Specifying a Linked Data Structure in JML for Formal Verification and Runtime Checking

Christoph Gladisch[1] and Shmuel Tyszberowicz[2]

[1] Institute for Theoretical Informatics, Karlsruhe Institute of Technology (KIT)
[2] School of Computer Science, The Academic College Tel Aviv-Yaffo

Abstract. We show how to write a concise and elegant specification of a linearly linked data structure that is applicable for both verification and runtime checking. A specification of linked lists is given as an example. The concept of a list is captured by an observer method which is a functional version of a reachability predicate. The specification is written in the Java Modeling Language (JML) and does not require extensions of that language. This paper addresses a mixed audience of users and developers in the fields of formal verification, runtime checking, and specification language design. We provide an in-depth description of the proposed specification and analyze its implications both for verification and for runtime checking. Based on this analysis we have developed verification techniques that fully automate the verification process, using the KeY tool, and that are also described here.

1 Introduction

Linked data structures have been specified and verified in many works. Yet, the specifications we found in the literature either are complex and therefore difficult to understand by engineers or use logics and formulas which cannot be employed by runtime checkers[1] for popular languages such as Java. JML [11] is a specification language that has been designed for verification and runtime checking, but the language is used differently depending on which of the two approaches is used. This often results in specifications that are incompatible for the other approach. L. du Bousquet et al. [7] show that specifications used for verification or for runtime checking, even if written in JML, often cannot be exchanged for the other purpose. The combination of both approaches is, however, important due to their complementary strengths (see, e.g., [7,18]).

We have developed JML specifications for a selection of methods that operate on linked lists (`get`, `size`, `acyclic`, `remove`, `insert`). They are compatible with deductive program verification on the actual source code level as well as with runtime checking tools. For the verification we have used KeY [3] and for runtime checking the testing tool JET [5]. Both tools use JML as the specification language. The goal of the paper, however, is not only to provide ready-to-use

[1] We use the term *runtime checker* as a synonym for *testing tool*. The term is motivated by the runtime assertion checker (RAC) that is provided with JML [11].

J. Iyoda and L. de Moura (Eds.): SBMF 2013, LNCS 8195, pp. 99–114, 2013.
© Springer-Verlag Berlin Heidelberg 2013

specifications, but to explain the design decisions with respect to verification and runtime checking. Our goal is to explain to engineers that use runtime checkers how to write specifications that are compatible with formal verification tools and vice-versa.

To achieve readable and executable specifications we have decided to use queries, also known as inspector or observer methods, instead of list abstractions using ghost fields. Since no ghost state has to be managed, (a) the implementation can be executed as it is, without the need to extend it with code that updates the ghost state in parallel to the normal execution, and (b) the user does not have to think about and to specify two kinds of states. However, regarding verification, reasoning with queries is not easy and has been even proposed as a verification challenge [12]. During this research we created experiments with over 5.000 LOC as steps towards a clear and automatically provable specification. A great amount of work was to extend the proving techniques of the KeY tool, as briefly described in Section 6. Specification readability and clear semantics of the specification elements are crucial for ensuring correctness. We have developed specifications that are readable and understandable also by software engineers that are not experts in deductive verification.

Reachability is crucial for reasoning about linked data structures [14]. To illustrate our approach, we specify the query method `Node get(Node o, int n)` (Figure 2) which provides access to the `n`'th node of the list starting at node `o`, following the field `next`. It can be seen as a functional version of a reachability predicate but additionally it identifies the position of list nodes. Quantification over the integer `n` (second parameter) results in quantification over all elements of the list `o` (first parameter). This enables to express properties that involve transitive closure of the list, that a requirement holds for all elements of the list or in a specific range, and that an element exists (is reachable) which fulfills a certain property. Transitive closure and reachability cannot be expressed in first-order logic, but they can be expressed in first-order logic with integers [4].

JML provides the reachability predicate `\reach`, which returns the set of objects reachable from a particular reference. Dealing with this predicate requires reasoning about sets, something that we tried to avoid in order to reduce complexity. Not all tools that use JML as a specification language fully support this predicate, e.g. KeY and JET. Also, sometimes different semantics of the predicate are needed [1]. In contrast, the semantics of `get` is given by its specification or implementation, providing an easy way of *exporting* the semantics to various tools. Using a self-defined method instead of a built-in function or predicate is also more flexible for the user.

Structure of the Paper. Section 2 describes related work. A short introduction to JML is given in Section 3. In Section 4 the query `get` is described which is the basis for our specifications. Section 5 describes the specification of modifier methods, i.e., methods that change the program state, as well as additional queries. Section 6 describes verification techniques we have developed, experience with runtime checking, and additional insights. Section 7 concludes the paper and describes future work.

