On using KeY to check object usage

João Mota Marco Giunti
António Ravara
NOVA LINCS and NOVA School of Science and Technology, Portugal

July 19, 2022

Contents

1 Introduction

The goal of this report is twofold: (1) assess if KeY [1] can check the **correct use of objects with protocols**, including **protocol completion**, even with objects **shared in collections**; (2) evaluate the **programmer's effort** in making the code acceptable to the tool.

When programming, one naturally defines objects where their method's availability depends on their state [8, 2]. One might represent their intended usage protocol with an automaton or a state machine [11, 12, 5]. **Behavioral types** allow us to statically check if all code of a program respects the protocol of each object. In session types approaches, objects with protocols are usually forced to be used in a linear way to avoid race conditions, which reduces concurrency and restricts what a programmer can do. Given that sharing of objects is very common, it should be supported. For example, pointer-based data structures, such as linked-lists, used to implement collections, usually rely on internal sharing. Such collections may also be used to store objects with protocols and state which needs to be tracked. Moreover, protocol completion is crucial to ensure necessary method calls are not forgotten and resources are freed.

We conclude that KeY supports most features of Java and that it provides a great deal of interactivity, allowing the programmer to guide the proofs, something that other tools rarely provide. Nonetheless, we find some drawbacks:

- Only the verification of sequential Java code is supported;
- The explicit mention of the heap in logical expressions reduces readability;

- The need to often show that two footprints are disjoint can become tedious.

This report is structured as follows:

- Section 2 presents the **motivating example**;
- Section 3 shows a **use example** of the classes explained in the previous section;
- Section 4 discusses how we ensure **protocol completion**;
- Section 5 presents our **detailed assessment**.

2 Motivating example

To make an assessment on KeY with respect to its ability to statically track the state of different objects inside a collection, we present the implementation of a linked-list collection, an iterator for such collection, and a file reader with a usage protocol. Following that, we show an example where file readers are stored in a linked-list and then used according to their protocol. All code is available online[1]. We believe this example is relevant because linked-lists are common data structures. Furthermore, their use of pointers often creates challenges for less expressive type systems[2], so they are great candidates for use case examples.

The linked-list[3] is single-linked, meaning that each node has a reference only to the next node. Internally, there are two fields, *head* and *tail*. The former points to the first node, the latter points to the last node. Items are added to the *tail* of the structure and removed from the *head*, following a FIFO discipline. The file reader[4] has a usage protocol such that one must first call the *open* method, followed by any number of *read* calls until the end of the file is reached (which is checked by calling the *eof* method), and then terminated with the *close* method.

File reader implementation

To model the file reader's protocol, we use pre- and post-conditions in all public methods indicating the expected state and the destination state after the call. To track the current state we use an integer ghost field (line 6 of List. 1). We use constants to identify different states, thus avoiding the use of literal numbers in specifications (lines 2-4). The field *remaining* is the number of bytes left to read. When it is 0, it means we reached the end of the file. Its value should always be equal or greater than zero, as enforced by the invariant in line 14. The *spec_public* annotation allows these fields to be visible in specifications outside the *FileReader* class even though they are private to code outside of the class.

We also declare the class as *final* (line 1) to disallow the existence of subclasses. This will allow us to open the invariant in proofs when necessary. Such would not be possible in the presence of subclasses because the dynamic type of an object is not generally known statically and a unknown subclass could potentially add more invariants.

[1] `https://github.com/jdmota/tools-examples/tree/main/key`

[2] For example, in Rust, one has to follow an ownership discipline, preventing one from creating linked-lists, unless *unsafe* code is used. GhostCell [13], a recent solution to deal with this, allows for internal sharing but the collection itself still needs to respect the ownership discipline. GhostCell uses *unsafe* code for its implementation but was proven safe with separation logic.

[3] `https://github.com/jdmota/tools-examples/blob/main/key/LinkedList.java`

[4] `https://github.com/jdmota/tools-examples/blob/main/key/FileReader.java`

Since KeY uses first-order dynamic logic to verify programs, instead of separation logic for instance, so the proofs explicitly mention the heap and how it is modified. Thus, each object should have a footprint, a set of locations that represent its state, and each of its methods should specify which locations may be modified and if the footprint itself is changed. This technique is based on **implicit dynamic frames** [10, 9], a variant of separation logic supporting heap-dependent expressions. In this example, we declare a set of locations and call it *footprint* (line 9), and specify that it is composed by fields *state* and *remaining* (line 11). The *footprint* is *public* which means that its content can be opened during proofs. The *accessible* annotation in line 10 specifies that *footprint* (i.e. the set itself) will only change if a location in the set mentioned on the right-side of the colon changes. In this particular example, the *footprint* is always the same, so we specify that nothing changes it.[5]

Listing 1: *FileReader* class (part 1)

```
1   public final class FileReader {
2     /*@ ghost public static final int STATE_INIT = 1;*/
3     /*@ ghost public static final int STATE_OPENED = 2;*/
4     /*@ ghost public static final int STATE_CLOSED = 3;*/
5
6     /*@ ghost private spec_public int state;*/
7     private /*@ spec_public @*/ int remaining;
8
9     /*@ public model \locset footprint;
10      @ accessible footprint: \nothing;
11      @ represents footprint = state, remaining;
12      @*/
13
14    /*@ invariant remaining >= 0; @*/
```

