Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

Citation

Bian, J. (2024, May 21). *Reasoning about object-oriented programs: from classes to interfaces*. Retrieved from https://hdl.handle.net/1887/3754248

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3754248

Note: To cite this publication please use the final published version (if applicable).

Chapter 5

Executable history-based reasoning: a case study

As a case study for the executable history-based reasoning approach, we describe a practical specification and verification effort of part of the `Collection` interface using KeY. To model the history as an ordinary Java class, we introduce a new specification method (in the KeY theorem prover) using histories, that record method invocations including their parameters and return value, on an interface. We provide source and video material of the verification effort to make the construction of the proofs fully reproducible.

This chapter is based on the following publications:

- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). History-based specification and verification of Java collections in KeY. In Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings 16 (pp. 199-217). Springer International Publishing.
- **Bian, J.**, Hiep, H. A. (2020). History-based Specification and Verification of Java Collections in KeY: Video Material. figshare. Collection. https://doi.org/10.6084/m9.figshare.c.5015645.v3
- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). History-based Specification and Verification of Java Collections in KeY: Proof Files. Zenodo. https://doi.org/10.5281/zenodo.3903204

5.1 Introduction

In this chapter, we demonstrate the feasibility of the *executable* history-based (EHB) approach by specifying part of the Java Collection Framework with promising results. The EHB approach allows us to embed histories and attributes in the KeY theorem prover [23] by encoding them as Java objects, thereby avoiding the need to change the KeY system itself. Interface specifications can then be written in the state-based specification language JML [24] by referring to histories and their attributes to describe the intended behavior of implementations. A detailed explanation of this methodology is described in Section 5.2. Further, a distinguishing feature of histories is that they support a *history-based reference implementation* for each interface which is defined in a systematic manner. This allows an important application of our methodology: the verification of the satisfiability of interface specifications themselves. This is done for part of the `Collection` interface in Section 5.3.

Our methodology is based on a symbolic representation of history. We encode histories as Java objects to avoid modifying the KeY system and thus avoid the risk of introducing an inconsistency. Such representation allows the expression of relations between different method calls and their parameters and return values, by implementing abstractions over histories, called *attributes*, as Java methods. These abstractions are specified using JML.

5.2 History-based specification in KeY

The main motivation of the EHB approach is derived from the fact that the KeY theorem prover uses the JML as the specification language and that both JML and the KeY system do not have built-in support for specification of interfaces using histories. Instead of extending JML and KeY, we introduce Java encodings of histories that can be used for the specification of the `Collection` interface, which as such can also be used by other tools [3].

Remark 1. JML supports model fields which are used to define an abstract state and its representation in terms of the concrete state given (by the fields) in a concrete class. For clients, only the interface type `Collection` is known rather than a concrete class, and thus a represents clause cannot be defined. Ghost variables cannot be used either, since ghost variables are updated by adding set statements in method bodies and interfaces do not have method bodies. What remains are model methods, which we use as our specification technique.

5.2.1 The `History` class for `Collection`

In principle, our histories are a simple inductive data type of a sequence of events. Inductive data types are convenient for defining attributes by induction. However, no direct support for inductive definitions is given in either Java or KeY. Thus, we encode histories by defining a concrete `History` class in Java itself, specifically for `Collection`. The externally observable behavior of any implementation of the `Collection` interface is then represented by an instance of the `History` class, and

specific attributes (e.g., patterns) of this behavior are specified by pure methods (which do not affect the global state of the given program under analysis). Every instance represents a particular history value.

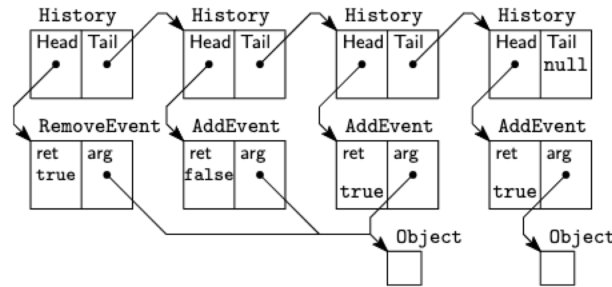

Figure 5.1: A number of history objects. The left-most represents the history of a collection in which `add` is called three times followed by a remove. Intuitively, this history captures the behavior of a set (the addition of an object already contained returns `false`).