2 Related Work

Specification Using Queries and/or Model Fields. linked list specifications mostly either (a) describe the effect of mutator methods in terms of query methods, or (b) use an abstraction of the concrete data structure implementation.

The usage of inspector methods within specifications to abstract away from the concrete implementation is promoted by [8]. An explicit heap encoding limits the information on which those methods depend. In [6], a formalization of pure methods is presented that allows reasoning about method calls in JML specifications. Pure methods are encoded by uninterpreted function symbols and axioms. The encoding can be applied to JML's model fields, specification-only fields that encode abstractions of the concrete state of a data structure.

A full JML specification for `java.util.LinkedList` can be found in [2]. It has complex dependencies due to its hierarchy of containers but it hides implementation details. Our focus is different, it is on proving and testing the actual implementation of a linked data structure. Ideas from both specifications can be combined. Some technical differences are: they use model fields, we do not; they use no recursive specification of the `get` method and it is not connected with a "next" pointer; their specification of `remove` uses disjunctions (DNF versus CNF) which is incompatible with our verification technique (item 3 in Section 6).

Dafny is used to specify and verify a linked list in [13]. The class node uses two ghost fields: the sequence of data values stored in a node and its successors, and a set consisting of the node and its successors. In contrast to our approach, the specifications do not use any reachability predicate.

Specification Using FOL. Analysis of programs that manipulate linked lists by using first-order axiomatizations of reachability information has been extensively studied (e.g., [16,10]). The verification in [10] provides a first-order approximation of a reachability predicate. Two predicates characterize reachability of heap cells. These predicates allow reasoning uniformly about both acyclic and cyclic lists. While theoretically incomplete, the authors of [10] believe that the approach is complete enough for most realistic programs.

In [14], the authors explore how to harness existing theorem provers for first-order logic to prove reachability properties of programs that manipulate dynamically allocated data structures. The paper also provides a set of axioms for characterizing the reachability predicate, which works only for acyclic lists.

Two abstractions, a predicate abstraction and a canonical abstraction, of a (cyclic) singly-linked list are studied in [15]. The state of a program is represented using a 3-valued FOL structure. The intuition is that a heap containing only singly-linked lists is characterized by the connectivity relations between a set of nodes and the length of list segments.

Specification Using HOL or Separation Logic. Zee et al. [20] verify full functional correctness of linked data structure implementations. The correctness properties include intractable constructs such as quantifiers and transitive closure. The specification is written in higher-order logic (including set comprehension, λ-expressions, transitive closure, cardinality of finite sets, etc.), and for verification

Java + JML

```
1  public class Node {
2   //@ public model static JMLDataGroup footprint;
3   public /*@ nullable */ Node next;  //@ in footprint;
4  ... }
```

Java + JML

Fig. 1. The class `Node`, representing list elements

the Jahob system was used. For some verifications (e.g., a sized list), additional provers such as SPASS, MONA, and BAPA have been used by Jahob.

Separation logic, a generalization of Hoare logic, is powerful for handling the framing problem which occurs with reasoning about heaps. In [9], linked lists with views are investigated which is not immediately expressible in frameworks such as JML. Separation logic is usually used for verification but it has also been utilized for runtime checking [17]. An approach that combines separation logic and dynamic frames is described in [19].

3 JML

Java Modeling Language (JML) is a behavioral interface specification language used to specify the behavior of Java modules. Following is a short description of JML clauses used in the paper. Full details can be found in [11].

The pre-state of a method call is the state of the program after passing parameters and before running the method's code. The post-state is the state of the program just before the method normally returns or throws an exception.

The `public normal_behavior` clause is used to specify behavior of method calls that return normally. The `requires` clause specifies the method's precondition, evaluated at the pre-state of the method call. The `ensures` clause specifies properties that are guaranteed to hold at the end of the method call, in case that the method returns normally. Two keywords that are used in `ensures` are `\old` and `\result`. The first refers to the value of fields at the pre-state, and the second is the value returned by the method when normally terminating. The expression `(\forall int i;`ϕ`;`ψ`)` denotes the formula $\forall i : int.(\phi \rightarrow \psi)$.

The clauses `assignable` and `accessible` declare the frame properties of a method. The former defines which (memory) locations can be updated during method execution and the latter states the locations that the method may read from. A set of locations can be declared using a model field of class `JMLDataGroup`. For instance, the model field `footprint` in Lines 2-3 of Figure 1 denotes the location set of the field `next` for all receiver objects of class `Node`. The empty set is denoted as `\nothing`.

The `measured_by` clause is used when the specification is recursive. It enables to describe a termination argument, ensuring that the specification is well-defined. It defines an integer-valued expression that must always be at least zero and it has to decrease strictly for each (recursive) call.

