History-based Specification and Verification of Java Collections in KeY

Hans-Dieter A. Hiep[0000−0001−9677−6644], Jinting Bian,
Frank S. de Boer, and Stijn de Gouw

CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands
{hdh,j.bian,frb,stijn.de.gouw}@cwi.nl

Abstract. In this feasibility study we discuss reasoning about the correctness of Java interfaces using histories, with a particular application to Java's `Collection` interface. 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 outline the challenges of proving client code correct with respect to arbitrary implementations, and describe a practical specification and verification effort of part of the `Collection` interface using KeY (including source and video material).

Keywords: Formal verification · Interface specification · KeY.

1 Introduction

Throughout the history of computer science, a major challenge has been how to assert that software is free of bugs and works as intended. In particular, correctness of software libraries is of the utmost importance because these are the building blocks of millions of programs, and they run on the devices of billions of users. Formal verification gives precise, mathematical proof of correctness of software, with respect to specifications of intended behavior expressed in formal logic. Formal verification can guarantee correctness of software (as opposed, for instance, to testing) but can be challenging in practice, as it frequently requires significant effort in specification writing and constructing proof.

Such effort can very well pay off, as is clearly demonstrated by the use of formal methods which led to the discovery of a major flaw in the design of TimSort—a crash caused by indexing an array out of bounds. TimSort is the default sorting library in many widely-used programming languages such as Java and Python, and platforms like Android. A fixed version, which is now used in all these platforms, was derived and has been proven correct [10] using KeY, a state-of-the-art theorem proving technology [1]. Use of formal methods further led to the discovery of some major flaws in the `LinkedList` implementation provided by Java's Collection Framework—erratic behavior caused by an integer overflow. A fixed version of the core methods of the linked list implementation in Java has also been formally proven correct using KeY [11].

However, some of the methods of the linked list implementation contain an interface type as parameter and were out of scope of the work in [11]. As example we could take the `retainAll` method. Verification of `LinkedList`'s implementation of `retainAll` requires the verification of the inherited `retainAll` method from `AbstractCollection`. The implementation in `AbstractCollection` (see Listing 1) shows a difficult method to verify: the methody body implements an interface method, acts as a client of the supplied `Collection` instance by calling `contains`, but it also acts as a client of the `this` instance by calling `iterator`. Moreover, as `AbstractCollection` is an abstract class and does not provide a concrete implementation of the interface, implementing `iterator` is left to a subclass such as `LinkedList`. Thus arises the need for an approach to specify interfaces which allows us to verify its (abstract) implementations and its clients.

```
public boolean retainAll(Collection c) {
    boolean modified = false;
    Iterator it = iterator();
    while (it.hasNext()) {
        if (!c.contains(it.next())) {
            it.remove();
            modified = true;
        }
    }
    return modified;
}
```

Listing 1. A difficult method to verify: `retainAll` in `AbstractCollection`.

More generally, libraries form the basis of the "programming to interfaces" discipline, which is one of the most important principles in software engineering. Interfaces abstract from state and other internal implementation details, and aids modular program development. However, tool-supported programming logics and specification languages are predominantly state-based which as such cannot be directly used for interfaces. The main contribution of this paper is to show the feasibility of an approach which overcomes this limitation, by integrating history-based reasoning with existing specification and verification methods. This work is the next step towards our ultimate goal of completely specifying and verifying complex software libraries such as the Java Collection Framework, including its `LinkedList` class and `addAll`, `removeAll` and `retainAll` methods.

The formal semantic justification of our approach is provided by the fully abstract semantics for Java introduced in [15] which characterizes exactly the minimal information about a method implementation in a class in a Java library that captures its external use. This minimal information consists of *histories* (also called *traces*) of method calls and returns, and provides a formal semantic justification of the basic observation that such histories completely determine the concrete state of any implementation and thus can be viewed as constituting the generic abstract state space of an interface. This observation naturally leads to the development of a history-based specification language for interfaces.

