint of the tree from the current `this` node until some node `u` which eventually appears in the left sub-tree. The footprint of the `u` node itself is excluded from the result. This is achieved in the base case of the definition when the currently visited node `this` is the same as the bordering node `u`, in which case the resulting footprint becomes empty. Otherwise, the definition progresses by adding the right footprint (if there is one) of the current node to the footprint collected so far and, if possible, recursively stepping into the left sub-tree of the current node.

The `fpUntilLeft(u)` model method is only defined for well-structured trees, this is stated in the precondition.

We define an analogous model method to build the representation of the tree until the `u` node:

```
requires treeInvUntilLeft(u) && u.treeInv();
accessible fpUntilLeft(u);
measured_by height;
helper model \seq treeRepUntilLeft(Tree u) { return
 (this == u ? \seq_empty : \seq_concat(
   left==null ? \seq_empty : left.treeRepUntilLeft(u),
   \seq_concat(\seq_singleton(this),
      right==null ? \seq_empty : right.treeRep()))); }
```

This definition is very similar to the definition of the above footprint method and also strictly excludes the representation of the node `u` from the result.

The essential property of all these `u`-limited model methods is that they only depend on the location set defined by `fpUntilLeft(u)`. In particular, evaluation of these methods should not depend on the footprint of `u`, but this is only true when the node `u` is actually in the left sub-tree.

The whole trick in our specification and the overall approach to the proof is to reverse this relation. That is, given that the left-footprint of the current tree until node `u` and complete footprint of the node `u` we search for are disjoint, we can check whether node `u` is a left subtree, and, if so, several further properties hold. The model method `leftSubTree` establishes whether a given node `u` is a sub-tree on the left side of the current tree under the mentioned conditions, with the following definition:

```
requires treeInvUntilLeft(u) && u.treeInv();
requires \disjoint(fpUntilLeft(u), u.fp());
ensures (* see below *);
accessible fpUntilLeft(u);
measured_by height;
helper model boolean leftSubTree(Tree u) { return
 (this==u || (left != null && left.leftSubTree(u))); }
```

The preconditions express what we have just stated and the well-formedness of the two trees involved, and similarly to other predicates, the recursion depth of method `leftSubTree` is bounded by the height of the tree. Again, it is important that this predicate only depends on the values of locations until the node `u`—once node `u` is reached, the search stops.

Given that the `u` node is indeed a left sub-node of the current tree several properties hold that are useful for the problem at hand. These properties are expressed through postconditions of `leftSubTree`. We want to state the following facts:

- If the left link of node `u` is not `null`, then the next left node is also a left sub-tree of the current tree:

```
ensures \result ==>
  (u.left == null || leftSubTree(u.left));
```

- The abstract representation of the current tree is a concatenation of the representation of the sub-node `u` with the partial representation starting from the current node until `u`:

```
ensures \result ==> treeRep() == \seq_concat(
  u.treeRep(), treeRepUntilLeft(u));
```

- Similarly, the footprint of the current tree consists of the footprint of the sub-tree rooted in `u` and the partial footprint from the current node until `u`:

```
ensures \result ==>
  fp() == \set_union(fpUntilLeft(u), u.fp());
```

  Note that the two footprints which build up the complete footprint of the current tree are guaranteed to be disjoint by the precondition of `leftSubTree`.
- Establishing the overall well-formedness of the current node is equivalent to checking the well-formedness until the limiting node `u` and the well-formedness of the node `u` itself:

```
ensures \result ==> (treeInv() <==>
  (treeInvUntilLeft(u) && u.treeInv()));
```

- When stepping one non-`null` node to the left from `u`, the partial tree representation until this child node becomes a concatenation of (1) the node `u`, (2) the representation of the right sub-tree of `u`, and (3) the previous partial representation until node `u`:

```
ensures \result ==> (u.left == null ||
  (treeRepUntilLeft(u.left) == \seq_concat(
    \seq_singleton(u),
    \seq_concat(u.right == null ? \seq_empty :
      u.right.treeRep(), treeRepUntilLeft(u)))));
```

  In other words, with this postcondition we describe how the partial representation is extended during traversal of the tree through `left` links.
- Similarly, we state what happens to the partial footprint when stepping one node to the left from `u`. The resulting extension of the partial footprint is entirely analogous to the extension of the partial representation:

```
ensures \result ==> (u.left == null ||
  (fpUntilLeft(u.left) ==
  \set_union(u.*, \set_union(u.right == null ?
     \empty : u.right.fp(), fpUntilLeft(u)))));
```

