Chapter 6

Logical history-based reasoning: an advanced case study

We discuss integrating abstract data types in the KeY theorem prover by a new approach to model data types using Isabelle/HOL as an interactive back-end and represent Isabelle theorems as user-defined taclets in KeY. As a case study of the logical history-based (LHB) approach, we reason about Java's `Collection` interface using histories, and we prove the correctness of several clients that operate on multiple objects, thereby significantly improving the state-of-the-art of history-based reasoning.

This chapter is based on the following publications and artifacts:

- **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.
- **Bian, J.**, Hiep, H. A., de Boer, F. S., de Gouw, S. (2021). Integrating ADTs in KeY and their application to history-based reasoning. In Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24 (pp. 255-272). Springer International Publishing.
- **Bian, J.**, Hiep, H. A. (2021). Integrating ADTs in KeY and their Application to History-based Reasoning: Video Material. figshare. Collection. https://doi.org/10.6084/m9.figshare.c.5413263.v1
- **Bian, J.**, Hiep, H. A., de Boer, F. S., de Gouw, S. (2022). Integrating ADTs in KeY and their Application to History-based Reasoning about Collection: Proof files. https://doi.org/10.5281/zenodo.7079126

6.1 Introduction

In this chapter, we present an advanced case study focused on the *logical* history-based approach (LHB approach). The main core of the LHB approach is to model histories logically as a user-defined abstract data type. Given that KeY has limited support for user-defined abstract data types (ADTs), we introduce a general *workflow* which integrates the domain-specific theorem prover KeY and the general-purpose theorem prover Isabelle/HOL [59] for the specification of ADTs.

More generally, in our set-up, we distinguish domain-specific theorem provers, in our case KeY, from general-purpose theorem provers, in our case Isabelle/HOL. The domain-specific theorem prover acts as a verification condition generator: KeY has domain-specific knowledge of the programming language (Java) and program specification language (JML) in question. The theorems of a domain-specific theorem prover are correct pairs of programs and specifications and thus can be seen as giving axiomatic semantics to programs and specifications. A general-purpose theorem prover, in contrast, is oblivious to the intricate details of programs and their specifications in question: e.g. it is not needed to formalize the semantics of Java nor JML in our general-purpose theorem prover Isabelle/HOL. Our set-up thus differs from other approaches, such as in the Bali [59, 60] and LOOP [61, 62] projects, that *embed* the semantics of the programming language and specification language within the general-purpose theorem prover.

The idea presented in this paper of integrating Isabelle/HOL and KeY arises out of the need for user-defined data types usable within specifications. Other tools, such as Dafny [63] and Why3 [64], support user-defined data types in the specification language, contrary to JML as it is implemented by KeY. However, the former tools are not suitable for verifying Java programs: for that, as far as the authors know, only KeY is suitable due to its modeling of the many programming features of the Java language present in real-world programs.

We apply our workflow to the Java Collection interface, study a number of example client use cases of the interface, and compare our new approach with the previous approach described in [53]. Although the EHB approach works *in principle*, with our new approach we can *practically* give a specification of the `addAll` method and verify the correctness properties of its clients. Going further, we are now able to reason about advanced, realistic use cases involving multiple instances of the same interface: we also have verified a complex client program that destructively compares *two* collections.

6.2 Intergrating Abstract Data Types in KeY

Abstract data types were introduced in 1974 by Barbara Liskov and Stephen Zilles [65] to ease the programming task: instead of directly programming with concrete data representations, programmers would use a suitable abstraction that instead exposes an interface, thereby hiding the implementation details of a data type. In most programming languages, such interfaces only fix the signature of an abstract data type (e.g. Java's interface or Haskell's typeclass). Further research has led to many approaches for specifying abstract data types, e.g. ranging from simple equational specifications to axiomatizations in predicate logic. See for an extensive treatment of the subject in the textbook [66].

In the context of our work, we need to distinguish the two levels in which abstract data types can appear: at the programming level, and at the specification level. In fact, Java supports abstract data types by means of its interfaces, and for example, the Java Collection Framework provides many abstractions to ease the programming task. The specification language JML does support reasoning about the instances of such interfaces, but does not allow user-defined abstract data types on the specification level only. The reason is that JML is designed to be "easier for programmers to learn and less intimidating than languages that use special-purpose mathematical notations" [67]. There are extensions of JML to support user-defined types on the specification level, e.g. model classes [68], but KeY does not implement them.