The `History` class implements a singly-linked list data structure: a history consists of a head `Event` and a tail `History`. The class `Event` has sub-classes, one for each method of the `Collection` interface. Moreover, there are sub-classes for each method of the `Iterator` interface that additionally track the iterator instance sub-objects. These events are also part of the history of a `Collection`. See Figure 5.1 and Listing 5.1.

```
public class History {
    Event Head;
    /*@ nullable @*/ History Tail;
    /*@ ghost int length; @*/
    // (attributes and their method contracts...)
    // (factory methods... e.g.)
    /*@ pure */ static History addEvent(/*@ nullable */ History h,
            /*@ nullable */ Object o, boolean ret) {
        return new History(new AddEvent(o, ret), h);
    }
}
```

Listing 5.1: The `History` class structure. Later on, the specification of the `addEvent` factory method is given in Listing 5.8.

Each sub-class of the `Event` class comprises the corresponding method's arguments and return value as data. For the `Collection` interface we have the events: `AddEvent`, `RemoveEvent`, `ContainsEvent`, `IteratorEvent`. `AddEvent` has an `Object` field `arg` for the method argument, and a Boolean field `ret` for the return value, that corresponds to the method declaration of `boolean add(Object)`. `RemoveEvent` and `ContainsEvent` are similar. `IteratorEvent` has an `Object` field `ret` for the return value, for `Iterator iterator()`, which is seen as a creation event for the iterator sub-object.

For the `Iterator` interface we have the following events: `IteratorHasNextEvent`,

`IteratorNextEvent`, `IteratorRemoveEvent`. `IteratorHasNextEvent` has a field `inst` for the sub-object instance of `Iterator`, and a Boolean field `ret` for the return value, that corresponds to the method declaration of `boolean hasNext()`. `IteratorNextEvent` has an instance field and an `Object` field `ret`, corresponding to the method declaration `Object next()`. `IteratorRemoveEvent` only has an instance field, since `void remove()` returns nothing.

As part of the `History` class, we define *footprint()* as a JML model method. The footprint of a history is a particular set of heap locations; if those locations are not modified then the value of attributes of the history remains unchanged. In our case, the footprint is the set of fields of events and the singly-linked history list, but we do not include in our footprint the fields of the objects that are elements of the collection, since those never influence any attribute value of a history (we never cast elements of a collection to a specific sub-class to access its fields).

We treat the history as an immutable data type[1]: once an object is created, its fields are never modified. History updates are encoded by the creation of a new history, with an additional new event as the head, pointing to the old history as the tail. Immutability allows us to lift any computed attribute of a history in some heap over heap modifications that do not affect the footprint of the given history. This turns out to be crucial in verifying that an implementation is correct with respect to interface method contracts, where we update a history to reflect that an incoming method call was performed. Such a contract expresses a particular relation between the history's attributes in the heap before and after object creation and history update: the value of an attribute of the old history in the heap before remains the same in the heap after these heap modifications.

5.2.2 Attributes of `History`

It is valuable to describe a specification technique first, that we commonly use, to specify that a particular Java method is a *function* of the heap and its arguments. A JML specification of a Java method is interpreted as a relation, that is, the return value of the method is not necessarily unique, e.g. see Listing 5.2.

```
/*@ ensures \result == 1 || \result == 2; @*/
/*@ strictly_pure @*/ static int nondeterministic(int x);
```

Listing 5.2: A non-deterministic method: its result is not a fixed value.

In KeY, every **pure** method has an *observer symbol* that denotes the outcome of the method call. This is also known as a query method: it typically is used to retrieve the value of encapsulated fields or compute some value without changing the heap. To enforce that the result of a method call is unique, we ensure that the result of the method is the same as its observer symbol, e.g. see Listing 5.3.

```
/*@ ensures \result == deterministic(x); @*/
/*@ strictly_pure @*/ static int deterministic(int x);
```

Listing 5.3: A deterministic method: its result is a fixed value.