Java + JML

```
1  /*@ public normal_behavior
2   requires   n>=0;
3   assignable \nothing;
4   accessible Node.footprint;
5   ensures (o==null || n==0) ==> \result == o;
6   ensures            n>0   ==> \result == (get(o,n-1)!=null?
7                                              get(o,n-1).next : null);
8   measured_by n;
9  @*/
10 public static /*@nullable pure*/ Node get(/*@nullable*/Node o, int n){
11   int i=0; Node oldo = o;    //oldo is a temporary variable
12    /*@ loop_invariant 0<=i && i<=n && o == get(oldo,i);
13        assignable \nothing;  //syntactically not supported by JET
14        decreases n-i; @*/
15    while(i<n && o!=null) {
16      o=o.next;
17      i++;
18    }
19   return o;
20 }
```

Java + JML

Fig. 2. Specification and implementation of the query method `get`

Member fields, formal parameters, and method return types are by default considered to be `non_null`. In order to enable them to have a null value, they explicitly have to be annotated with the modifier `nullable`.

4 The Observer Method `Get`

In order to express properties of a list, we use the method `Node get(Node o, int n)` (Figure 2) which provides access to the `n`'th node of the list starting at node `o`, following the field `next`. It can be seen as a functional variant of a reachability predicate, allowing quantification over list elements. The chosen signature is a functional version of a *get*-method where the usually implicit `this` pointer is made explicitly as the first argument. The rational was to allow the first element to be the `null` pointer during our experiments. However, different signatures can be used, e.g. `get(int n)`, where `o` is a field or the `this` pointer.

4.1 Specification of the `Get` Query

Figure 2 presents a recursive specification of the method `get`. Line 5 defines the base-case, where either the element at position 0 is accessed or the list is empty, i.e. `null`. Lines 6-7 define the step-case for `n>0`, with a case distinction that checks whether the element at position `n-1` is `null`. If it is not `null`, then `get(o,n)` is defined as `get(o,n-1).next`; otherwise, it is also `null`.

For modular reasoning, the framing properties of `get` must be defined in addition to the functional specification. Framing properties are important to help verification and do not have to be used for runtime checking, as the latter does not abstract the code. The `assignable` clause expresses what locations might have been modified by a method (cf. Section 3). More interesting, however, is the inverse, i.e., the locations that have not been modified. For every field `f` not mentioned in the `assignable` clause, the implicit postcondition `f==\old(f)` can be assumed. This information is important for verification to relate pre- and post-state. The method `get` is (strictly) pure, thus it does not modify the heap's state and can be used in specifications.

The result of `get` depends not only on the values of the parameters `o` and `n`, but also on the `next` field values of the `Node` objects of the list starting at node `o`. Hence, whenever an assignment to `next` has been made, the value of the method may have been changed. The difficulty in specification and verification when using observer methods is in tracking the return values of the observer methods according to the changes in states they depend on. The `accessible` clause, also called dependency clause, describes which memory locations the method depends on. The dependency clause in Line 4 of Figure 2 is an over-approximation. The state of all locations that are not mentioned in the dependency clause can be ignored when evaluating the method, which considerably simplifies verification.

4.2 Implementation of the `Get` Query

The specification of `get` can be used both for specifying and for verifying properties of a list. The first method that we have verified using `get` is the implementation of `get` itself. A recursive code is the most trivial to implement and to verify. However, we demonstrate the more interesting iterative implementation, since a loop invariant, which uses the recursively defined query `get`, has to be provided or computed. As can be seen in Figure 2, the loop invariant is very concise—which is one of our goals. The code annotations, however, bear some problems that will be described next, together with the solutions that we have successfully applied.

Required Lemma. The specification of `get` (Figure 2) implicitly implies that if i is the last element's index, then for all n, with $i < n$, `get` returns `null`. This is needed for the verification, to prove the postcondition for the case that the loop in Lines 15-18 terminates due to the condition `o==null`, and `i<n` evaluates to true. If a verification system is not able to derive this knowledge automatically, it must be provided by the user—for instance as a lemma. One possibility to do this in JML is to declare a pure void method, say `lem_getTransNull`, which contains the lemma in its postcondition (see Figure 3). A runtime checker, however, will not be able to execute the postcondition as it uses unbounded quantification. In such cases it may just ignore the postcondition. Since the lemma is needed only as a hint for the verification tool, this lack of compatibility with runtime checking is not a problem. To use it for verification, `lem_getTransNull` can be inserted into the code in Figure 2 after Line 18. An implementation that ensures `i==n` at loop exit does not need the lemma.