Deletion of the minimal element The main purpose of the `leftSubTree` method is to maintain the information about how the properties defined with the other model methods are affected and which part of the tree they (do not) depend on during a single step of the tree traversal. Since most of the 'property weight' is delegated to the `leftSubTree` method, the specification of the main `while` loop of the deletion method `deleteMin()` only needs to maintain that the node `p2` keeps being in the left sub-tree of `t` and that the preconditions of `leftSubTree` are still satisfied. Together

```
/*@ maintaining p != null && p2 != null;
  @ maintaining t.treeInv() && p2.treeInv();
  @ maintaining t.treeInvUntilLeft(p2);
  @ maintaining p2.left == p && p.left == tt;
  @ maintaining t.leftSubTree(p2);
  @ maintaining
  @       \disjoint(t.fpUntilLeft(p2), p2.fp());
  @ decreasing tt == null ? 0 : tt.height;
  @ assignable \strictly_nothing; @*/
while (tt != null) { p2 = p; p = tt; tt = p.left; }
```

Listing 11 Traversal loop in `deleteMin` with loop specifications

with a couple of expressions which define the dependencies between nodes `p`, `p2`, and `t`, this makes up the loop specification in Listing 11. The variant of the loop specified in L. 8 is self-explanatory. Further, with the `assignable` clause in L. 9 we specify that the loop does not modify the heap memory—all the modified variables in the loop (`p`, `p2`, `tt`) are local. Given the definitions of the corresponding model methods, it should be easy to see that these loop invariants are indeed maintained.

The actual modification only happens after the loop. Our top-level specification for the `deleteMin()` method states that the rearrangement of nodes removes only the left-most node and that the resulting tree is still well formed. The special case in which the input tree contained only one node is specified separately. The result of removing the left-most node from the abstract representation of the tree is expressed using existential quantification over the removed node. From the point of view of this top-level method the `assignable` clause can be only over-approximated with the whole footprint of the input tree `t`:

```
  @ requires t.treeInv();
  @ ensures \result == null ==>
  @     \old(t.treeRep() == \seq_singleton(t));
  @ ensures \result != null ==> (\result.treeInv() &&
  @     (\exists Tree p; \old(t.treeRep()) ==
  @          \seq_concat(\seq_singleton(p),
  @          \result.treeRep())));
  @ assignable t.fp(); @*/
static /*@ nullable @*/ Tree deleteMin(Tree t) { ... }
```

5.3 Proving correctness

The complete correctness proof for `deleteMin()` entails showing the following proof obligations to hold. For each of the declared and specified model methods we have to show that

- The method implementation (body) respects the `accessible` clause, i.e., show the method's dependency contract to be correct.
- Given its definition, the method fulfils the specified contract, i.e., show the method's properties to hold.

For `deleteMin()` we need to show that it fulfils its top-level specification, which in turn entails showing that the main `while` loop maintains the specified loop invariants, and that the loop invariants upon the exit of the loop are sufficient to establish the final properties after the tree nodes are rearranged. Obviously, it is possible to use all already proven properties and contracts during any of the proofs as long as no circular proof dependencies are introduced. In particular, for the recursive model methods it is important that contracts are applied only on strictly smaller sub-trees with respect to the initial proof obligation.

Most of the proofs with KeY for this challenge were done manually by careful application of the proof rules in the interactive mode of KeY. The technical reasons for being forced into the manual mode have to do with the experimental character of the model methods extension that we introduced. In particular, we did not yet implement proper control of progression over recursive application of contracts to avoid circular dependencies in the proof and consequently establish termination. The formerly existing control in KeY over such calls turned out to be too restrictive; in particular, it practically prohibits complete reasoning about mutually recursive references in model methods.

Apart from these technical aspects, subject to future improvements we need to introduce into KeY, the correctness proof required actual proof guidance in terms of providing cut formulae to direct further reasoning, as follows:

The two most difficult parts are proving the contract for the `leftSubTree` method to be correct, i.e., the properties which continue to hold during traversal of the tree and proving the top-level property after the node rearrangement. In particular, this main proof produces a Dynamic Logic formula which states a relationship between the original tree representation and the new representation after the heap is modified (and a similar formula for `treeInv` that is also part of the postcondition):

$$\exists_{p:\mathtt{Tree}} \mathrm{treeRep}(\mathtt{heap}) \doteq [p] \oplus \mathrm{treeRep}(\mathtt{updatedHeap}),$$

where the operators $[\cdot]$ and $\oplus$ denote singleton sequence and sequence concatenation, respectively, and the updated heap is constructed from the original heap by updating the `p2.left` node:

$$\mathtt{updatedHeap} \doteq \mathtt{heap}[\mathtt{p2.left} \mapsto \mathtt{heap}(\mathtt{p.right})]$$

The witness for the existential quantifier is produced by the last loop iteration. Thus, a simple instantiation with the node `p` is sufficient.