The background of our approach is given in Sect. 2. An important use case, which leads us to formal requirements on interface specifications, is to reason about the correctness of clients, viz. programs that use instances of an interface by calling methods on it. In Sect. 3 we analyze concrete examples that motivates the design choices that leads us to the core of our approach: we associate to each instance of an interface a history that represents the sequence of method calls performed on the object since its creation. For each method call, the parameters and return value are recorded symbolically in the history. This crucially allows us to define abstractions over histories, called *attributes*, used to describe all possible behaviors of objects regardless of its implementation.

Our methodology is to embed histories and attributes in the KeY theorem prover [1] 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 [13] by referring to histories and its attributes to describe the intended behavior of implementations. This methodology is described in Sect. 4. 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 Sect. 5. We provide source and video material of the verification effort to make the construction of the proofs fully reproducible.

We now discuss related work. It can be empirically established that Java libraries, and Java's Collection Framework in particular, are heavily used and have many implementations [8]. Recently, several issues with parts of the Collection Framework were revealed [10,11]. Such issues are hard to discover at run-time due to their heap size requirements, necessitating a static approach to analysis. Static verification of the Collection Framework was already initiated almost two decades ago, see e.g. the work by Huisman *et al.* [12,14]. What complicates static verification is that it requires formal specifications. Two known approaches are by Huisman [12] and Knüppel *et al.* [16], but their specifications are not complete nor demonstrate the verification of various clients and implementations. Generally speaking, there seems to be no obvious strategy in specifying Java interfaces so that its clients and its implementations can be verified statically by means of a theorem prover. However, for the purpose of run-time verification, numerous approaches exist to specify and check Java programs, such as [3,4,5,6]. Most of these approaches are based on histories. LARVA [7], a tool also mainly developed for run-time verification, was extended in e.g. [2] to optimize away checks at run-time that can be established statically. But, there, static guarantees are limited by expressivity (no fully-fledged theorem prover is used) and interfaces are not handled by the static analysis. Closest to the nature of this work is [17] by Welsch and Poetzsch-Heffter, who reason about backwards compatibility of Java libraries in a formal manner using histories to capture and compare the externally observable behavior of two libraries. In [17], however, two programs are compared, and not a program against a formal specification.

2 Background

In this section, we first provide the context of our work on history-based specification and verification, by giving an overview of the relevant basic concepts, followed by a brief overview of the specification language JML and theorem prover KeY, which are used to realize our approach.

At the lowest level of abstraction, a history is a sequence of events. So the question arises: what events does it contain, and how are the events related to a given program? To concretize this, we first note that in our setting we focus on histories for single-threaded object-oriented programs, and classes and interfaces of Java libraries in particular. For such programs, there are two main kinds of histories: (a) a single global history for the entire program, and (b) a local history *per object*. The first kind, a global history, does not result in a modular specification and verification approach: such a history is specific to a particular program and thus cannot be reused in other programs, since as soon as other objects or classes are added this affects the global history. A global history is therefore not suitable for specifying and verifying Java libraries, since libraries are reused in many different client programs. Hence, in our setting, we tend towards using a local history for each object separately.[1]

Following the concept of information hiding, we assume that an object encapsulates its own state, i.e. other objects cannot directly access its fields, but only indirectly by calling methods. This is not a severe limitation: one can introduce getter and setter methods rather than reading and writing a field directly. But this assumption is crucial to enable any kind of (sound) reasoning about objects: if objects do not encapsulate their own state, any other object that has a reference to it can simply modify the values of the fields directly in a malicious fashion where the new internal state breaks the class invariant of the object[2] without the object being able to prevent (or even being aware of) this.