However, KeY does extend JML in an important way: several built-in abstract data types at the specification level are provided [23, Section 2.4.1]. There is the abstract data type of *sequences* that consists of finite sequences of arbitrary elements. Further, KeY provides the abstract data type of *integers* that comprises the mathematical integers (and not the integers modulo finite storage, as used in the Java language) to interpret JML's `\bigint`. Elements of these abstract types are not accessible by Java programs and are not stored on the heap. It is possible to reason about elements of such abstract data types since the KeY theorem prover allows the definition of their *theories* implemented by inference rules for deducing true statements involving these elements.

When introducing user-defined abstract data types, KeY does allow the specification of abstract data types by adding new sorts, function symbols, and inference rules. These new sorts and function symbols can be used in JML by a KeY-specific extension. A drawback is that KeY provides no guarantee that the resultant theory is *consistent*. Thus, a small error in a user-defined abstract data type specification could lead to unsound proofs. In contrast, Isabelle/HOL (Isabelle instantiated with Church's type theory) includes a definitional package for data types [36] that provides a mechanism for defining so-called *algebraic data types*, which are freely generated inductive data types: the user provides some signature consisting of constructors and their parameters, and the system automatically derives characteristic theorems, such as a recursion principle and an induction principle. Under the hood, each algebraic data type definition is associated with a Bounded Natural Functor (BNF) that admits an initial algebra [37], but for our purposes, we simply trust that the system maintains consistency.

The overall approach of integrating ADTs in KeY can be summarized by a workflow diagram, see Figure 6.1.

What is common between Isabelle/HOL and KeY are the *abstract* data types. From KeY, the underlying definition of the algebraic data type is not visible, nor are the Java-specific types visible in Isabelle/HOL. This allows us to make use of the best of both worlds: Isabelle/HOL is used as a general-purpose theorem prover, while KeY is used as a domain-specific theorem prover for showing the correctness of Java

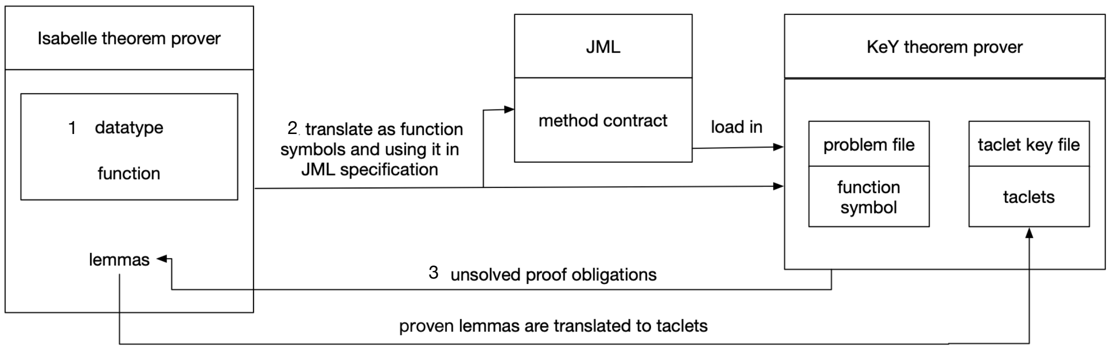

Figure 6.1: The workflow of integrating ADTs in KeY.

programs. Essentially, we will be following three steps for defining the abstract data type between the two provers:

1. We define algebraic data types and functions in Isabelle/HOL to logically model domain-specific knowledge of the Java program that we want to verify.
2. We take the signature of our data types and functions from Isabelle/HOL and add corresponding sorts and function symbols in KeY, using a type mapping for common types. Then we write specifications of the Java program in JML that makes use of the new sorts and function symbols by using a KeY-specific extension of JML.
3. We use the KeY system to perform symbolic execution of the Java program. This leads to proof obligations in which the imported symbols are uninterpreted, meaning that one is limited in reasoning about them in KeY. Sometimes, contracts in JML specify sufficient detail such that the proof obligations can already be closed in KeY. Other times, specific properties of the imported symbols are needed. At this stage, properties can be formulated that capture our expectations, and after formulating these properties in Isabelle/HOL we can prove them also in Isabelle/HOL. If we succeed in proving a lemma, that lemma is added to KeY by representing it as an inference rule called a *taclet*.

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.

Below we give more detail on each of these main steps.

Step 1. Formalizing ADTs in Isabelle/HOL. One defines data types and functions in Isabelle/HOL in the usual manner: using the **datatype** command to define a data type and the **fun** command to define functions. There are a number of caveats when working in Isabelle/HOL, to ensure a smooth transfer of the theory to KeY:

- For data types that contain Java objects, we have to work around the limitation that Java types are not available in Isabelle/HOL. We can instead introduce a polymorphic type parameter. Below we show how in our translation back to KeY, we put back the original types by instantiating the polymorphic type parameters by Java types which are available in KeY.