The most essential part of the proof is to perform a cut over the `t.leftSubTree(p2)` to establish that it is actually equal in the two states before and after the node rearrangement, i.e.,

leftSubTree(`heap, p2`) $\doteq$ leftSubTree (`updatedHeap,p2`)

This is relatively easy to show under the disjointness conditions between `t.fpUntilLeft(p2)` and `p2.fp()` carried over across the loop. That is, it is at this point that we use the fact that the parts of the trees below and above the `p2` are strictly disjoint in terms of footprints and hence make the corresponding properties which refer to the *upper* part of tree equivalent between the old `heap` and the changed `updatedHeap`. In KeY, the so-called dependency contracts generated from the accessibility clauses are used to establish such equivalences [26]. Then, from the contract of `t.leftSubTree(p2)` we extract several other properties that show equivalences between these two heaps. In particular, for proving the tree representation property we get the fact that the representation above the node `p2` is not changed between `heap` and `updatedHeap`.

Although the principles of the proof we just described are not that complex, the interactions imposed by KeY made the proof substantially complex. Whenever possible, i.e., for the remaining parts of the proof where no more cuts or contract applications were necessary, we still employed the automatic mode of KeY. Regardless of that, the proof required slightly over 1,200 manual proof rule applications and a total of 133,000 automatic rule applications to finish the proof. The final proof took us half a day of work with automated mode running time of 6 min.

6 Conclusion and outlook

Competitions such as the present put under test both the abilities of the competitors as well as their verification tool in a very controlled setting. On-site competitions, in addition, offer the possibility for an immediate comparison of approaches and results with the other teams, however, considering the time pressure this is usually done to a very limited extent. Thus, we consider this paper to be a contribution to the other competing teams to study and analyse our solutions to the competition challenges in more depth.

To the authors, the most important aspect of the competition is the evaluation of the KeY verification system. The inability to solve the *Prefix Sum* and *Tree Deletion* challenges within the competition time triggered efforts to improve and add new features to the system. Apart from helping to solve the competition challenges, most of these new features made it to the freshly released stable version 2.0 of KeY:

- JML model methods have been added to KeY as a means for abstraction from the concrete implementation. This makes the KeY system the first JML-based tool to fully support this concept in interactive verification.
- It is now possible to inductively reason about integer product comprehensions.
- In addition to skolemisation, KeY offers automated induction over quantified formulae over the integers.
- Strategy macros (see Sect. 2.3) have been introduced to allow a more high-level user interaction thus significantly reducing the total number of required interactive steps.

The most difficult part in software verification usually is to find appropriate specifications. The ability to conduct proofs by symbolic execution interactively turned out to be advantageous in this regard. Symbolic execution directly provides human-readable feedback on program control flow—similar to software debugging. Interactive proving furthermore allows users to make sensible case distinctions. This does not only provide some intuition on whether the used specifications are correct or sufficient, but also on how specification and implementation could be refactored to simplify the proof which helped us in all three challenges.

The challenges clearly point out directions for further improvement. In particular, proving well-foundness of mutually recursive methods, both regular and model, is yet to be addressed. The currently implemented mechanism in KeY essentially requires all methods to share a common integer recursion variant given by the `measured_by` clause. In practice this is very restrictive. One of the possible solutions to this problem would be to allow more structured variant expressions, like tuples of integers which are then compared lexicographically. A pair can be used, e.g., where the first value indicates the dependency depth of mutual calls and the second one denotes the actual recursion depth. Another possibility would be to manually restrict the set of applicable method contracts in a particular verification condition, e.g. by specifying (and checking) JML `callable` clauses.

On a more general level, the competition challenges emphasised the gap between recursive and iterative programs, both in terms of actual implementations and the verification effort. This gap is not always obvious to predict. For the *Tree Deletion* challenge we initially expected the iterative nature of the algorithm over a recursive data structure to be the source of difficulties. Although, judging from a similar previous challenge [5], we still think that the recursive implementation would be simpler to verify; with the help of model methods we employed the recursion on the specification level and provided an elegant solution to this challenge.

The last point to discuss is the performance of verification tools in the scope of competition-style challenges versus realistic programs. The research efforts in KeY have been concentrated on fully supporting concrete Java features, like inheritance, static initialisation, the use of API libraries, and to support specification features important for production software, like information-flow properties for security-critical applications [21]. In the past, this approach to the construction of the

KeY verifier enabled us to successfully verify seriously sized (for a verification effort) case studies, see e.g. [18,22], and KeY was a pioneer in fully supporting a real Java execution environment, the Java Card platform [17].