Assuming encapsulation, each object has full control over its own internal state, it can enforce invariants over its own fields and its state can be completely determined by the sequence of method calls invoked on the object. How an object realizes the intended behavior of each method may differ per implementation: to a client of the object, the internal method body is of no concern, including any calls to other objects that may be done in the method body. We name the calls that an object invokes on other objects inside a method outgoing calls (their direction is out of the object, into another object), and we name the calls made to the object on methods it exposes incoming calls. The above discussion makes clear that the semantics of an object-oriented program can be described purely in terms of its behavior on incoming method calls. Indeed, formally, this is confirmed by Jeffrey and Rathke's work [15] which presents a fully abstract semantics for Java based on traces.

[1] A more sophisticated approach will be introduced for inner classes (see Section 3).

[2] Roughly speaking, a class invariant is a property that all objects of the class must satisfy before and after every method call. Class invariants typically express consistency properties of the object.

KeY and JML. KeY [1] is a semi-interactive theorem prover for Java programs (typically $> 95\%$ of the proof steps are automated). The input for KeY is a Java program together with a formal specification in a KeY-dialect of JML. The user proves the specifications method-by-method. KeY generates appropriate proof obligations and expresses them in a sequent calculus, where the formulas inside the sequent are multi-modal dynamic logic formulas in which Java program fragments are used as the modalities. To reduce such dynamic logic formulas to first-order formulas, KeY symbolically executes the Java program in the modality (it has rules for nearly all sequential Java constructs). Once the program is fully symbolically executed, only formulas without Java program fragments remain.

JML, the Java Modeling Language [13], is a specification language for Java that supports the design by contract paradigm. Specifications are embedded as Java comments alongside the program. A method precondition in JML is given by a `requires` clause, and a postcondition is given by `ensures`. JML also supports class invariants. A class invariant is a property that all instances of a class should satisfy. In the design by contract setting, each method is proven in isolation (assuming the contracts of methods that it calls), and the class invariant can be assumed in the precondition and must be established in the postcondition, as well as at all call-sites to other methods. To avoid manually adding the class invariant at all these points, JML provides an `invariant` keyword which implicitly conjoins the class invariant to all pre- and postconditions. Method contracts may also contain an `assignable` clause stating the locations that may be changed by the method (if the precondition is satisfied), and an `accessible` clause that expresses the locations that may be read by the method (if the precondition is satisfied). Our approach uses all of the above concepts.

Our methodology is based on a symbolic representation of histories. 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.

3 Specification and Verification Challenges for `Collection`

In this section, we highlight several specification and verification challenges with histories that occur in real-world programs. We guide our discussion with examples based on `Collection`, the central interface of the Java Collection Framework. However, note that our approach, and methodology in general, can be applied to all interfaces, as our discussion can be generalized from `Collection`.

A collection contains elements of type `Object` and can be manipulated independently of its implementation details. Typical manipulations are adding and removing elements, and checking whether it contains an element. Sub-interfaces of `Collection` may have refined behavior. In case of interface `List`, each element is also associated to a unique position. In case of interface `Set`, every element is

contained at most once. Further, collections are extensible: interfaces can also be implemented by programs outside of the Java Collection Framework.

How do we specify and verify interface methods using histories?

We focus our discussion on the core methods `add`, `remove`, `contains`, and `iterator` of the `Collection` interface. These four methods comprise the events of our history. More precisely, we have at least the following events:

- `add`$(o) = b$,
- `remove`$(o) = b$,
- `contains`$(o) = b$,
- `iterator`$() = i$,

where o is an element, b is a Boolean return value indicating the success of the method, and i is an object implementing `Iterator`. Abstracting from the implementations of these methods we can still *compute* the contents of a collection from the history of its add and remove events; the other events do not change the contents. This computation results in a representation of the contents of a collection by a multiset of objects. For each object its multiplicity then equals the number of successful add events minus the number of successful remove events. Thus, the contents of a collection (represented by a multiset) is an attribute.