- Isabelle/HOL allows higher-order definitions, whereas the dynamic logic of KeY is first-order. Thus, for function symbols that we wish to import in KeY, we limit ourselves to first-order type signatures, therefore we only allow a subset of Isabelle/HOL to be imported in KeY.

As a simple example, we declare a new parameterized data type (in Isabelle type parameters, such as α, are written prefixed to the parameterized type):

$$\textbf{datatype}\ \alpha\ \textit{option} = \textit{None} \mid \textit{Some}(\alpha)$$

This data type allows us to model partially defined functions: an element of α *option* represents either 'nothing' or an element of the given type α. The definition introduces the constructors *None* : α *option* and *Some* : $\alpha \Rightarrow \alpha$ *option*. We can define functions recursively over the structure of a user-defined data type. The latter is illustrated in Section 6.3.

Step 2. Using ADTs in JML specifications. The dynamic logic underlying KeY is multi-sorted. To declare new data types and functions, we may introduce sorts and function symbols. The behavior of these function symbols is encoded as proof rules, which we formulate using an extensible formalism called *taclets* [69, 70]. Taclets in KeY are stored in plain-text files alongside the Java program sources that comprise the following blocks:

- We declare sorts corresponding to our data types in a block named `\sorts`. KeY has no parameterized sorts. So, we instantiate each type (where the type parameters are replaced by corresponding sorts provided by KeY) and introduce a sort with a suitable name for each type instantiation.
- We declare the signatures of each function in a block named `\functions`. A function signature consists of its arity and the sorts corresponding to its parameters. We erase polymorphic type parameters, by replacing them with their instantiated sorts. Also, we ensure that Isabelle/HOL's built-in types are mapped to the corresponding KeY built-in types, e.g. for **int** and **bool**.
- We add axioms to specify properties of functions in a block named `\axioms`.

Listing 6.1 shows how to represent the above data type α *option*. We have instantiated the type parameter α with the `java.lang.Object` sort.

```
\sorts { option; }
\functions { option Some(java.lang.Object); option None; ... }
\axioms { ... }
```

Listing 6.1: Declaring sorts and function symbols for new ADTs in KeY.

The new function symbols can then be used in JML specifications (such as method contracts and class invariants) by prefixing their name with `\dl_`. For example, the function symbol *None* can be referred to in a JML contract by writing it as `\dl_None`. Axioms are not (yet) needed to use our function symbols in JML specifications. Therefore, in step two of our workflow, we do not specify any axioms. We describe adding axioms in more detail in step three below.

Step 3. Using the imported ADTs during verification. We now focus on using the new ADTs in proofs of Java programs with KeY. When one starts proving that a Java program satisfies its JML specification and that specification contains function symbols as above (prefixed with `\dl_`), KeY treats these as uninterpreted symbols (with unknown behavior, other than their signature). In other words: without adding any axioms, only facts about general predicates and functions that are universally valid can be used in KeY proofs. Typically this is insufficient to complete the proof: one needs specific properties that follow from the underlying definition in Isabelle/HOL.

There are two ways of "importing" such properties in KeY. The first way is to specify expected properties in JML contracts (e.g. preconditions, postconditions, invariants) where the data type is used: this defers the moment in which the expected properties are actually proved, e.g. if used in the contracts for interface methods. The second way is to "import" such properties about the behavior of user-defined functions into KeY by defining inference rules in the `axioms` block. These rules allow the inference of properties that KeY can not derive from any other inference rules. By combining these two ways, the human-proof engineer has some flexibility when the proofs of specific properties are done.

We leverage Isabelle/HOL to prove the soundness and consistency of the imported axioms. In essence, this provides a way to use Isabelle/HOL as an interactive back-end to KeY. Our workflow supports a lazy approach that minimizes the amount of work: we only add axioms about functions *when they are necessary*, i.e., when we are stuck in a proof situation that requires more knowledge of the function behavior.

Let us consider a simple concrete example that illustrates the above concepts. Suppose we have a proof obligation in KeY in which $\mathit{Some}(o) = \mathit{None}$ appears as an assumption (it occurs as an antecedent of an open goal, and to discharge this proof obligation it is sufficient to show this assumption leads to a contradiction). We need to show that if there is some object o, then $\mathit{Some}(o) \neq \mathit{None}$. KeY can not proceed in proving this goal without any axioms because *Some* and *None* are uninterpreted symbols in KeY. We thus formulate in Isabelle/HOL, abstracting from the particular sorts as they appear in KeY, the following lemma

$$\textbf{lemma}\ \mathit{option\_distinct} :: \mathit{Some}(o) \neq \mathit{None}$$

which we easily verified (in Isabelle) using a characteristic theorem of α *option*.

```
\axioms {
  option_distinct {
    \schemaVar \term java.lang.Object o1;
    \find(Some(o1) = None)
     \replacewith(false)
  };
}
```

Listing 6.2: Adding a taclet to KeY that expresses the distinctness of constructors.