[1]By immutable, we mean an object for which its fields after construction are never modified, and its reference type fields point only to immutable objects.

To avoid tying ourselves to a particular history representation, the linked list of events in the history itself is not exposed and cannot be used in specifications. Rather, the history is accessed exclusively through "observer symbols" that map the history to a value. Such observer symbols we call *attributes*. Attributes are defined as **strictly pure** methods since their computation cannot affect the heap. Strictly pure methods are also easier to work with than non-strict or non-pure methods, especially when these methods are used in specifications of the `Collection` interface: these methods evaluate in one heap without modifying it.

The advantage of the use of KeY is that pure methods that appear in specifications as observer symbols can be translated into a modal JavaDL expression, and this allows, more generally, reasoning about pure methods [58]. The rule in the proof system, that replaces observer symbols associated with pure method by a modal expression that expresses the result of a separate symbolic execution of calling the method, is called *query evaluation* [23, Section 11.4].

Attributes are defined inductively over the history. To prove their termination we also introduce a ghost field *length* that represents the length of the history. A ghost field logically assigns to each object a value used for the purpose of verification but is not present at run-time. In each call on the tail of the history, its length decreases, and the length is always positive, thus realizing a so-called decreasing term.

Attributes are functions of the history. The functionality of an attribute amounts to showing dependence (only on the footprint of a history), determinism (uniqueness of result), and termination. To verify that an attribute is deterministic involves two steps: we first symbolically execute the method body until we obtain a proof obligation in which we have to show that the post-condition holds. The post-condition consequently contains, as an observer symbol, the same method applied to the same formal parameters: we use query evaluation to perform *another* symbolic execution of the same method. We need to prove that their outcomes are identical, to verify that the method is deterministic. Not every method can be proven to be deterministic: e.g. if a method body contains a call to a method that cannot be unfolded and that has an unspecified result, then the two symbolic executions (first directly, and secondly through an evaluated query of the observer symbol) need not pick the same result in each method call.

Contents of a `Collection`: The multiset attribute of a `Collection` represents its content and is defined inductively over the structure of the history: the events corresponding to a successful `add` and `remove` call of the `Collection` interface increase and decrease the multiplicity of their argument. Note that removing an element never brings it down to a negative multiplicity. Moreover, `remove` of the `Iterator` interface also decreases the multiplicity; but no longer an argument is supplied because the removed element is the return value of the previous `next` call of the corresponding iterator sub-object. Thus, we define an attribute for each iterator that denotes the object returned by the last `next` call. Calling `remove` on an iterator without a preceding `next` call is not allowed, so neither is calling `remove` consecutively multiple times.

```
/*@ normal_behavior
  @ requires h != null && \invariant_for(h);
  @ ensures \result == History.Multiset(h,o) && \result >= 0;
  @ measured_by h.length;
  @ accessible h.footprint(); // dependency contract
  @*/
/*@ strictly_pure */ static int Multiset(
      /*@ nullable */ History h, /*@ nullable */ Object o) {
    if (h == null) return 0;
    else {
        int c = History.Multiset(h.Tail, o);
        if (h.Head instanceof AddEvent &&
                ((AddEvent) h.Head).arg == o &&
                ((AddEvent) h.Head).ret == true) { // important
            return c + 1;
        } else ...
        return c;
    }
}
```

Listing 5.4: Part of `Multiset` method of the `History` class, with one JML contract.

Listing 5.4 shows part of the implementation of the *Multiset* attribute that is computed by the `Multiset` static method. It is worthwhile to observe that `AddEvent` is counted only when its result is **true**. This makes it possible to compute the *Multiset* attribute based on the history: if the return value is omitted, one cannot be certain whether an add has affected the contents. With this design, further refinements can be made into lists and sets.

Iterating over a `Collection`: Once an iterator is obtained from a collection, the elements of the collection can be retrieved one by one. If the `Collection` is subsequently modified, the iterator becomes invalidated. An exception to this rule is if the iterator instance itself directly modifies the collection, i.e. with its own `Iterator.remove()` method (instead of `Collection.remove(Object)`): calling that method invalidates all *other* iterators. We have added an attribute *Valid* that is true exactly for valid iterators (definition omitted).