For example, for two separate elements o and o',

`add`$(o) =$ **true**, `add`$(o') =$ **true**, `add`$(o') =$ **false**, `remove`$(o') =$ **true**

is a history of some collection (where the left-most event happens first). The multiplicity of o in the multiset attribute of this history is 1 (there is one successful add event), and the multiplicity of o' is 0 (there is one successful add event, and one successful remove event).

The main idea is to associate each instance to its own history. Consequently, we can use the multiset attribute in method contracts. For example, we can state that the `add` method ensures that after returning `true` the multiplicity of its argument is increased by one, that the `contains` method returns `true` when the argument is contained (i.e. its multiplicity is positive), and that the `remove` method ensures that the multiplicity of a contained object is decreased by one.

How can we specify and verify client-side properties of interfaces?

Consider the client program in Listing 2, where `x` is a `Collection` and `y` is an `Object`. To specify the behavior of this program fragment, we could now use the multiset attribute to express that the contents of the `Collection` instance `x` is not affected.

```
if (x.add(y)) x.remove(y);
```

Listing 2. Adding and removing an element does not affect contents.

Another example of this challenge is shown in Listing 3: can we prove the termination of a client? For an arbitrary collection, it is possible to obtain an object that can traverse the collection: this is an instance of the `Iterator` interface containing the core methods `hasNext` and `next`. To check whether the traversal is still on-going, we use `hasNext`. Subsequently, a call to `next` returns an object that is an element of the backing collection, and continues the traversal. Finally, if all objects of the collection are traversed, `hasNext` returns `false`.

```
Iterator it = x.iterator();
while (it.hasNext()) it.next();
```

Listing 3. Iterating over the collection.

How do we deal with intertwined object behaviors?

Since an iterator by its very nature directly accesses the internal representation of the collection it was obtained from[3], the behavior of the collection and its iterator(s) are intertwined: to specify and reason about collections with iterators a notion of *ownership* is needed. The behavior of the iterator itself depends on the collection from which it was created.

How do we deal with non-local behavior in a modular fashion?

Consider the example in Listing 4, where the collection `x` is assumed non-empty. We obtain an iterator and its call to `next` succeeds (because `x` is non-empty). Consequently, we perform the calls as in Listing 2: this leaves the collection with the same elements as before the calls to `add` and `remove`. However, the iterator may become invalidated by a call that modifies the collection; then the iterator `it` is no longer valid, and we should not call any methods on it—doing so throws an exception.

```
Iterator it = x.iterator(); it.next();
if (x.add(y)) x.remove(y); // may invalidate iterator it
```

Listing 4. Invalidating an iterator by modifying the owning collection.

Invalidation of an iterator is the result of non-local behavior: the expected behavior of the iterator depends on the methods called on its owning collection and also all other iterators associated to the same collection. The latter is true since the `Iterator` interface also has a `remove` method (to allow the in-place removal of an element) which should invalidate all other iterators. Moreover, a successful method call to `add` or `remove` (or any mutating method) on the collection invalidates all its iterators.

We can resolve both phenomena by generalizing the above notion of a history, strictly local to a single object, without introducing interference. We take the iterator to be a 'subobject' of a collection: the methods invoked on the iterator are recorded in the history of its owning collection. More precisely, we also have the following events recorded in the history of `Collection`:

- `hasNext`$(i) = b$,
- `next`$(i) = o$,
- `remove`(i),

where b is a Boolean return value indicating the success of the method, and i is an iterator object. Now, not only can we express what the contents of a collection is at the moment the iterator is created and its methods are called, but we can also define the validity of an iterator as an attribute of the history of the owning collection.

[3] To iterate over the content of a collection, iterators are typically implemented as so-called inner classes that have direct access to the fields of the enclosing object.