The constructor's specification indicates that it will not throw exceptions (*normal_behavior* annotation in line 17 of List. 2), it will not modify (i.e. assign to) any location that exists when it is called (line 18), it ensures that the initial state is *Init* (line 19), and that the set of locations in footprint is now new, thus disjoint from other heap locations (line 20). The *open* method requires that the file reader is in the *Init* state (line 29), and ensures the new state is *Opened* (line 31). The state change is performed with the *set* ghost instruction (line 34). To modify the state, we also need to indicate that the footprint is assignable (line 30). Given that the file reader's footprint is very simple and declared with *public* visibility, we may omit a post-condition indicating that the footprint does not change. The rest of the implementation is similar.[6]

[5] *\nothing* denotes the empty set of locations.

[6] `https://github.com/jdmota/tools-examples/blob/main/key/FileReader.java`

Listing 2: *FileReader* class (part 2)

```
  /*@
    @ normal_behavior
    @ assignable \nothing;
    @ ensures state == STATE_INIT;
    @ ensures \fresh(footprint);
    @*/
  public FileReader() {
    /*@ set state = STATE_INIT;*/
    this.remaining = 20;
  }

  /*@
    @ normal_behavior
    @ requires state == STATE_INIT;
    @ assignable footprint;
    @ ensures state == STATE_OPENED;
    @*/
  public void open() {
    /*@ set state = STATE_OPENED;*/
  }
  ...
}
```

To verify this implementation, we have to prove each method correct, according to each specification, and the fact that nothing changes the footprint (line 10). Each of these proofs is done separately since KeY provides modular verification: the facts provided in the pre-conditions should be enough to show the post-condition holds. Given the simplicity of this class, it was not necessary to guide the proofs, so all the methods were verified automatically using KeY's default proof search strategy. All the proof files showing the verification proof, for this example and the following ones, are available online.[7]

Single file reader usage

Listing 3 presents a method[8] that takes a file reader in the *Init* state and uses it until it reaches the *Closed* state, as indicated by the pre- and post-conditions (lines 4 and 7). To verify the use of the file reader, it is necessary that its invariant holds. Such fact needs to be stated explicitly in the pre-condition using the \invariant_for annotation (line 3). For client users of this code, we state that only the file reader's footprint is modified (line 5). For the loop to be verified, we need to provide some conditions. First, we specify that the file reader's invariant holds for the entirely of the loop's execution and that it is in the *Opened* state (line 12). Then we indicate that only the file's footprint is modified (line 13). Finally, we provide *remaining* as the decreasing value (line 14), thus ensuring termination.

[7] `https://github.com/jdmota/tools-examples/tree/main/key/proofs`
[8] `https://github.com/jdmota/tools-examples/blob/main/key/Main.java#L79-L98`

Listing 3: Single file reader usage

```
 1 /*@
 2   @ normal_behavior
 3   @ requires \invariant_for(f);
 4   @ requires f.state == FileReader.STATE_INIT;
 5   @ assignable f.footprint;
 6   @ ensures \invariant_for(f);
 7   @ ensures f.state == FileReader.STATE_CLOSED;
 8   @*/
 9 private static void use(FileReader f) {
10   f.open();
11   /*@
12     @ loop_invariant \invariant_for(f) && f.state == FileReader.STATE_OPENED;
13     @ assignable f.footprint;
14     @ decreases f.remaining;
15     @*/
16   while (!f.eof()) {
17     f.read();
18   }
19   f.close();
20 }
```

To verify this code, there was no need to guide the proof: KeY generated it fully automatically. Unfortunately, we only succeeded after many attempts at finding the correct specification. Recall that, as stated in the method's contract, only the file's footprint should be modified (line 5). But to prove that, we actually need to show that the footprint itself is not modified. This seems to be necessary because, in the **implicit dynamic frames** approach, one may have heap-dependent expressions in assertions. One would expect that declaring that nothing changes the footprint would be enough (line 10 of List. 1), but sadly that does not seem to be the case. Before we found out that we could make the footprint public, we had to indicate in the post-conditions of file reader's methods that the footprint was not modified by them. By carrying this information with each method call, we could establish that the footprint itself remained unchanged. Now, with the footprint public, we can drop the extraneous post-conditions.

It is important to note that, in general, each method of an object should indicate if and how the footprint itself is modified. But in the file reader's case, since the footprint is a fixed set composed of two field locations, that is not necessary if we make it public. In any case, making the footprint public ended up being very useful in other examples to allow KeY to prove that, for example, the footprints of two objects of unrelated classes were disjoint. Although this seems an obvious fact, it was not always easy to show it, as we will demonstrate later.

Linked-list implementation

The linked-list implementation[9] was heavily inspired in a tutorial by Hiep et al. [6], except we did not implement a doubly-linked-list. To model it we declare several fields (List. 4): *size*, to count the number of values; *head* and *tail*, which point to the first and last nodes, respectively, or *null*, in case the list is empty; *nodeList*, containing a sequence of nodes; and *values*, containing a sequence of values. The values the linked-list is going to store are file readers. Unfortunately, KeY does not support generics[10] so we cannot take a parametric type instead. We could use the *Object* type, which in fact would match Java's semantics (since generics are implemented by

[9]`https://github.com/jdmota/tools-examples/blob/main/key/LinkedList.java`
[10]`https://www.key-project.org/docs/user/RemoveGenerics/`

erasure[11]), but for the purposes of our example, we use the *FileReader* type.