For each iterator, there is another multiset attribute, *Visit* (definition omitted), that tracks the multiplicities of the objects already visited. Intuitively, this visited attribute is used to specify the `next` method of an iterator. Namely, `next` returns an element that has not yet been visited. Calling `Iterator.next` increases the *Visit* multiplicity of the returned object by one and leaves all other element multiplicities the same. Intuitively, the iterator increases the size of its *Visit* multiset attribute during traversal, until it completely covers the whole collection, represented by the *Multiset* attribute: then the iterator terminates.

Although these two attributes are useful in defining an implementation of an iterator, they are less useful in showing the client-side correctness of code that uses an iterator.

To show the termination of a client that iterates over a collection, we introduce two *derived* attributes: *CollectionSize* and *IteratorSize*. One can think of the collection's size as a sum of the multiplicities of all elements, and similar for an iterator size of its visited multiset.

5.2.3 The `Collection` interface

```
public interface Collection {
    /*@ model_behavior
      @ requires true;
      @ model nullable History history();
      @*/
    // (interface methods and their method contracts ...)
}
```

Listing 5.5: The *history*() model method of the `Collection` interface.

The `Collection` interface has an associated history that is retrieved by an abstract model method called *history*(). This model method is used in the contracts for the interface methods, to specify what relation must hold of the attribute values of the history in the heap before and after executing the interface method.

As a typical example, we show the specification of the `add` method in terms of the *Multiset* attribute of the new history (after the call) and the old history (prior to the call). The specification of `add` closely corresponds to the informal Javadoc specification written above it. Similar contracts are given for the `remove`, `contains`, and `iterator` methods. In each contract, we implicitly assume a single event is added to the history corresponding to a method call on the interface. The assignable clause is important, as it rules out implementations from modifying its past history: this ensures that the attributes of the old history object in the heap before executing the method have the same value in the heap after the method finished execution.

```
/** Ensures that this collection contains the specified element (optional
  * operation). Returns true if this collection changed as a result of the call.
  * Returns false if this collection does not permit duplicates and already
  * contains the specified element. ... **/
/*@ public normal_behavior
  @ ensures history() != null;
  @ ensures History.Multiset(history(),o) ==
      History.Multiset(\old(history()), o) + (\result ? 1 : 0);
  @ ensures History.Multiset(history(),o) > 0;
  @ ensures (\forall Object o1; o1 != o; History.Multiset(history(),o1) ==
      History.Multiset(\old(history()), o1));
  @ assignable \set_minus(\everything, (history() == null) ? \empty :
      history().footprint());
  @*/
boolean add( /*@ nullable */ Object o);
```

Listing 5.6: The use of *Multiset* in the specification of `add` in the `Collection` interface.

It is important to note that the value of $\backslash result$ is unspecified. The intended meaning of the result is that it is **true** if the collection is modified. There are at least two implementations: that of a set, and that of a list. For a set, the result is **false** if the multiplicity prior to the call is positive, for a list the result is always **true**. Thus it is not possible to specify the result any further in the `Collection` interface that is compatible with both `Set` and `List` sub-interfaces. In particular, consider the following refinements [23, Section 7.4.5] of `add`:

- The `Set` interface ***also*** specifies that $\backslash result$ is **true** if and only if the multiset attribute before execution of the method is zero, i.e.
  ensures `History`.$Multiset(\backslash old(history()), o) == 0 \Longleftrightarrow \backslash result ==$ **true**;
- The `List` interface ***also*** specifies that $\backslash result$ is **true** unconditionally, i.e.
  ensures $\backslash result ==$ **true**;

As in another approach [11], one could use a static field that encodes a closed enumeration of the possible implementations, e.g. set or list, and specify $\backslash result$ directly. Such a closed world perspective does not leave room for other implementations. In our approach, we can obtain refinements of interfaces that inherit from `Collection`, while keeping the interface open to other possible implementations, such as Google Guava's `Multiset` or Apache Commons' `MultiSet`.