4 History-based Specification in KeY

We start with an overview of our methodology: through what framework can we see the different concepts involved? The goal is to specify interface method contracts using histories. This is done in a number of steps:

1. We introduce histories by Java classes that represent the inductive data type of sequences of events, and we introduce attributes of histories encoded by static Java methods. These attributes are defined inductively over the structure of a history. The attributes are used within the interface method contracts (of `Collection`) to specify the intended behavior of every implementation (of `Collection`) in terms of history attributes.
2. Attributes are deterministic and thus represent a function. Certain logical properties of and between attributes hold, comparable to an equational specification of attributes. These are represented by the method contracts associated to the static Java methods that encode the attributes.
3. Finally, we append an event to a history by creating a new history object in a static factory method. The new object consists of the new event as head, and the old history object as tail. The contract for these static methods also expresses certain logical properties of and between attributes, of the new history related to the old history.

The main motivation of our methodology is derived from the fact that the KeY theorem prover uses the Java Modeling Language 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 [4].

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.

4.1 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.

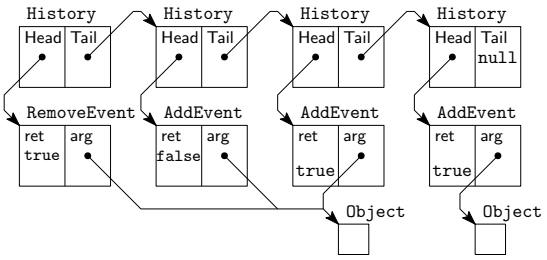

Fig. 1. A number of history objects. The left-most represents the history of a collection in which `add` is called three times followed by a remove. Intuitively, this history captures the behavior of a set (the addition of an object already contained returns `false`).

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 1 and Listing 5.

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 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.

```
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. The `History` class structure. Later on, the specification of the `addEvent` factory method is given in Listing 10.

Remark 2. 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[4]: 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 head, pointing to the old history as 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.

4.2 Attributes of `History`

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", also called "query methods", 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 [9]. The rule in the proof system, that replaces observer symbols associated to pure method by a modal expression that expresses the result of a separate symbolic execution of calling the method, is called *query evaluation* [1, Section 11.4].

Attributes are defined inductively over the history. In order 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.

[4] 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.

Attributes are functions of the history. 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 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 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 6. Part of `Multiset` method of the `History` class, with one JML contract.

Listing 6 shows part of the implementation of the *Multiset* attribute that is computed by the `Multiset` static method. It is worthwile 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 iterators that are valid (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 had 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 client-side correctness of code that uses an iterator. To show 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 multiplicties of all elements, and similar for an iterator size of its visited multiset.

4.3 The `Collection` interface

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

Listing 7. 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 8. 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 [1, 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`.$Multiset(\backslash old(history()), o) == 0 \iff \backslash result ==$ **true**;
- The `List` interface ***also*** specifies that $\backslash result$ is **true** unconditionally, i.e.
  ensures $\backslash result ==$ **true**;

As in another approach [16], 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 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`.

4.4 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 9. One of the possible implementations of *add* in `BasicCollection`.

See Listing 9 for an implementation of `add`, that inherits the contract in Listing 8. Note that due to underspecification of *\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.

```
/*@ 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 10. 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 10 (see also Listing 5). 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 6). 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 History-Based Verification in KeY

This section describes our verification work which 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 article, we have recorded est. 2.5 hours of video[5] showing how to produce some of our proofs using KeY. A repository of all our produced proof files is available on Zenodo[6] and includes the KeY version we used.

The proof statistics are shown in Table 1. These statistics must be interpreted with care: shorter proofs (in the number of nodes and interactive steps) may exists, 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.**	**Loop inv.**	**Time**
171,543	3,771	1,499	965	79	263	1	388 min

Table 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 instansiation 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 a number of 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 verification challenge in Listing 2 makes use of the contracts of `add` and `remove`, to establish that the contents of the `Collection` parameter passed to the program in Listing 2 remains unchanged. More technically, during symbolic execution of a Java program fragment in KeY, one can replace the execution of a method by its associated method contract. The contract we have formulated for `add` and `remove` is sufficient in proving the client code in Listing 2: 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.