Java + JML

```
1 /*@ public normal_behavior
2   assignable \nothing;
3   ensures (\forall int j; 0<=j && get(o,j)==null;
4                      (\forall int k; j<k; get(o,k)==null)); @*/
5 public static void lem_getTransNull(/*@nullable */ Node node){};
```

Java + JML

Fig. 3. Encoding of a null transitivity lemma of the query method `get`

Java + JML

```
1  ... measured_by n; */
2 public /*@nullable pure*/ Node getImpl(/*@nullable*/Node o, int n){
3   int i=0;  Node oldo = o;
4   /*@ loop_invariant 0<=i && i<=n && o == get(oldo,i);  ...
```

Java + JML

Fig. 4. Implementation of the query method `get`

Well-definedness Issues: The `measured_by` Clause. The specification of the `get` query (Figure 2) contains the `measured_by n;` clause. This clause requires that each time the method is called a) the value of the argument `n` is decreased and b) $n \geq 0$. These conditions ensure the method's termination and hence its well-definedness. However, the loop invariant in Figure 2 is problematic, as it permits `i==n`. When the subformula `o==get(oldo,i)` is evaluated and `i==n`, the call `get(oldo,n)` is encountered which violates condition a) of the `measured_by` clause. This can be a problem also for runtime checking, and not only for verification: the checker may not terminate when checking the loop invariant. To enforce the first condition, the following two solutions can be applied:

Solution 1. Distinction between the Program and the Specification Function. This solution explicitly distinguishes between the `get` query, which will be used for specification only, and the method used for implementation, say `getImpl` (Figure 4). Both queries `get` and `getImpl` co-exist. The expression `n` following the `measured_by` clause of `get` is independent of that employed in `getImpl`. In order to use `get` also by a runtime checker, the implementation must be provided.

Solution 2. Expanding the Definition of `get` in the Loop Invariant. Since the second argument of `get` is decreased in each recursion step, manual expansion of the specification of `o==get(oldo,i)` ensures the satisfaction of the required conditions of the `measured_by` clause. However, the specification is larger and less readable.

5 Specification of Modifier Methods

Modifier methods, also called mutators, are non-pure ones, i.e., they can modify fields of objects. We will show two of them: `remove` and `insert`. Figure 5 shows

an implementation of the `remove` method. A trivial specification of the method is shown in Figure 6, where Line 5 of the specification describes the effect of the assignment in Line 3 of the implementation. The specification formalizes how the `next` field is changed by the method when the precondition is satisfied. The specification is strong and correct, but it is not suitable for our approach as it does not specify how the result of `get` has changed.

The problem is that, in contrast to runtime checking, in verification a query that uses recursion or a loop cannot be simply executed as this execution would not terminate for arbitrary inputs. Instead, the value of the query has to be deduced. This is not just the *framing problem* but the question of *how exactly* values have changed. The addressed problem is typical for specifications with queries used in modular verification. Handling this problem has been proposed as a verification challenge [12] and is addressed in different works (see Section 2). It can be explained using an example, summarized by the following three lines:

assume	value of `get(o,i)` is known
assign	`u.next:=b`
assert	ϕ(`get(o,i)`)

Assume that the value of `get(o,i)` is known, e.g. from a precondition. Then a reference value `b` is assigned to the field `next` of an object `u` of class `Node`. Such an assignment may occur in a modifier method, for instance `remove`. Since the field `next` has been modified and the query is heap-dependent, the return value of the query may have changed after the assignment. The problem is in determining the value of the `get(o,i)` query after the state change in order to check whether it fulfills some condition ϕ, e.g. the postcondition. In contrast to runtime checking, in modular verification the query is not executed but rather only the information from its specification is used[2]. However, whereas the value of the field `next` in Figure 6 is specified, the evaluation of the `get` query is not.

```
— Java —
1  public static void remove(Node o, int i){
2     Node n=get(o,i-1);
3     n.next=n.next.next;
4  }
— Java —
```

Fig. 5. Implementation of the modifier method `remove`

A similar problem occurs also with the other queries: `size` and `acyclic`. When verifying a program which invokes the query `remove` two times in a row, e.g. `remove(o,i);remove(o,k);`, the postcondition of the first invocation of `remove` must imply the precondition of the second invocation:

[2] Also in verification the implementation of the query can be used instead of the specification, but since the state space is infinite, or arbitrarily large, it cannot be flattened to a finite set of executions.

```
JML
1  public normal_behavior
2  requires   0<i && i<size(o) && acyclic(o);
3  assignable Node.footprint; //for KeY: get(o,i-1).next;
4  accessible Node.footprint;
5  ensures    \old(get(o,i-1)).next==\old(get(o,i+1));
                                                              JML
```

Fig. 6. A precise specification of the method `remove`

```
JML
1 /*@ public normal_behavior
2  requires 0<i && i<size(o) && acyclic(o);
3  assignable Node.footprint; //for KeY: get(o,i-1).next;
4  accessible Node.footprint;
5  ensures  (\forall int j;0<=j && j<i; get(o,j)==\old(get(o,j)));
6  ensures  (\forall int k;i<k && k<=\old(size(o));
7                                  get(o,k)==\old(get(o,k+1)));
8  ensures  size(o) == \old(size(o))-1 && acyclic(o);          @*/
                                                              JML
```