5.2.4 History-based refinement

Given an interface specification, we can extract a history-based implementation, that is used to verify there exists a correct implementation of the interface specification. The latter establishes that the interface specification itself is satisfiable. Since one could write inconsistent interface specifications for which there does not exist a correct implementation, this step is crucial.

The state of the history-based implementation `BasicCollection` consists of a single *concrete* history field **this**.`h`. Compare this to the model method of the interface, which only exists *conceptually*. By encoding the history as a Java object, we can also directly work with the history at run-time instead of only symbolically. The concrete history field points to the most recent history, and we can use it to compute attributes. The implementation of a method simply adds for each call a new corresponding event to the history, where the return value is computed depending on the (attributes of the) old history and method arguments. The contract of each method is inherited from the interface.

```
public boolean add(/*@ nullable */ Object o) {
    boolean ret = true;
    this.h = History.addEvent(this.h, o, ret);
    return ret;
}
```

Listing 5.7: One of the possible implementations of *add* in `BasicCollection`.

See Listing 5.7 for an implementation of `add`, that inherits the contract in Listing 5.6. Note that due to underspecification of $\backslash result$ there are several possible

implementations, not a unique one. For our purposes of showing that the interface specification is satisfiable, it suffices to prove that *at least one correct implementation exists*.

For each method of the interface we have specified, we also have a static factory method in the history class which creates a new history object that consists of the previous history as tail, and the event corresponding to the method call of the interface as head. We verify that for each such factory method, the relation between the attributes of the old and the resulting history holds. It is not possible to directly use the constructor of `History` to create a new history, because some methods have the same signature (such as `Collection`'s `add`, `remove`, `contains`). So we introduce an indirection: the constructor for `History` takes an event and another history as tail, but does not have a method contract. For each event we add to the history we define a static factory method, for which we will later prove that the relevant relations between the values of the attributes of the old and new history hold.

```
/*@ normal_behavior
  @ requires h != null ==> \invariant_for(h);
  @ ensures \result != null && \invariant_for(\result);
  @ ensures History.Multiset(\result,o) ==
      History.Multiset(h,o) + (ret ? 1 : 0);
  @ ensures (\forall Object o1; o1 != o;
      History.Multiset(\result,o1) == History.Multiset(h,o1));
  @ ensures \result.Tail == \old(h); */
/*@ pure */ static History addEvent(
      /*@ nullable */ History h, /*@ nullable */ Object o, boolean ret);
```

Listing 5.8: The contract for the factory method for `AddEvent` in class `History`.

For example, the event corresponding to `Collection`'s `add` method is added to a history in Listing 5.8 (see also Listing 5.1). We have proven that the *Multiset* attribute remains unchanged for all elements, except for the argument o if the return value is **true** (see Listing 5.4). This property is reflected in the factory method contract. Similarly, we have a factory method for other events, e.g. corresponding to `Collection`'s `remove`.

5.3 History-based verification of Collection

This section describes the verification work that we performed to show the feasibility of our approach. We use KeY version 2.7-1681 with the default settings. For the purpose of this chapter, we have recorded est. 2.5 hours of video[1] showing how to produce some of our proofs using KeY. A repository of all our produced proof files is available on Zenodo[2] and includes the KeY version we used. The proof files include:

- Contracts for the `Event` class hierarchy, corresponding to methods of the interfaces `Collection` and `Iterator`. These can be verified almost without human

[1] https://doi.org/10.6084/m9.figshare.c.5015645
[2] https://doi.org/10.5281/zenodo.3903203

intervention.

- Contracts specifying properties of history attributes (*Multiset*, *CollectionSize*, *IteratorSize*, *Last*, *LastValid*) and history factory methods corresponding to the events. The implementation of *Multiset* and factory methods not related to iterators have been verified. The verification of these contracts requires significant human intervention.
- Contracts for the history-based implementation of the interfaces `Collection` and `Iterator`. We have verified the contracts for the `add`, `remove` and `contains` events. Importantly, the verification of these methods required the majority of the application of dependency contracts. Also, the verification of these contracts requires significant human intervention.
- Contracts for four example client-side programs. Also, the verification of these contracts requires significant intervention.

