Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

Chapter 4

History-based reasoning about interfaces

Programming to interfaces is one of the core principles in object-oriented programming. However, 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.

In this chapter, we discuss reasoning about the correctness of Java interfaces using histories, with a particular application to Java's Collection interface. Histories, defined as sequences of method calls and returns, offer a novel approach to specifying state-hiding interfaces. We outline the challenges of proving client code correct with respect to arbitrary implementations with histories. To specify interface method contract using histories, we present two approaches: the *executable* history-based approach, which models histories as an ordinary Java class, and the *logical* history-based approach, which models histories as an external abstract data type with functions.

This chapter is based on the following publications:

- **Bian, J.**, Hiep, H. A., de Boer, F. S., de Gouw, S. (2023). Integrating ADTs in KeY and their application to history-based reasoning about collection. Formal Methods in System Design, 1-27.
- 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.

4.1 Introduction

The importance and potential of formal software verification as a means of rigorously validating state-of-the-art, real software and improving it, is convincingly illustrated by its application to `LinkedList` implementation within the Java Collection Framework. In the previous chapter, we focused on the specification and verification of the `add` and `remove` methods. A fixed version of the core methods of the linked list implementation in Java has also been formally proven correct using KeY [5].

However, some of the methods of the linked list implementation contain an interface type as a parameter and were out of the scope of the work in [5]. For 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 4.1) shows a difficult method to verify: the method 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 that 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 4.1: 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 aid modular program development. However, tool-supported programming logic and specification languages are predominantly state-based which as such cannot be directly used for interfaces. For example, JML is inherently state-based. JML mainly provides support for the specification of classes in terms of their fields, including so-called model fields that represent certain aspects of the data structure underlying the implementation.

The main contribution of this chapter is to show the feasibility of an approach that overcomes state-based limitations, by integrating history-based reasoning with existing specification and verification methods. The approach detailed in [12] comes

closest to our desired goal: it supports data and integrates with JML. The specification language is based on attribute grammars. Call or return events are represented as grammar terminals with attributes that store e.g. the actual parameters. To specify properties of sequences of such events, the user first introduces attributes of non-terminals (where each grammar production is annotated with code that computes the value of the attribute) and then uses assertions over this attribute to specify properties. The original program is instrumented with calls to update the history at the beginning and end of a method. The new values of the attributes are determined by parsing the new history in the grammar.

The formal semantic justification of our approach is provided by the fully abstract semantics for Java introduced in [22]. This semantic framework precisely isolates the essential elements of a Java class method that are visible during external use. Such essential elements are captured as *histories* of method calls and returns. These histories not only serve as a full representation of an implementation's concrete state but also define a universal abstract state for any given interface. Our methodology is based on a symbolic representation of histories. Such representation allows the expression of relations between different method calls and their parameters and return values, by implementing abstractions over histories, called *attributes*. These abstractions are specified using JML.

The background of our approach is given in Sect. 4.2. 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. 4.3 we analyze concrete examples that motivate the design choices that lead 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 their implementation. Our methodology for embedding histories and attributes into the KeY theorem prover is described in Sect. 7.3.1. We explore two approaches: either encoding histories and attributes as Java objects or encoding them as abstract data types. We discuss these two approaches and compare their respective strengths and weaknesses across several aspects in the last section.

4.2 Background

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 without the object being able to prevent (or even being aware of) this. 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 the consistency properties of the object. For example, an instance of `ArrayList` has a `size` field that is supposed to represent the number of items in the list. A simple class invariant is $\texttt{size} \geq 0$. If the `ArrayList` does *not* encapsulate the `size` field, the following can happen:

```
ArrayList ℓ = new ArrayList();
ℓ.size = −1; // size becomes negative!
```

Without encapsulation, the list cannot enforce its own class invariant that its size is non-negative! This causes many issues. For example, the list exposes an `add` method which executes `elementData[size++] = e;` so calling `add` on the above list causes a crash because it accesses an array at -1, a negative index!