Listing 4: *LinkedList* class (part 1)

```
1  public final class LinkedList {
2    private /*@ spec_public @*/ int size;
3    /*@ nullable @*/ private /*@ spec_public @*/ Node head;
4    /*@ nullable @*/ private /*@ spec_public @*/ Node tail;
5    //@ private spec_public ghost \seq nodeList;
6    //@ private spec_public ghost \seq values;
```

As we did for the file reader, we define the linked-list's footprint (List. 5). Such footprint is composed by all the list's fields (line 11) and the fields of each node in the sequence (line 12). We also specify that the footprint itself only changes if the sequence of nodes changes (line 9), and that the list's invariant may only change if the locations in the footprint change. Notice how the footprints of the values are not part of list's footprint. This is to allow the values to change independently of the linked-list. The proof that the footprint only depends on the nodes sequence was generated automatically by KeY using the default proof search strategy. Showing that the invariant only depends on the locations in the footprint required some interactivity to guide the proof.

Listing 5: *LinkedList* class (part 2)

```
8    /*@ public model \locset footprint;
9      @ accessible footprint: nodeList;
10     @ accessible \inv: footprint;
11     @ represents footprint = size, head, tail, nodeList, values,
12     @     (\infinite_union \bigint i; 0 <= i < nodeList.length; ((Node)nodeList[i])
           .*);
13     @*/
```

The list's invariant is the most verbose part of the implementation (List. 6). First, we specify that *size* is equal to the number of nodes, which is then equal to the number of values (lines 16-17). We also specify that such number cannot overflow (line 18). Then we enforce that the values in the list are not *null* (lines 19-22). One may notice the use of an existential quantifier, indicating that in each position of the sequences, elements exist. This is necessary because KeY treats sequences in a way where they may occasionally contain not-yet-created objects. Additionally, since the sequence declarations (lines 5-6 in List. 4) do not enforce the type of their elements, we need to do it explicitly, either using the *instanceof* operator, or using an existential quantifier. Furthermore, we need to cast the result of accessing a position in a given sequence. Following that, we have to take into account that the list may be empty. So, we define that either the nodes sequence is empty, and the *head* and *tail* fields are *null* (line 23), or the nodes sequence is not empty, and the *head* points to the first node, and *tail* points to the last one (lines 24-26). To ensure we actually have a linked-list, we enforce that the *next* field of each node points in fact to the following node in the sequence (lines 27-28). We also need to ensure that all the nodes are distinct (lines 29-32). Finally, we specify that each value in the values sequence corresponds to the value stored in each node in the same position (lines 33-34).

[11] `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`

Listing 6: *LinkedList* class (part 3)

```
15  /*@ invariant
16    @   nodeList.length == size &&
17    @   nodeList.length == values.length &&
18    @   nodeList.length <= Integer.MAX_VALUE &&
19    @   (\forall \bigint i; 0 <= i < nodeList.length;
20    @       (\exists Node n; n == nodeList[i] && n.value != null)) &&
21    @   (\forall \bigint i; 0 <= i < values.length;
22    @       (\exists FileReader f; f == values[i] && f != null)) &&
23    @   ((nodeList == \seq_empty && head == null && tail == null)
24    @    || (nodeList != \seq_empty && head != null && tail != null &&
25    @         tail.next == null && head == (Node)nodeList[0] &&
26    @         tail == (Node)nodeList[nodeList.length-1])) &&
27    @   (\forall \bigint i; 0 <= i < nodeList.length-1;
28    @       ((Node)nodeList[i]).next == (Node)nodeList[i+1]) &&
29    @   (\forall \bigint i; 0 <= i < nodeList.length;
30    @     (\forall \bigint j; 0 <= j < nodeList.length;
31    @       (Node)nodeList[i] == (Node)nodeList[j] ==> i == j
32    @   )) &&
33    @   (\forall \bigint i; 0 <= i < values.length;
34    @      values[i] == ((Node)nodeList[i]).value);
35    @*/
```

The constructor's contract (List. 7) specifies that no locations that are not fresh are modified (line 39), that the initial sequence of values is empty (line 40), and that the list's footprint is fresh (line 41). Verification was done automatically using the default strategy.

Listing 7: Constructor's contract

```
37  /*@
38    @ normal_behavior
39    @ assignable \nothing;
40    @ ensures values == \seq_empty;
41    @ ensures \fresh(footprint);
42    @*/
```

The *add* method[12] stores a new value at the end of the list. Its contract (List. 8) requires that there must be space to store one more element (line 53), and that only the list's footprint is assignable (line 54). It also indicates that, after the method call, there is a new value at the end of the sequence of values (line 55), and that any locations in the footprint either were present in the old footprint, or are fresh locations (line 56). The verification of the *add* method was also done automatically using the default strategy.