The proof statistics are shown in Table 5.1. These statistics must be interpreted with care: shorter proofs (in the number of nodes and interactive steps) may exist, and the reported time depends largely on the user's experience with the tool. The reported time does not include the time to develop the specifications.

Nodes	Branches	I.step	Q.inst	O.Contract	Dep.	Inv.	Time
171,543	3,771	1,499	965	79	263	1	388 min

Table 5.1: Summary of proof statistics. Nodes and branches are measures of proof trees, I.step is the number of interactive proof steps, Q.inst is the number of quantifier instantiation rules, O.Contract is the number of method contracts applied, Dep. is the number of dependency contracts applied, Loop inv. is the number of loop invariants, and Time is an estimated wall-clock duration for interactively producing the proof tree.

We now describe several proofs, that also have been formally verified using KeY. Note that the formal proof produced in KeY consists of many low-level proof steps, of which the details are too cumbersome to consider here.

To verify clients of the interface, we use the interface method contracts. In particular, the client code given in Listing 5.9 makes use of the contracts of `add` and `remove`, to establish that the contents of the `Collection` parameter passed to the program remains unchanged.

```
/*@ ...
  @ ensures (\forall Object o1; !\fresh(o1) ;
     History.Multiset(x.history(),o1) == History.Multiset(\old(x.history()),o1));
  @*/
public static void add_remove(Collection x, Object y) {
    if (x.add(y)) x.remove(y);
}
```

Listing 5.9: Adding an object and if successful removing it again, leaves the contents of a `Collection` the same.

More technically, during the symbolic execution of a Java program fragment in KeY, one can replace the execution of a method with its associated method contract. The contract we have formulated for `add` and `remove` is sufficient in proving the client code in Listing 5.9: the multiset remains unchanged. In the proof, the user has to interactively replace occurrences of history attributes by their method contracts. Method contracts for attributes can in turn be verified by unfolding the method body, thereby inductively establishing their equational specifications. The specification of the latter is not shown here but can be found in the source files.

During the verification, we faced another interesting challenge: dealing with object creation is difficult. We have an alternative program to Listing 5.9 that adds to and removes a newly created object, given in Listing 5.10. Part of the verification of this alternative program (where the object is created by the program) takes 60 minutes, and we fail to close some of the proof branches. In particular, we are unable to show that the multiplicity of newly created objects is zero. Intuitively, we know that a newly created object cannot be part of the past history, as the history only refers to created objects, and a new object was not yet created before. The verification of the program in Listing 5.9 is considerably shorter and takes 4 times less time to prove. Except for our own inability, there seems to be no clear explanation for why these programs have different difficulties.

```
public static void example(Collection x) {
    Object y = new Object(); // Is x.Multiset(y) = 0 true?
    if (x.add(y))
        x.remove(y);
}
```

Listing 5.10: What is the multiplicity of a newly created element?

We needed to introduce a quite technical lemma to show that created objects, e.g. `History` and `Event` objects in the history-based implementation of the `add` method, have the same multiplicity as in the old history: since histories and events inherit from `Object`, in principle these newly created objects could be elements of the collection too. But our lemma shows that this cannot be the case, since these objects are newly created and cannot thus be referred to from the old history (at the time the old history was created, these objects did not yet exist). We could refine the specification of the `add` method of `Collection`: objects created by the implementation must have a zero multiplicity in the post-heap. To see why, consider an implementation that creates a new object. New objects cannot occur in the old history and not as method argument, since new objects are not yet created and all objects referenced and arguments are already created. Hence, the multiplicity of the new object in the old history must be zero, since the multiplicity of any object not occurring in a history is zero. The multiplicity of the new object in the new history must be zero because the new object is not equal to the argument. This argument also applies to an implementation such as `LinkedList`, which creates internal `Node` objects [5].