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 [22] which presents a fully abstract semantics for Java based on traces.

4.3 Specification and verification challenges for the `Collection` interface

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.

[1] A more sophisticated approach will be introduced for inner classes (see Section 4.3).

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 they contain an element. Sub-interfaces of `Collection` may have refined behavior. In the case of interface `List`, each element is also associated with a unique position. In the 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:

- $\texttt{add}(o) = b$,
- $\texttt{remove}(o) = b$,
- $\texttt{contains}(o) = b$,
- $\texttt{iterator}() = i$,

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 content of a collection (represented by a multiset) is an attribute.

For example, for two separate elements o and o',
$\texttt{add}(o) = \textbf{true},\ \texttt{add}(o') = \textbf{true},\ \texttt{add}(o') = \textbf{false},\ \texttt{remove}(o') = \textbf{true}$
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 with 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 4.2, 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 content of the `Collection` instance `x` is not affected.

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

Listing 4.2: Adding and removing an element does not affect contents.

Another example of this challenge is shown in Listing 4.3: 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 ongoing, 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 4.3: 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,[1] 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 4.4, 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 4.2: 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 4.4: 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 with 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

[1] 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.

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 content 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.

This does warrant a short discussion about the consistency of a history: not all histories are consistent. By consistency, we mean there exists a client and a correct implementation that can produce the history. To see why, consider the program where an iterator invalidates some other iterator, in Listing 4.5.

```
static void example(Collection x) { // assume non−empty x
    Iterator it = x.iterator();
    Iterator jt = x.iterator();
    it.next();
    it.remove(); // invalidates jt
    jt.next(); // should throw exception
}
```

Listing 4.5: Invalidating an iterator by another iterator.

Suppose the history for collection x is consistent and we record the method invocations that return normally. The last `next` method on jt is not recorded in the history. Thus, there are sequences of events that are never produced by any client, because somewhere in the middle of those sequences an exception is thrown. A history is consistent if none of the methods associated with the recorded events throw an exception.

How do we deal with client-side correctness with multiple objects implementing the same interface?

Binary methods are methods that act on two objects that are instances of the same interface. The difficulty in reasoning about binary methods [52] lies in the fact that one instance may, by its implementation of the interface method, interfere with the other instance of the same interface. For example, as shown in Listing 4.6, the method `Collection#addAll(Collection)` is a binary method: both the receiving object and the supplied argument are instances of the interface `Collection`.

```
/** Adds all of the elements in the specified collection to this collection
  * (optional operation).
  * The behavior of this operation is undefined if the specified collection is
  * modified while the operation is in progress.
  * This implies that the behavior of this call is undefined if the specified
  * collection is this collection, and this collection is nonempty. **/
boolean addAll(Collection c);
```

Listing 4.6: The `Collection#addAll(Collection)` method.

By using our history-based approach, we can limit such interference by requiring that the history of the other instances remains the same during the execution of a method on some receiving instance. Consequently, properties of other collection's histories remain *invariant* over the execution of methods on the receiving instance.

For client-side verification, verifying clients that operate on two collections concurrently is interesting. This is because each collection may have a distinct implementation, and there's potential for mutual interference. Our applied strategy here emphasizes the identification of properties that consistently remain *invariant* across histories of all collections. For instance, invoking a method on one collection should not alter the history of any other collection.

4.4 History-based reasoning approach

Reasoning about the correctness of interfaces involves two aspects: the client side and the implementation side of an interface. A client of an interface is a program fragment that uses instances of the interface by calling methods on it. An implementation of an interface is an instance of a class that implements the interface, by providing a method body for each of the methods defined in the interface. Clients of interfaces need not have knowledge of their implementations. Thus, the state of an implementation is hidden from the client. Moreover, clients accept any implementation, even those not conceived at the moment the client is designed: verification of an interface client is in that sense open-ended.