Listing 8: *add* method's contract

```
51  /*@
52    @ normal_behavior
53    @ requires values.length + (\bigint)1 <= Integer.MAX_VALUE;
54    @ assignable footprint;
55    @ ensures values == \seq_concat(\old(values), \seq_singleton(value));
56    @ ensures \new_elems_fresh(footprint);
57    @*/
```

The *remove* method[13] extracts a value from the start of the list. Its contract (List. 9) requires

[12]https://github.com/jdmota/tools-examples/blob/main/key/LinkedList.java#L51-L69
[13]https://github.com/jdmota/tools-examples/blob/main/key/LinkedList.java#L71-L92

that the list cannot be empty (line 73), and that only the list's footprint is assignable (line 74). After the call, it ensures the new sequence of values is the result of taking the old sequence and removing the first element (line 76), the value returned by the method is such element (line 77), and that the new footprint is a subset of the old one (line 78). The verification of the *remove* method required some interactivity, namely to show that the first value was the value of the *head*.

Listing 9: *remove* method's contract

```
 71    /*@
 72      @ normal_behavior
 73      @ requires values.length > 0;
 74      @ assignable footprint;
 75      @ ensures
 76      @   values == \seq_sub(\old(values), 1, \old(values).length) &&
 77      @   \result == (FileReader)\old(values)[0];
 78      @ ensures \subset(footprint, \old(footprint));
 79      @*/
```

The *iterator* method[14] returns a new iterator for this list (List. 10). Its contract specifies that no locations that are not fresh are modified (line 96), that the iterator's invariant holds (line 97), that the iterator and its footprint are fresh (line 98), that the sequence of values iterated through is empty, that the sequence of values still to iterate is equal to the sequence of all values of the list (line 99), and that the iterator is associated with this list (line 100). The iterator's implementation will be presented in more detail in the following section. The verification of the *iterator* method was performed automatically using the default strategy.

Listing 10: *iterator* method

```
 94 /*@
 95   @ normal_behavior
 96   @ assignable \nothing;
 97   @ ensures \invariant_for(\result);
 98   @ ensures \fresh(\result) && \fresh(\result.footprint);
 99   @ ensures \result.seen == \seq_empty && \result.to_see == values;
100   @ ensures \result.list == this;
101   @*/
102 public /*@ pure @*/ LinkedListIterator iterator() {
103   return new LinkedListIterator(this, head);
104 }
```

Iterator implementation

To model the iterator[15] we declare several fields (List. 11): *list*, to point to the original linked-list; *curr*, which points to the current node that show be iterated through or *null*, if we are done iterating the list; *index*, which indicates in what position the current node is in the nodes sequence; *seen*, the sequence of values already iterated through; and *to_see*, the sequence of values still to be iterated.

[14] https://github.com/jdmota/tools-examples/blob/main/key/LinkedList.java#L94-L104
[15] https://github.com/jdmota/tools-examples/blob/main/key/LinkedListIterator.java

Listing 11: *LinkedListIterator* class (part 1)

```
1  public final class LinkedListIterator {
2    private /*@ spec_public @*/ LinkedList list;
3    /*@ nullable @*/ private Node curr;
4    //@ private ghost \bigint index;
5    //@ private spec_public ghost \seq seen;
6    //@ private spec_public ghost \seq to_see;
```

As usual, we define the iterator's footprint (List. 12). Such footprint is composed only by the iterator's fields (line 11). Notice that we do not include the list's footprint as part of the iterator's footprint. We also specify that the footprint itself does not change (line 9), and that the iterator's invariant depends on its footprint and on the list's footprint (line 10). The proof that nothing changes the footprint was done automatically. To prove that the invariant only depends on the iterator's and list's footprints, using KeY's default strategy was not helpful. The reason for this was that KeY was performing multiple "cut" tactics to attempt to close the proof for each possible value of *list.size*. Because of this, we had to guide the proof. Nonetheless, it was mostly straightforward. We just had to use the "observerDependency" tactic which allowed us to establish that the iterator's invariant does not change in the presence of heap updates on locations that do not belong to its footprint or the list's footprint. Having the footprints public was also key to facilitate this result.

Listing 12: *LinkedListIterator* class (part 2)

```
8    /*@ public model \locset footprint;
9      @ accessible footprint: \nothing;
10     @ accessible \inv: footprint, list.footprint;
11     @ represents footprint = list, curr, index, seen, to_see;
12     @*/
```

The iterator's invariant enforces properties over its fields and sequences (List. 13). First, we specify that *index* is a value between zero and the number of values (line 15). This number may be equal to the number of values if and only if we have already iterated through all values. Then we enforce that the values in both sequences are not *null* (line 16-19). This could be derived from the fact that the values in the list are not *null*, but it is useful to keep this statement around. Following that, we define that the *seen* sequence corresponds to the values already seen, from position zero (inclusive) to *index* (exclusive) (line 20), and that *to_see* corresponds to the values to see, from position *index* (inclusive) to the end (line 21). Since *curr* points to the current node, we map it to position *index* in the nodes sequence, or we specify that it is *null*, when iteration is done (line 22). Finally, we keep the information that the list's invariant holds (line 23).