For the client in Listing 5.11, we make use of the contracts for `iterator` and the methods of the `Iterator` interface. The `iterator` method returns a fresh `Iterator`

sub-object that is valid upon creation, and its owner is set to be the collection. The history of the owning collection is updated after each method call to an iterator sub-object. Each iterator has as derived attribute *IteratorSize*, the size of the visited multiset. It is a property of the *IteratorSize* attribute that it is not bigger than *CollectionSize*, the size of the overall collection. To verify the termination of a client using the iterator in Listing 5.11, we can specify a loop invariant that maintains the validity and ownership of the iterator, and take as a decreasing term the value of *CollectionSize* minus *IteratorSize*. Since each call to `next` causes the visited multiset to become larger, this term decreases. Since an iterator cannot iterate over more objects than the collection contains, this term is non-negative. We never needed to verify that the equational specification for the involved attributes holds and this can be done separately from verifying the client, thus allowing modular verification.

```
public static void iter_only(Collection x) {
    Iterator it = x.iterator();
    /*@ ...
      @ decreasing History.CollectionSize(x.history()) –
      History.IteratorSize(x.history(),it);
      @*/
    while (true) {
      if (!it.hasNext()) {break;}
        it.next(); }
}
```

Listing 5.11: Iterating over the collection.

The other problem we encountered is program rules for dealing with **while** statements with side-effectful guard expressions are difficult to work with. Since a side-effectful guard expression may throw an exception and change the heap, the assumption that the guard completes normally with a true result leads to some post-expression heap. During the symbolic execution of the loop body, the guard expression is also executed so the loop body is executed in the same heap. However, this requires comparing two separate heaps, making it difficult to lift properties from one heap to another if the guard is not deterministic. A workaround is to change the program, where we take a side-effect free guard (such as **true**) and evaluate the side-effectful expression within the loop body: if it is false, we break out of the loop. This avoids working with two different heaps and relating them.

One of the complications of our history-based approach is reasoning about invariant properties of (immutable) histories, caused by potential aliasing. This currently cannot be automated by the KeY tool. We manually introduce a general but crucial lemma, that addresses the issue, as illustrated by the following verification condition that arises when verifying the reference implementation.

One specific verification condition is a conjunct of the method contract for the `add` method of `Collection`, namely that in the post-condition, $Multiset(history(), o) == Multiset(\backslash old(history()), o) + (\backslash result\ ?\ 1 : 0)$ should hold. We verify that in class `BasicCollection` the `add` method is correct with respect to this contract.

Within `BasicCollection`, the model method *history*() is defined by the field **this**.`h`, which is updated during the method call with a newly created history using the factory method `History.addEvent`. We can use the contract of the `addEvent` factory method to establish the relation between the multiset value of the new and old history (see Listing 5.8); this contract is in turn simply verified by unfolding the method body of the multiset attribute and performing symbolic execution, which computes the multiplicity recursively over the history and adds one to it precisely if the returned value is true. Back in `BasicCollection`, after the update of the history field **this**.`h`, we need to prove that the post-condition of the interface method holds (see Listing 5.6); but we already have obtained that this property holds after the static factory method `add` before **this**.`h` was updated.

$$\forall\ \mathbf{int}\ n;\ (n \geq 0 \rightarrow \forall\ \mathit{History}\ g;\ (g.\langle \mathit{inv}\rangle \wedge g.\langle \mathit{created}\rangle = \mathbf{true} \wedge g.\mathit{history_length} = n \rightarrow \mathbf{this}.\mathtt{h} \notin g.\mathit{footprint}()))$$

The update of the history field, as a pointer to the `History` linked list, does not affect this structure itself, i.e. the values of attributes are not affected by changing the history field. This is an issue of aliasing, but we know that the updated pointer does not affect the attribute values of *any* `History` linked list. This can not be proven automatically: we need to interactively introduce a cut formula (shown above) so that the history field does not occur in the footprint of the history object itself. The formula can be proven by induction on the length of the history.

5.4 Summary

In this chapter, we show a new systematic method for history-based reasoning and reusable specifications for Java programs that integrates seamlessly in the KeY theorem prover, without affecting the underlying proof system (this ensures our method introduces no inconsistencies). Our approach includes support for reasoning about interfaces from the client perspective, as well as about classes that implement interfaces. To show the feasibility of our EHB approach, we specified part of the Collection Framework with promising results. We showed how we can reason about clients with these specifications, and showed the satisfiability of the specifications by a witness implementation of the interface. We also showed how to handle inner classes with a notion of ownership. This is essential for showing termination of clients of the `Iterator`.