Fig. 7. Specification of the method `remove` using the query `get`

$$\ldots \overbrace{\texttt{\textbackslash old}(\texttt{get}(o, i-1)).\texttt{next} = \texttt{\textbackslash old}(\texttt{get}(o, i+1))}^{\text{postcondition of } \texttt{remove(o,i)}} \rightarrow \overbrace{0 < k \wedge k < \texttt{size}(o) \wedge \texttt{acyclic}(o)}^{\text{precondition of } \texttt{remove(o,k)}}$$

where '...' stands for additional assumptions, e.g. $0 < k < i < \texttt{size}(o)$, to ensure validity of the formula. In order to prove this formula, knowledge is needed of how the new information about the field `next` changes the evaluation of `size` and `acyclic`. This knowledge can be provided either in the form of a lemma, or, following our approach, in the postcondition of the modifier method.

5.1 Specification of the Method `Remove` Using Queries

A specification of the `remove` method that uses the `get` query is provided in Figure 7. Since the specification describes the return value of `get` after calling `remove`, one can regard it also as a specification of the query with respect to the execution of `remove`. The specification contains three postconditions. The first

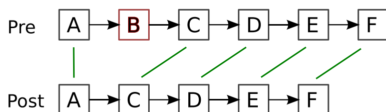

Fig. 8. Correspondence of list nodes before and after removing element B

Java + JML

```
1  /*@ public normal_behavior
2   requires 0<i && i<=size(o) && acyclic(o);
3   requires e.next==null && (\forall int i;0<=i && i<=size(o);get(o,i)!= e);
4   assignable Node.footprint;
5   accessible Node.footprint;
6   ensures (\forall int j;0<=j && j<i;get(o,j)==\old(get(o,j)));
7   ensures get(o,i) == e;
8   ensures (\forall int k; i<k && k<=\old(size(o))+2;get(o,k)==\old(get(o,k-1)));
9   ensures  size(o) == \old(size(o))+1 && acyclic(o); @*/
10 public static void insert(/*@nullable */ Node o, int i, Node e){
11  Node tmp = get(o,i-1);
12  lem_getTransNull(o); //this is a lemma, see Figure 3
13  e.next = tmp.next;
14  tmp.next = e;
15 }
```

Java + JML

Fig. 9. Specification and implementation of `insert` using queries

Java + JML

```
1 /*@ public normal_behavior
2  requires (\exists int i; ((o==null && i==0) ||
3              (i>0 && get(o,i-1)!=null && get(o,i)==null)) );
4  assignable \nothing;
5  accessible Node.footprint;
6  ensures ((o==null && \result==0) ||
7   (\result>0 && get(o,\result-1)!=null && get(o,\result)==null));  @*/
8 public static int /*@ pure */ size(/*@nullable */ Node o){...};
```

Java + JML

Fig. 10. Specification of the query method `size`

(Line 5) describes the new value of the query with respect to its old one (i.e., before executing the method) for the list interval that has not been changed. The second postcondition (Lines 6 and 7) describes the interval after the removed element; here the list has been shifted as depicted in Figure 8. These two postconditions solve the problem of proving the following assertion:

assume	value of `get(o,i)` is known
invoke	`remove(o,j)`
assert	ϕ(`get(o,i)`)

If the formula ϕ(`get(o,i)`) is true, it can be proved using the specification given in Figure 7 instead of the one in Figure 6. The reason is that full information about `get` is available after invoking `remove`.

The third postcondition (Figure 7, Line 8) specifies the return values of `size` and `acyclic` in the post state of `remove`. Hence, when calling `remove` twice in a row, sufficient knowledge is provided to the theorem prover to prove that the postcondition of the first invocation implies the precondition of the second one.

The specification of `insert` (Figure 9) follows similar principles to those of `remove`. Programs constructed with these methods can therefore be verified using the methods' contracts.

5.2 List Size and Acyclicity

The `size` query, which returns the list's length, is used in Lines 2, 6, and 8 of Figure 7. Note that the length of the list is arbitrary and not fixed, i.e., the correctness proof is valid for every length. The specification is also correct without the upper bound `\old(size(o))` of the quantification (Line 6), as `get(o,i)==null` when `i>size(o)`. Omitting the upper bound even simplifies the verification since less queries are used, the formula is smaller, less case distinctions have to be made, and quantifier instantiation—a well-known problem in theorem proving—is simpler. However, we have included the quantification bound as it is important for runtime checking tools. Such tools check the quantified formula explicitly for all elements of the quantification domain, e.g., by using a for-loop, thus they usually cannot handle unbounded quantifiers.