Listing 13: *LinkedListIterator* class (part 3)

```
14   /*@ invariant
15     @   0 <= index && index <= list.values.length &&
16     @   (\forall \bigint i; 0 <= i < seen.length;
17     @       (\exists FileReader f; f == seen[i] && f != null)) &&
18     @   (\forall \bigint i; 0 <= i < to_see.length;
19     @       (\exists FileReader f; f == to_see[i] && f != null)) &&
20     @   seen == \seq_sub(list.values, 0, index) &&
21     @   to_see == \seq_sub(list.values, index, list.values.length) &&
22     @   (index < list.values.length ? curr == (Node)list.nodeList[index] : curr ==
           null) &&
23     @   \invariant_for(list);
24     @*/
```

The constructor accepts two parameters: the list and the initial node. Its contract (List. 14) requires that the initial node be the *head* of the list (line 29), and that the list's invariant holds. Then it guarantees that no location that is not fresh is modified (line 31). Finally, it ensures that, after initialization, this iterator is "connected" to the given list (line 33), that the sequence of seen values is empty (line 34), that the sequence of values to see corresponds to all values in the list (line 35), that the list's invariant still holds (line 36), and that the iterator's footprint is fresh (line 37). To verify the constructor, we had to guide the proof, mostly to establish the invariants of the iterator and list, since KeY was applying "cut" multiple times in the default strategy, as happened previously.

Listing 14: Constructor's contract

```
26    /*@
27      @ normal_behavior
28      @ requires
29      @   l.head == head &&
30      @   \invariant_for(l);
31      @ assignable \nothing;
32      @ ensures
33      @   list == l &&
34      @   seen == \seq_empty &&
35      @   to_see == l.values &&
36      @   \invariant_for(l);
37      @ ensures \fresh(footprint);
38      @*/
```

The *hasNext* method[16] returns *true* is there are still values to iterate through. Its contract (List. 15) indicates that strictly nothing (i.e. no field of the object nor any other location, fresh or not) is assigned to (line 49), and that the boolean result is *true* if and only if the sequence of values to see is not empty (line 50). This method was verified automatically with the default strategy.

Listing 15: *hasNext* method's contract

```
47    /*@
48      @ normal_behavior
49      @ assignable \strictly_nothing;
50      @ ensures \result == (to_see.length > 0);
51      @*/
```

The *next* method[17] advances the iterator. Its contract (List. 16) requires that there are values to see (line 58). Then it specifies that the list's and iterator's footprints are the only accessible (i.e. read) heap locations (lines 59-60), and that only the iterator's footprint is assignable (line 61). When *assignable* annotations are used without *accessible* annotations, it is implicit that the assignable heap locations are the only accessible. In this instance, the list's footprint is not modified but it is read, so we need to use the *assignable* annotation explicitly. Given that such an annotation is used, then we also need to say that the iterator's footprint is read. Following that, the method ensures that the returned value goes from the "to see" sequence to the "seen" sequence (lines 62-65). Finally, we state that the iterator is still "connected" to the same list as before (line 66), so that we do not lose this information.

[16]https://github.com/jdmota/tools-examples/blob/main/key/LinkedListIterator.java#L47-L54
[17]https://github.com/jdmota/tools-examples/blob/main/key/LinkedListIterator.java#L56-L75

Listing 16: *next* method's contract

```
/*@
  @ normal_behavior
  @ requires to_see.length > 0;
  @ accessible list.footprint;
  @ accessible footprint;
  @ assignable footprint;
  @ ensures
  @   seen == \seq_concat(\old(seen), \seq_singleton(\result)) &&
  @   to_see == \seq_sub(\old(to_see), 1, \old(to_see).length) &&
  @   \result == (FileReader)\old(to_see)[0];
  @ ensures list == \old(list);
  @*/
```

To verify the *next* method, we need to prove that: 1. only the list's and iterator's footprints are accessible; 2. the post-condition holds after execution and that only the iterators footprint is modified. Both proof requirements required a lot of work, likely because of the relation between *index* and *curr*, which was probably not obvious to KeY in the default strategy. Some examples of goals which required some effort to prove were: showing that the value of the current node was the first value in the "to see" sequence, proving that such a value was a file reader, and that the new sequences respected the invariant (after the current value was moved to the "seen" sequence).

3 Use example

To exemplify the use of the linked-list with file readers, we instantiate a linked-list and add three file readers in the *Init* state to it (lines 2-8 of List. 17). Then we pass the list to the *useFiles* method (line 9) which iterates through all the files and executes their protocol until all of them reach the *Closed* state. Note that this method may accept a list of any size.

Listing 17: *main* method

```
1  public static void main(String[] args) {
2    LinkedList list = new LinkedList();
3    FileReader f1 = new FileReader();
4    FileReader f2 = new FileReader();
5    FileReader f3 = new FileReader();
6    list.add(f1);
7    list.add(f2);
8    list.add(f3);
9    useFiles(list);
10 }
```

The contract of the *useFiles* method (List. 18) requires that the list's invariant holds (line 4), that all the file readers' invariant holds, that they be in the *Init* state (lines 5-6), and that all the file readers in the sequence of values of the list be distinct (lines 7-10). This method then ensures that all the file readers are in the *Closed* state (lines 13-14). Only the file readers are modified by this method (line 11).