The verification of interface clients and verification of interface implementations depends on the specification technique applied to the interface. We need a way to encode histories in the formalism used in expressing specifications. There are two approaches we identify, we refer to model histories using Java classes [53] as the *executable* history-based (EHB) approach, and model histories using abstract data types as the *logical* history-based (LHB) approach.

4.4.1 The executable history-based approach

The EHB approach is 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.

We give an overview of the EHB approach: 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 to specify the intended behavior of every implementation 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 with 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 the head and the old history object as the 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 practical specification and verification effort of a part of the `Collection` interface employing the EHB approach is detailed in Chapter 5.

4.4.2 The logical history-based approach

In state-based approaches, including the work by Knüppel et al. [11], (dynamic) frames [54] inherently heavily depend on the chosen representation, i.e. at some point, the concrete fields that are touched or changed must be made explicit. The same holds for separation logic [55] approaches for Java [56]. Since interfaces do not have a concrete state-based representation, *a priori* specification of frames is not possible. Instead, for each class that implements the interface, further specifications must be provided to name the concrete fields. One can abstract from these concrete fields by using a *footprint* model method that specifies the frame dynamically, i.e. a frame may depend on the state. However, the footprint model method itself also requires a frame, leading to recursion in dependency contracts [57]. Moreover, any specification that mentions (abstract or concrete) fields can be problematic for clients of classes, since the concrete representation is typically hidden from them (using an interface), which raises the question: how to verify clients that make use of interfaces?

The LHB approach avoids specifying such frames, thus eliminating much effort needed in specification: there is no need to introduce *ad hoc* abstractions of the underlying state, as the complete behavior of an interface is captured by its history. The main core of the LHB approach is modeling histories as abstract data types, ADTs for short. Additionally, since we model such histories as elements of an ADT

separate from the sorts used by Java, histories can not be touched by Java programs under verification themselves, and so we never have to use dependency contracts for reasoning about properties of histories. This allows us to avoid the bottlenecks that arise in the approach of EHB approach, which uses an encoding of histories as ordinary Java objects living on the heap.

The LHB approach can be done in a number of steps:

1. We begin by defining algebraic data types and functions to logically model the domain-specific knowledge relevant to the Java program we aim to verify. These definitions rely on polymorphic type parameters, abstracting away from specific Java types and allowing for a more generalizable and flexible model.
2. We translate the signatures of our data types and functions into a formal verification environment, mapping them to appropriate sorts and function symbols. This step involves writing specifications for the Java program in a way that integrates these new sorts and function symbols, ensuring that our model aligns with the program's structure and behavior.
3. We proceed with symbolic execution of the Java program, leading to the generation of proof obligations. These obligations might initially contain uninterpreted symbols, limiting direct reasoning. To address this, we specify and prove additional properties that capture our expectations about these symbols. Successful proofs are then incorporated back into the symbolic execution environment, enhancing our ability to reason about and verify the Java program.

The last step will usually be repeated many times until we finish the overall proof because typically one can not find all required lemmas at once.

The case study demonstrating the LHB approach, which reasons about Java's Collection interface using histories and proves the correctness of several clients that operate on multiple objects, is presented in Chapter 6.

4.4.3 Comparative analysis

At its core, the LHB approach can be viewed as an advanced approach of the EHB approach, encapsulating the growth and advancement of our research. We explain the advantages of the LHB approach in the following aspects: logical representation, expressiveness of attributes, logical consistency, and verification complexity.