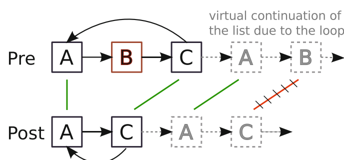

Fig. 11. Removing element B within a cycle of a cyclic list

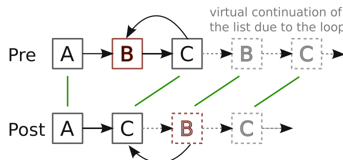

Fig. 12. Removing element B at the beginning of the cycle of a cyclic list

One way to refer to the list's size is by storing it in a field of the class `Node`. The value of this field can be defined using a class invariant and has to be explicitly updated by the methods that modify the list. This approach simplifies verification since in our approach the return value of `size` has to be deduced from the list structure every time it is used. Nevertheless, we decided to use a query in order to follow rigorously one approach.

Figure 10 shows the specification of `size` that we have used for verification. It uses neither recursion nor quantifiers. Since the query is used inside other specifications, keeping it small and simple is very important for reducing proof complexity. When using other variants to specify `size`, e.g. a recursive specification, automatic proof attempts of `remove` were more complicated or even failed. For cyclic and infinite lists the precondition of `size` is *false* and its return value is undefined. Acyclicity of the list must be ensured from the context where the query is used as it is the case for the specifications of `remove` and `insert`.

Acyclicity is required for using the modifier methods (e.g., Figure 7, Line 2). It can be implicity expressed as `\exists int i;i==size(o)`. However, using an explicit specification of acyclicity is much more efficient and practical. Figure 13 shows the specification of the query `acyclic`. A cycle exists if there are two distinct integers `i` and `j` such that `get(o,i)==get(o,j)` and `get(o,i)!=null`.

The methods `remove` and `insert` can be generalized for cyclic lists. However, then the specification and verification become more complicated. The problem occurs when removing an element within the cycle, as shown in Figure 11. When traversing the list in the pre- and poststate of `remove`, the size of the interval on which the list is shifted is increased each cycle. A solution is to redefine `size`

Java + JML
```
1  /*@ public normal_behavior
2   assignable \nothing;
3   accessible Node.footprint;
4   ensures \result == (\forall int i;0<=i && i<=size(o);
5                        (\forall int j;i<j && j<=size(o);
6                         (get(o,i)!=null==>get(o,i)!=get(o,j)))); @*/
7  public static boolean /*@ pure */ acyclic(/*@nullable*/ Node o){...};
```
Java + JML

Fig. 13. Specification of the query method `acyclic`

such that it will return the length of the list before the cycle repeats. To handle cyclic lists, also the implementation of `remove` (Figure 5) has to be changed. The reason is that if the element that is removed is the first in the cycle, then two pointers, rather than one, must be changed. Otherwise, the shape of the list may change without actually removing the element (Figure 12).

5.3 The Order of Postconditions Reflects Semantic Dependencies

The postconditions are connected via a conjunction. Nevertheless, the order of the postconditions is structured in a specific way to assist a verification tool in finding a proof. To prove one of the postconditions, but the first, in the specification of `remove` and `insert`, the preceding postcondition must be assumed as a premise. For instance, a proof of the postcondition in Lines 6-7 of Figure 7 requires the postcondition in Line 5 as an assumption. The reason is that to prove that the list has been shifted by one element after the removed element (Lines 6-7), the assumption is needed that it was not shifted on the interval before the removed element (Line 5); see also Figure 8. The postcondition in Line 8 adds another layer to the specification, which semantically depends on the postconditions in Line 5-7. It formalizes properties of `size` and `acyclic` in the poststate of `remove`. These queries are defined in terms of `get`, i.e., when replacing the queries by their postconditions, a formula is obtained that uses `get` as the only query. Hence, to prove the postcondition in Line 8, those in Lines 5-7 must be used as premises, as full information about `get` in the poststate of `remove` is needed. A similar argumentation explains also the sequential dependency of the postconditions of `insert` in Figure 9. This is a new technique, hence existing tools need to be extended, as we did in KeY, to utilize the postcondition order.

6 Verification, Runtime Checking, and Discussion

Verification. To verify the code presented in the listings we have used an extended version of KeY, a tool that enables automatic and interactive verification. Some techniques used by it are: symbolic execution of Java programs, handling of pointer aliasing, first-order theorem proving with quantifier handling via E-matching, and reasoning with integer arithmetics. It also allows applying an induction rule interactively by supplying an induction hypothesis. Such features, or equivalent ones, are needed for the verification of the code.