Listing 18: *useFiles* contract

```
/*@
  @ normal_behavior
  @ requires
  @   \invariant_for(list) &&
  @   (\forall \bigint i; 0 <= i < list.values.length;
  @       (\exists FileReader f; f == list.values[i] && f != null && \invariant_for(f) && f.state == FileReader.STATE_INIT)) &&
  @   (\forall \bigint i; 0 <= i < list.values.length;
  @     (\forall \bigint j; 0 <= j < list.values.length;
  @       (FileReader)list.values[i] == (FileReader)list.values[j] ==> i == j
  @   ));
  @ assignable (\infinite_union \bigint i; 0 <= i < list.values.length; ((FileReader)list.values[i]).footprint);
  @ ensures
  @   (\forall \bigint i; 0 <= i < list.values.length;
  @     (\exists FileReader f; f == list.values[i] && f != null && \invariant_for(f) && f.state == FileReader.STATE_CLOSED));
  @*/
public static void useFiles(LinkedList list) { ... }
```

The verification of the *main* method (List. 17) required a lot of interactivity, namely to show that the pre-condition of the *useFiles* method was satisfied. Showing that the list's invariant was straightforward. To prove that the file readers in the values sequence were all distinct, we used the Z3 solver. Proving that all the file readers were in the initial state was the most laborious task. The technique we used was to "cut" on all the possible index values (i.e. 0, 1, 2), and prove for each file reader independently that it was in fact in the initial state. This also requires showing that the *add* operations on the list do not modify the file readers, which in turn requires showing that the list's footprint is disjoint from the file readers' one. Having the footprints public was also important here.

The initial code for the *useFiles* method was based on using an iterator to iterate through each file using a *while* loop (List. 19). Unfortunately, it was very difficult to verify such implementation, because symbolic execution would open all the heap accesses in the hypothesis and goal, which resulted in several *if* statements that could not be simplified, making it very hard to read what was available. These statements relied on showing that, for example, the iterator's footprint was disjoint from the resulted file reader. Enabling the automatic opening of class invariant would help with this but it would also be unnecessary in several cases. To facilitate the proofs, we decided to instead implement a recursive version, although this would not be the most efficient or natural implementation.[18]

Listing 19: *useFiles*: initial implementation

```
LinkedListIterator it = list.iterator();
// loop invariant omitted
while (it.hasNext()) {
  FileReader f = it.next();
  use(f);
}
```

The recursive version of the *useFiles* method (List. 20) starts by creating an iterator for the linked-list (line 1) and then passing it to an helper method called *recursive* (line 2), which iterates for each file reader.[19] The most difficult part was to show that the sequence of values was equal

[18] https://github.com/jdmota/tools-examples/blob/main/key/Main.java#L18-L36

[19] https://github.com/jdmota/tools-examples/blob/main/key/Main.java#L38-L77

to the "seen" sequence of file readers, and that the original values sequence was not modified. We could show this because *recursive* returns leaving the "to see" sequence empty.

Listing 20: *useFiles*: recursive version

```
LinkedListIterator it = list.iterator();
recursive(list, it);
```

The *recursive* method's contract is very verbose (List. 21). It requires that the iterator's and list's invariants hold (lines 4-5), that both are "connected" (line 6), that the files "seen" are *Closed* (lines 7-8), that the files "to see" are in the *Init* state (lines 9-10), and that the file readers in both sequences are distinct (lines 11-18). It then ensures that only the iterator and files "to see" are modified (lines 19-20), and that the pre-conditions are preserved (lines 22-31) except that the "to see" sequence returns empty (line 32). We also provide a decreasing argument (line 21), the size of the "to see" sequence, using the *measured_by* annotation, to ensure termination and allow us to use its own contract when checking the recursive call. Omitting it or using the *decreases* annotation (which does nothing in methods) prevents us from doing this. This method calls *hasNext* to check for file readers to read (line 35). If there are, we extract one (line 36) and call the *use* method, previously presented (List. 3), to read the file and close it (line 37). Then we perform a recursive call (line 38).

Listing 21: *recursive* method

```
1  /*@
2    @ normal_behavior
3    @ requires
4    @   \invariant_for(it) &&
5    @   \invariant_for(list) &&
6    @   it.list == list &&
7    @   (\forall \bigint i; 0 <= i < it.seen.length;
8    @     (\exists FileReader f; f == it.seen[i] && f != null && f.state ==
         FileReader.STATE_CLOSED && \invariant_for(f))) &&
9    @   (\forall \bigint i; 0 <= i < it.to_see.length;
10   @     (\exists FileReader f; f == it.to_see[i] && f != null && f.state ==
         FileReader.STATE_INIT && \invariant_for(f))) &&
11   @   (\forall \bigint i; 0 <= i < it.seen.length;
12   @     (\forall \bigint j; 0 <= j < it.seen.length;
13   @       (FileReader)it.seen[i] == (FileReader)it.seen[j] ==> i == j
14   @   )) &&
15   @   (\forall \bigint i; 0 <= i < it.to_see.length;
16   @     (\forall \bigint j; 0 <= j < it.to_see.length;
17   @       (FileReader)it.to_see[i] == (FileReader)it.to_see[j] ==> i == j
18   @   ));
19   @ assignable it.footprint;
20   @ assignable (\infinite_union \bigint i; 0 <= i < it.to_see.length; ((FileReader
       )it.to_see[i]).footprint);
21   @ measured_by it.to_see.length;
22   @ ensures
23   @   \invariant_for(it) &&
24   @   \invariant_for(list) &&
25   @   it.list == list &&
26   @   (\forall \bigint i; 0 <= i < it.seen.length;
27   @     (\exists FileReader f; f == it.seen[i] && f != null && f.state ==
         FileReader.STATE_CLOSED && \invariant_for(f))) &&
28   @   (\forall \bigint i; 0 <= i < it.seen.length;
29   @     (\forall \bigint j; 0 <= j < it.seen.length;
30   @       (FileReader)it.seen[i] == (FileReader)it.seen[j] ==> i == j
31   @   )) &&
32   @   it.to_see == \seq_empty;
```

```
33     @*/
34   private static void recursive(LinkedList list, LinkedListIterator it) {
35     if (it.hasNext()) {
36       FileReader f = it.next();
37       use(f);
38       recursive(list, it);
39     }
40   }
```