Logical representation

Modeling histories modeled as ADT is possible by declaring a new logical sort for histories and events. Thus histories do not have any representation in the program itself, but only in the theorem prover and they are immutable and inaccessible: no program can modify or even inspect a history value from this sort. Since histories thus have a run-time representation in the program (they are ordinary Java objects on the heap) they can potentially be modified by a program. To ensure meaningful specifications one thus needs to ensure that histories are not accessed by a program under analysis. Histories are extended by the creation of a new history object with a new event corresponding to a method call, its return value, and points to an old and

unmodified history object. This requires showing that history/event objects occupy a separate part of the heap: modifications during the creation of new histories and events do not affect old history objects. Clearly, modeling histories and events as a separate ADT avoids these non-trivial proof steps in the first place. Thus, for static verification, one can consider the logical representation to be a positive aspect of the LHB approach and a negative aspect of the EHB approach.

Expressiveness of attributes

In the LHB approach, with ADT, history attributes can be defined logically by specifying attributes using a (recursive) system of equations. The definitions can use idealized mathematical data types and operations, such as mathematical (unbounded) integers. Attributes of objects need to be expressed within the Java programming language. Java is Turing-complete so in principle all computable functions are available to define attribute values. While Java does contain an unbounded integer type (`BigInteger`) its use typically complicates reasoning.

Logical consistency

In the EHB approach, we introduce no new rules of the proof system: the consistency is thus the same as that of the base system. In the LHB approach, we add new rules to the proof system so there is a risk of introducing an inconsistency. However, our LHB approach allows leveraging Isabelle/HOL to guarantee, for example, meta-properties such as the consistency of axioms about user-defined ADT functions.

Verification complexity

The encoding of the EHB approach made use of pure methods in its specification and thus required extensive use of so-called *accessibility* clauses, which express the set of locations on the heap that a method may access during its execution. These accessibility clauses must be verified. Furthermore, for recursively defined pure methods we also need to verify their *termination* and *determinism* [25]. Essentially, the associated verification conditions boil down to verifying that the method under consideration computes the same value starting in two heaps that are different except for the locations stated in the accessibility clause. To that end, one has to symbolically execute the method more than once (in two different heaps) and relate the outcome of the method starting in different heaps to one another. After such proof effort, accessibility clauses of pure methods can be used in the application of *dependency contracts*, which are used to establish that the outcome of a pure method in two heaps is the same if one heap is obtained from the other by assignments outside of the declared accessible locations. The degree of automation in the proof search strategy with respect to pure methods, accessibility clauses, and dependency contracts turned out to be rather limited in KeY. So, while the methodology works in principle, in practice, for advanced use, the pure methods were a source of large overhead and complexity in the proof effort. In contrast, in the LHB approach, elements of abstract data types are not present on the heap, avoiding the need to use dependency contracts to prove that heap modifications affect their properties.

While the LHB method offers advantages over the EHB in various dimensions, it

comes with its own set of demands. Adopting the LHB approach necessitates not only proficiency in Java and JML but also an in-depth understanding of taclets and JavaDL. Moreover, verifiers need to be well-acquainted with domain-specific theorem provers like KeY, as well as general-purpose theorem provers, such as Isabelle/HOL in our case.

To conclude, for those who are new to history-based reasoning, the EHB approach offers a gentler introduction. It situated specifications within the programming language context, making functions computable and available in Java. However, the advantage of the LHB approach is that it opens up the possibility of defining many more functions on histories, thus furthering the ability to model complex object behavior: this we demonstrated by verifying complex and realistic client code that uses collections in Chapter 6.

4.5 Summary

In this chapter, we address a key challenge in object-oriented programming: the specification and verification of state-hiding interfaces. Traditional static analysis tools, such as KeY theorem prover, often fall short in this context due to their emphasis on state-based reasoning. Our work introduces a novel methodology for reasoning about Java interfaces through the use of histories, which are sequences of method calls and returns. We particularly apply this reasoning to Java's Collection interface and discuss several challenges associated with using histories in real-world programs. We explore two approaches to implementing the concept of histories: the EHB approach, which models histories as standard Java classes, and the LHB approach which treats histories as external ADTs with associated functions. These methodologies offer new possibilities for verifying the correctness of client code in relation to the expected behavior of interface implementations.