To achieve fully automatic verification of the presented code we have investigated the verification conditions that arose and developed techniques that increase KeY's power by several orders of magnitude for programs with recursive specifications and queries. I.e., each improvement eliminated a big set of user interactions of a certain category that were needed. A detailed description of these techniques cannot be given in this paper due to lack of space. We briefly point out three techniques that we have developed as a result of this research:

(1) *A set of strategies for replacing occurrences of queries in verification conditions by their definitions, i.e. by their pre- and postconditions.* Originally, KeY performed such replacements randomly, but for handling recursive queries such as `get` well-designed strategies are needed. Since the query `get` is specified recursively, the effect of the query expansion is that the second argument of the query is subtracted by one. Performing such a replacement is required in order to prove equality between terms. For instance, in order to prove $\Phi \rightarrow get(o,i) = get(o,i+1)$, where Φ contains some assumptions that are not shown here, it may be necessary to apply query expansion to the term $get(o,i{+}1)$, i.e., to get a term with $get(o,(i+1)-1)$ that will match the term $get(o,i)$. Originally KeY has chosen randomly which queries to expand but this approach did not lead to successful proofs. We have developed several query expansion heuristics which improved verification also of other kinds of programs than those described in this paper. The following three query expansion heuristics are required: expansion of queries after execution of the loop body; breadth-first query expansion (all queries expanded once, then twice, etc.); and detection and suppression of infinite loops in the proof caused by unfolding of recursive queries.

(2) *Automatic application of integer induction on postconditions that use quantifiers* (Figures 7 and 9). Induction is essential to prove these postconditions. A characteristic of the quantified formulas in the specifications is that they put two terms with the query `get` which are evaluated in two different states, i.e. pre- and poststate, into relation, e.g.:

```
get(o,j)==\old(get(o,j)).
```
(1)

The only useful reasoning step that can be applied to this equation is unfolding these queries which, leads among other formulas to the equation

```
get(o,j-1)==\old(get(o,j-1)).
```
(2)

Hence, if we assume (2), then the original Equation (1) can be proved. However, repeating such unfolding does not terminate, because `j` stands for an arbitrary number. Only for a concrete value, e.g. where $j = 0$, the Equation (1) can be proved using unfolding. Looking closely at these steps one can see that this is induction. We have extended KeY to perform automatically integer induction on the postconditions with quantified formulas. Fortunately, it is sufficient to use the quantified formulas as induction hypotheses that occur in the postconditions to prove them, hence no additional complicated techniques are needed to generate induction hypotheses.

(3) *Reuse already proved formulas as lemmas for further proofs.* The postconditions are proved sequentially and used as premises or lemmas for proving

following postconditions. This extension was necessary due to the semantic dependencies between the postconditions (see Section 5.3). This approach mimics the proving style of the theorem prover Isabelle. Hence, from a broad perspective this idea is not new but we have not seen this style of specification in JML or being applied for the specification of lists in the related work.

With these improvements the verification proof of `remove` involves approximately 100.000 rule applications and the proof of `insert` is in the range of 150.000 rule applications. In comparison, using KeY to verify code of similar size (not related to lists) that does not use recursion and that does not require induction can be typically proved using approximately 1.000 rule applications.

Runtime Checking. For testing the code and the specifications using a runtime checker we have used the automatic random testing tool JET [5]. No changes to the code have been required. However, to create more meaningful tests we have encapsulated the test code with a test driver. The goal was not to check if the code and specification are correct—they already have been formally verified—but rather to check if the specification is compatible with a runtime checker. KeY and JET do not use the same JML dialect, thus we have been required to change the way frame conditions are written (see remark in Section 4.1). Since our approach does not use sophisticated frame conditions but rather only safe approximations, the transformation of specifications between both dialects is safe and trivial. The most important change we have used for an intermediate specification was to introduce upper bounds of quantification in the specification, using the query `size`. Adding the `size` query and the upper bound to the quantifications made the verification more difficult, thus more improvements in KeY were required.

Discussion. We have compared our approach to alternative ones. One alternative is to use arrays or sequences as abstract data types for lists. Such an abstraction stores a copy of the list and provides direct access to its elements via an index, e.g. `a[i]`, similar to `get(o,i)`. The fundamental difference is that an array (sequence) has its own (ghost-) state that exists in parallel to the state of the actual list, whereas the method `get` *derives* a value from the state of the list. Specification and verification of list operations using arrays (sequences) abstraction differ from approaches employing query methods. A coupling invariant that relates the content of the array (sequence) with the state of the list is needed. When the list is modified, the array (sequence) must also be changed explicitly using JML's `set` keyword. For runtime checking this means that the original code must be extended with ghost code. We found that these additional annotations and ghost state simplify verification, since induction is not needed. However, this overhead can make specifications larger and harder to understand, issues which we tried to avoid by using the suggested approach. The approach we followed can also be generalized for handling data types other than lists.