To verify this method, we guided the proof, including the steps of symbolic execution, for mainly two reasons: to control exactly what would be opened and to show early one that *hasNext* returning *true* means that the "to see" sequence is not empty. Of course, the goals that could be closed automatically with KeY or Z3, we did not prove manually.

The most challenging part of verifying this method was (again) related with showing that some parts of the heap were not modified (for example, that the *use* call in line 37 did not modify the iterator nor the sequences), which relied on showing that certain footprints were disjoint. It was also difficult to show that only the file readers in the initially received "to see" sequence were modified. The solution was to open two goals: one where a given field was considered to be in the infinite union described in line 20, and another where that was not the case. Finally, establishing the pre-condition of the recursive call required some work because of the file reader that was moved from the "to see" sequence to the "seen" one.

4 Protocol completion

To ensure that all file readers reach the end of their protocol, we define a contract for the *main* method such that for all file readers created at some point in the program, they are in the *Closed* state (line 3 of List. 22). Since KeY is not aware of the objects created or not before, we define as pre-condition that no file readers exist when *main* is called (line 2). Given that *main* is the first method called in a Java program, this requirement is actually an assumption. In this example, the *use* calls return with the file readers they are given in the *Closed* state (lines 10-12).

Listing 22: *main* method with protocol completion

```
1    /*@
2      @ normal_behavior
3      @ requires !(\exists FileReader f; true);
4      @ ensures (\forall FileReader f; f.state == FileReader.STATE_CLOSED);
5      @*/
6    public static void main(String[] args) {
7      FileReader f1 = new FileReader();
8      FileReader f2 = new FileReader();
9      FileReader f3 = new FileReader();
10     use(f1);
11     use(f2);
12     use(f3);
13   }
```

Because KeY's verification procedure is modular, and we are not interested in unfolding any method's implementation, instead just relying on their contracts, we actually have to specify that no fresh file readers are created in all other methods. This is necessary because it would be possible for methods to create file readers inside, which would exist in the heap as created objects, but for which we would know nothing about. This post-condition, written as `!(\exists FileReader f; \fresh(f))`, was added in all methods of the file reader and in the *use* method.

In the file reader's constructor, we also had to say that f was different from *this*, since the newly created reference is fresh.

All the adapted file reader's methods were verified with the default strategy. The *main* and *use* methods were verified by first running the default strategy following by Z3 to close all the remaining goals. This example with protocol completion is also available online.[20]

Because KeY is based on first-order dynamic logic, it is very simple to use quantifiers to specify that all file readers should have their protocol completed, independently from where they were created during the execution of *main*. Although this seems very powerful, we have to adapt all other methods to specify that no new file readers are created inside.

5 Assessment

KeY's use of JML for specifications, a language for formally specifying behavior of Java code, used by various tools, reduces the learning curve for those that already know JML. Furthermore, KeY supports the greatest number of Java features, allowing one to verify real programs considering the actual Java runtime semantics. Nonetheless, generics are not supported, although there is an automated tool to remove generics from Java programs, which can then be verified with KeY.[21] Because of its focus on Java, KeY is not overly suitable for the verification of algorithms that require abstracting away from the code [4].

One important aspect that makes KeY stand out from other tools is the support for interactivity, which allows the programmer to guide the proofs. This is an aspect that we missed when experimenting with other tools. Additionally, KeY provides useful macros and a high degree of automation, as well as support for SMT solvers, such as Z3. We used Z3 often to more quickly close provable goals, specially those involving universal quantifiers.

Since KeY's core is based on first-order dynamic logic, one can express heap-dependent expressions. In fact, the heap is an object in the logic that is mentioned explicitly. Although this allows for much expressiveness, it often becomes very difficult to verify programs. For example, when attempting to prove that the footprints of two objects were disjoint, we also had to account for possible changes in the footprints themselves, which became very cumbersome. To do this, we depended on making the footprints public so that they could be opened in proofs, otherwise we are not sure if we would have been able to finish the proofs. We think that a variant of KeY that would use separation logic, to abstract away the heaps and the notion of disjointness, would be very helpful and improve readability. Additionally, it would avoid the need to explicitly specify that, for example, the values in a sequence are all distinct, which would reduce the number of verbose specifications. Because we had to be explicitly aware of issues regarding heap disjointness, it was often the case that we had to repeat proofs because we realized that the specification was incorrect (e.g. wrong *assignable* clause) or insufficient to prove something (e.g. missing the *assignable* clause).