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

For the verification challenge in Listing 3, 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 termination of a client using the iterator in Listing 3, we can specify a loop invariant that maintains the validity and ownership of the iterator, and take as 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 hold and this can be done separately from verifying the client, thus allowing modular verification.

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 verification condition is a conjunct of the method contract for the `add` method of `Collection`, namely that in the post-condition, $\mathit{Multiset}(\mathit{history}(), o) == \mathit{Multiset}(\backslash\mathit{old}(\mathit{history}()), o) + (\backslash\mathit{result}\ ?\ 1 : 0)$ should hold. We verify that `BasicCollection`'s `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 10); 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 8); 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) 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.

6 Conclusion

Programming to interfaces is one of the core principles in object-oriented programming and central to the widely-used Java Collection Framework, which provides a hierarchy of interfaces and classes that represent object containers. But current practical static analysis tools, including model checkers and theorem provers such as KeY, are primarily state-based. Since interfaces do not expose a state or concrete representation, a major question is how to support interfaces.

The main contribution of this paper is 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 novel method, 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`.

This work is the next step in the formal verification of Java's Collection Framework. With our novel method we can continue our specification and verification work on `LinkedList`, including methods with arguments of interface type such as `addAll`, and its inherited methods `removeAll` and `retainAll`.

A direction for future work is to further improve practicality of history-based specification and verification: for example, (a) considering client-side correctness with multiple (potentially aliasing) objects implementing the same interface, (b) considering client-side correctness that involves objects that implement multiple (potentially interfering) interfaces, (c) developing techniques to show that certain combinations of interfaces are inconsistent, such as an object implementing both `List` and `Set`, (d) considering implementations that initialize the value of attributes by an arbitrary value at creation time (e.g. a non-empty collection when it is constructed) which necessitates an object creation event, and (e) encoding histories as built-in abstract data types with special proof rules, to avoid modeling histories as Java objects.

Acknowledgements The authors thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016)
2. Azzopardi, S., Colombo, C., Pace, G.J.: CLARVA: Model-based residual verification of Java programs. In: Model-Driven Engineering and Software Development (MODELSWARD). pp. 352–359. SciTePress (2020)
3. de Boer, F.S., de Gouw, S., Vinju, J.J.: Prototyping a tool environment for run-time assertion checking in JML with communication histories. In: Formal Techniques for Java-Like Programs (FTfJP). pp. 6:1–6:7. ACM (2010)
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7**(3), 212–232 (2005)
5. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 569–588. ACM (2007)
6. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal **15**(1), 7–25 (2007)
7. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java programs (tool paper). In: Software Engineering and Formal Methods (SEFM). pp. 33–37. IEEE Computer Society (2009)
8. Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical study of usage and performance of Java collections. In: Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering. pp. 389–400 (2017)
9. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 4422, pp. 336–351. Springer (2007)
10. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reasoning **62**(1), 93–126 (2019)
11. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 12079, pp. 217–234 (2020)
12. Huisman, M.: Verification of Java's AbstractCollection class: A case study. In: International Conference on Mathematics of Program Construction. pp. 175–194. Springer (2002)
13. Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java Modeling Language. In: [1], pp. 193–241. Springer (2016)
14. Huisman, M., Jacobs, B., van den Berg, J.: A case study in class library verification: Java's Vector class. Int. J. Softw. Tools Technol. Transf. **3**(3), 332–352 (2001)
15. Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java language. In: Programming Languages and Systems (PLS). LNCS, vol. 3444, pp. 423–438. Springer (2005)
16. Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. In: Workshop on Formal Integrated Development Environment (F-IDE). EPTCS, vol. 284, p. 53–70. OPA (2018)
17. Welsch, Y., Poetzsch-Heffter, A.: A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. Science of Computer Programming **92**, 129–161 (2014)