To ensure that the specifications of the methods also work for verification in practice when reasoning with method contracts, we have verified some simple programs that use these methods. Specifying and automatically verifying disjointness of two lists after calling the modifier methods was no problem. We yet have not investigated programs with shared lists, where the nodes u and o are

distinct and there exist integers i and j such that $\texttt{get}(u, i) = \texttt{get}(o, j)$. To verify such programs, additional lemmas are needed. We have verified some of them, such as $\forall o, u : Node. \forall i : int. u = \texttt{get}(o, i) \rightarrow \texttt{get}(u, j) = \texttt{get}(o, i + j)$.

We have experimented also with specifications for trees. Quantification over the nodes of a tree is complicated due to the branching nature of a tree. One possibility is using quantification over arrays which describe paths in the tree. However, runtime checking tools have problems with such quantification and also reasoning is difficult. Several possibilities exist for precisely indexing nodes in a tree using integers. Quantification over integers works for runtime checking but the arithmetics is very complicated for verification. A more suitable specification approach for verification is using a "contains" query for specifying containment of nodes and subtrees. This approach is, however, problematic for runtime checking due to quantification over nodes. Whether it is possible to write specifications for trees that are compatible with verification and runtime checking is thus an open question.

7 Conclusion and Future Work

The paper describes how a specification of a linked data structure can be written that is compatible with runtime checking and verification—a goal that existing specifications often do not satisfy [7]. As an example, we presented a specification of linked list operations using JML that is readable, that is based on first-order logic with integers, and that is, to the best of our knowledge, unique considering all its characteristics. Along that presentation we elaborated problems that arise, related to verification and runtime checking, and our solutions. We developed the ideas and techniques based on several hundred experiments consisting of verification tasks that were conducted during this research.

Using queries for specification makes verification difficult, and has been proposed as a challenge in verification [12]. However, such specifications are easily readable, can be executed by runtime checkers, and can be used as abstractions in verification. Using the self-defined query `get` rather than a special construct, i.e. JML's reachability predicate, enables flexibility, as users can define their own queries. The semantics of such a query is given by its implementation and specification, thus it can be understood by other tools.

We have investigated what verification techniques are needed for automatic verification. Additional techniques we have developed are: strategies for replacing (recursive) queries by their definition in formulas; automatic application of integer induction on the postconditions that contain quantifiers; and reuse of already proven postconditions as premises for proving succeeding postconditions.

Future plans are handling of shared lists and extension of the approach to other linked data structures. One idea is deducing of framing conditions for queries.

Acknowledgment. We would like to thank Thorsten Bormer and Mattias Ulbrich for valuable discussions. We also thank the referees for their helpful comments and suggestions.

References

1. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Román-Díez, G.: Verified resource guarantees for heap manipulating programs. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 130–145. Springer, Heidelberg (2012)
2. Becker, K., Leavens, G.T.: Class LinkedList, `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/java/util/LinkedList.html#removeint`
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Beckert, B., Trentelman, K.: Second-order principles in specification languages for object-oriented programs. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 154–168. Springer, Heidelberg (2005)
5. Cheon, Y.: A quick tutorial on JET. Technical Report UTEP-CS-07-40, Department Technical Reports (CS), University of Texas at El Paso (June 2007)
6. Darvas, A., Müller, P.: Reasoning about method calls in interface specifications. Journal of Object Technology 5(5), 59–85 (2006)
7. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: Reusing a JML specification dedicated to verification for testing, and vice-versa: Case studies. Journal of Automated Reasoning 45, 415–435 (2010), 10.1007/s10817-009-9132-y
8. Jacobs, B., Piessens, F.: Inspector methods for state abstraction. Journal of Object Technology 6(5), 55–75 (2007)
9. Jensen, J.B., Birkedal, L., Sestoft, P.: Modular verification of linked lists with views via separation logic. Journal of Object Technology 10(2), 1–20 (2011)
10. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: Proceedings of POPL, pp. 115–126. ACM (2006)
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SEN 31(3), 1–38 (2006)
12. Leavens, G.T., Leino, R., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
13. Leino, K.R.M.: Specification and verification of object-oriented software. In: Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security, vol. 22, pp. 231–266. IOS Press (2009)
14. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. Logical Methods in Computer Science 5(2) (2009)
15. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
16. Nelson, G.: Verifying reachability invariants of linked structures. In: Proceedings of POPL, pp. 38–47. ACM (1983)
17. Nguyen, H.H., Kuncak, V., Chin, W.-N.: Runtime checking for separation logic. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 203–217. Springer, Heidelberg (2008)
18. Rajamani, S.K.: Verification, testing and statistics. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, p. 25. Springer, Heidelberg (2009)
19. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
20. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: Proceedings of PLDI, pp. 349–361 (2008)