Nonetheless, there are probably other alternatives to separation logic that would help in solving the aforementioned issues. In a private conversation with KeY's group leaders [7], they point out that the connectives of separation logic "would get in the way of automation", a crucial feature of KeY. For example, it seems that "the heap separation rule tends to split proofs too early". Furthermore, the problems we encountered can be summarized in two main points: (1) insufficient abstract specification primitives; (2) inability to prove heap and functional properties

[20]`https://github.com/jdmota/tools-examples/tree/main/key/protocol-completion`
[21]`https://www.key-project.org/docs/user/RemoveGenerics/`

separately, in a modular fashion. KeY's team is aware of these issues and will address them in the future (at the time of writing).

In the context of typestates, checking for protocol completion is crucial to ensure that necessary method calls are not forgotten and that resources are freed, thus avoiding memory leaks. Although such concept is not built-in in separation logic, this logic makes it immediately clear which heap chunks are accessed and potentially modified. Together with resource leaking prevention (except for objects with completed protocol), one could ensure protocol completion without the need for adding extraneous post-conditions to all methods. This would be another reason why we believe separation logic would be preferable over first-order dynamic logic, but it is possible there are other alternatives.

We enjoyed the user experience of the interactive theorem prover and appreciated the fact that KeY comes with examples to experiment with. Nonetheless, we believe there is room for improvement. For instance, when we first tried to use KeY, we right-clicked different expressions, which became highlighted as we moved the mouse over them. But the menu that opens up does not provide all the available options. Only later we realized that left-clicking shows a menu with more options. Hitting "Ctrl+Z" to go back is very useful, but we think if it would be more helpful to go back to the last interactive step, instead of returning one step back regardless if it was an automatic step part of a macro or if it was an interactive one. Additionally, we think symbolic execution should avoid opening heap accesses and other kinds of expressions by default, to facilitate the reading process, as we discussed previously. We also noticed that often several equalities were generated, some mentioning names such as *LinkedList_values_5<<selectSK>>*, which are difficult to understand. It would be useful to have a macro that would simplify repeated names, like the *subst* tactic in Coq.[22] Furthermore, we got surprised that expressions such as *list.length* could be parsed, while expressions like *list.length@heap_afterAdd* could not, when using the "cut" tactic, for example. These two expressions are both the prettified version of their more verbose counterpart, but one is recognized and the other is not. It was often the case that we had to turn off "Use Pretty Syntax" to copy expressions in their "real" form and use them it tactics.

Finally, the support for saving and reusing parts of proofs was very useful to us. We wonder if it would be possible to improve such reutilization of proofs in the presence of slight changes to specifications. It was sometimes the case that a whole proof tree would need to be discarded because some part of a specification or code was modified, for example, when we forgot a cast once in the specification. When it is necessary to guide a big part of a proof, that kind of situation can be very frustrating to the user. A questionnaire evaluating the usability of KeY also makes this point, among others we mentioned, like the need to improve readability [3].

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral

[22] `https://coq.inria.fr/refman/proofs/writing-proofs/equality.html#coq:tacn.subst`

types in programming languages. *Foundations and Trends in Programming Languages*, 3 (2-3):95–230, 2016. doi: 10.1561/2500000031.

[3] Bernhard Beckert and Sarah Grebing. Evaluating the Usability of Interactive Verification Systems. In *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*, volume 873 of *CEUR Workshop Proceedings*, pages 3–17. CEUR-WS.org, 2012. URL `http://ceur-ws.org/Vol-873/papers/paper_4.pdf`.

[4] Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *Int. J. Softw. Tools Technol. Transf.*, 17(6):729–744, 2015. doi: 10.1007/s10009-013-0293-y. URL `https://doi.org/10.1007/s10009-013-0293-y`.

[5] José Duarte and António Ravara. Retrofitting Typestates into Rust. In *SBLP'21: 25th Brazilian Symposium on Programming Languages, 2021*, pages 83–91. ACM, 2021. doi: 10.1145/3475061.3475082.

[6] Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. A Tutorial on Verifying LinkedList Using KeY. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 221–245. Springer, 2020. doi: 10.1007/978-3-030-64354-6_9.

[7] Reiner Hähnle. Private communication, July 2022.

[8] Oscar Nierstrasz. Regular types for active objects. *ACM sigplan Notices*, 28(10):1–15, 1993.

[9] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008. doi: 10.1007/978-3-540-78743-3_19.

[10] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012. doi: 10.1145/2160910.2160911.

[11] André Trindade, João Mota, and António Ravara. Typestates to Automata and back: a tool. In *Proceedings 13th Interaction and Concurrency Experience, ICE 2020*, volume 324 of *EPTCS*, pages 25–42, 2020. doi: 10.4204/EPTCS.324.4.

[12] André Trindade, João Mota, and António Ravara. Typestate Editor. `https://typestate-editor.github.io/`, 2022.

[13] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473597.