It seems that for competition style challenges, which are essentially purely algorithmic problems, what is required of the verification tool is a strong support for abstraction decoupling conceptual problems from the implementation issues of the realisation in a programming language. Such an abstraction mechanism is inherent to some verification approaches (for instance by using abstraction predicates in separation logic oriented tools like VeriFast [12]).

By introducing model methods to KeY we strive at closing this gap for our tool. But it would also have to include convenient facilities to declare abstract data types and to provide means of (inductively) reasoning about them. We believe that developing the tool and methodology towards either end of the application spectrum requires different effort and priorities, which are often difficult to bring together.

Acknowledgments The work of Daniel Bruns is supported by the German National Science Foundation (DFG) under project "Program-level Specification and Deductive Verification of Security Properties" within priority programme 1496 "Reliably Secure Software Systems—RS^3". Wojciech Mostowski is supported by European Research Council (ERC) grant 258405 for the VerCors project.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), International Workshop, 2004, Revised Selected Papers. LNCS, vol. 3362, pp. 49–69. Springer (2005)
2. Beckert, B., Grebing, S.: Evaluating the usability of interactive verification systems. In: Klebanov, V., Beckert, B., Biere, A., Sutcliffe, G. (eds.) 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE) 2012, CEUR Workshop Proceedings, vol. 873 (2012)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer (2007)
4. Blelloch, G.E.: Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (1990)
5. Bormer, T., Brockschmidt, M., Distefano, D., Ernst, G., Filliâtre, J.C., Grigore, R., Huisman, M., Klebanov, V., Marché, C., Monahan, R., Mostowski, W., Polikarpova, N., Scheben, C., Schellhorn, G., Tofan, B., Tschannen, J., Ulbrich, M.: The COST IC0701 verification competition 2011. In: Beckert, B., Damiani, F., Gurov, D. (eds.) Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011). LNCS, vol. 7421, pp. 3–21. Springer (2012)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Arts, T., Fokkink, W. (eds.) Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03). ENTCS, vol. 80, pp. 73–89. Elsevier (2003)
7. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**(6), 583–599 (2005)
8. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
9. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification. LNCS, vol. 4590. Springer (2007)
10. Filliâtre, J.C., Paskevich, A., Stump, A.: The 2nd verified software competition: experience report. In: Beckert, B., Biere, A., Klebanov, V., Sutcliffe, G. (eds.) 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE) 2012, CEUR Workshop Proceedings, vol. 873 (2012)
11. Harel, D.: Dynamic logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic, pp. 497–604. D. Reidel Publishing Co., Dordrecht (1984)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods—Third International Symposium, 2011. LNCS, vol. 6617, pp. 41–55. Springer (2011)
13. Kassios, I.T.: The dynamic frames theory. Form. Asp. Comput. **23**(3), 267–288 (2011)
14. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: an experience report. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software Theories Tools Experiments (VSTTE) 2012. LNCS, vol. 7152, pp. 196–208. Springer (2012)
15. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: experience report. In: Butler, M., Schulte, W. (eds.) Proceedings, 17th International Symposium on Formal Methods (FM) 2011. LNCS, vol. 6664, pp. 154–168. Springer (2011)
16. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT **31**(3), 1–38 (2006)
17. Mostowski, W.: Formal reasoning about non-atomic Java Card methods in dynamic logic. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Proceedings, Formal Methods (FM) 2006. LNCS, vol. 4085, pp. 444–459. Springer (2006)
18. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B. (ed.) 4th International Verification Workshop, CEUR WS, vol. 259 (2007)
19. The on-line encyclopedia of integer sequences. http://oeis.org/A006519/. Retrieved 30/07/2013
20. Parkinson, M., Bierman, G.: Separation logic and abstraction. SIGPLAN Notes **40**(1), 247–258 (2005)
21. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: Formal Verification of Object-Oriented Software. LNCS, vol. 7421, pp. 232–249. Springer (2012)
22. Schmitt, P.H., Tonin, I.: Verifying the Mondex case study. In: Hinchey, M., Margaria, T. (eds.) Proceedings of the 5th IEEE International Conference on Software Engineeging and Formal Methods (SEFM), pp. 47–56. IEEE Press (2007)
23. Schulte, W., Songtao, X., Smans, J., Piessens, F.: A glimpse of a verifying C compiler. In: C/C++ Verification Workshop (2007). https://lirias.kuleuven.be/handle/123456789/146853
24. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Addison-Wesley, Reading (2011)

25. Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer (2004)
26. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)