Our next objective is to import this lemma to KeY to make it available during the proving process. We do this by formulating the lemma as a taclet in the block `axioms`, as can be seen in Listing 6.2.

This taclet states that the name of the inference rule is `option_distinct`. The keyword `find` states to which expression or formula the rule can be applied (on either side of the sequent). The placeholder symbols, called schema variables, are used to stand for, in this case, the argument of the *Some* function. The placeholders are instantiated when the inference rule is applied in a concrete proof. The keyword `replacewith` states that the expression or formula in the `find` clause to which the rules is applied, is replaced after application by a new expression or formula (which in this case is the formula `false`) in the resulting sequent. One may also express side conditions on other formulas that need to be present in the sequent with the clause `assumes` (as shown in Listing 6.13 later on).

Another example shown below expresses the injectivity of the function `Some`. This lemma can also be verified using the characteristic theorems of the data type.

$$\textbf{lemma}\ \mathit{Some\_injective} :: \mathit{Some}(a) = \mathit{Some}(b) \leftrightarrow a = b$$

We can express this injectivity rule by using the `find` clause with the expression $\mathit{Some}(o1) = \mathit{Some}(o2)$, and use $o1 = o2$ as the `replacewith` clause. A taclet that uses `find` and `replacewith` on formulas corresponds to a logical equivalence in Isabelle/HOL, since the formula can appear either as an antecedent or a succedent in a sequent in KeY. A full exposition of the taclet language is out of the scope of this thesis, we instead refer to the KeY book [23].

6.3 History-based specification

As a particular case study of working with abstract data types in KeY, we will employ ADTs to support history-based reasoning [53]. In this section, we will motivate our approach, and give specifications of the `Collection` interface in terms of histories. In Section 6.4, we will illustrate the use of these specification in the verification of the correctness of clients of the `Collection` interface.

Listing 6.3 shows some of the main methods of the `Collection` interface. We want to give a specification of these methods, which formalizes the informal Javadoc documentation [71], by means of preconditions and postconditions using JML. As already pointed out in the introduction, such a JML specification is intrinsically state-based, describing properties of instance variables. But interfaces abstract from any information about instance variables because these expose details about the underlying implementation.

Existing approaches model the general properties of a collection using model fields in JML [10, 11]. However, there are two main methodological problems with using model fields: first, adding model fields to an interface is *ad hoc*, e.g., they capture specific properties, and, second, model fields denote locations on the heap and thus require (dynamic) frame conditions (see e.g. [54]) for each method of the interface. From a client perspective, however, what is only observable about any implementation of the `Collection` interface is the sequence of calls and returns of the methods of the `Collection` interface. This sequence of events is also called the history of the instance of the interface. Therefore in our approach, the methods of a collection are formally specified by mathematical relations between user-defined abstractions of such sequences. Histories thus can be viewed to constitute the canonical abstract state space of an interface [22, 53]: by modeling the interface using its history, we no longer need ad hoc abstractions at the level of the interface. Further, since histories are modeled using ADTs of which elements are not stored on the heap, we do not have to specify frame conditions when reasoning about general properties of histories.

All implementations of the `Collection` interface have certain constraints on sequences of method calls and returns in common, which characterize valid behavior. These constraints are formalized as pre- and postcondition specifications of the interface methods. In fact, the signature of the methods of the `Collection` interface has been designed to allow for the expression of such constraints, e.g., the Boolean value returned by the `add` method, according to the informal documentation, expresses whether the specified element has been added:

> `boolean add(Object o)`
> Ensures that this collection contains the specified element. 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. ... [A] collection always contains the specified element after this call returns [normally]. [71]

Whether the element is actually added to the `Collection` is thus, in some cases, left to the underlying implementation to decide. However, we can still infer from a sequence of calls of `add` and `remove` and their corresponding returns what is the *content* of the `Collection`, abstracting from the underlying implementation.

The Java Collection Framework has a behavioral subtype hierarchy [28]. Here, `Collection` is the topmost type, that has two subtypes `List` and `Set`. These two subtypes are incompatible: no set can be considered a list. As we shall see in the next subsection, it is quite surprising that we can make use of multisets to formally capture the content of a collection, since in algebraically specified data types multiset is a subtype of list and a supertype of set.

```
public interface Collection {
    boolean add(Object o);
    boolean addAll(Collection c);
    boolean remove(Object o);
    boolean contains(Object o);
    boolean isEmpty();
    Iterator iterator();
    ...
}
```

Listing 6.3: The `Collection` interface.

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Listing 6.4: The `Iterator` interface.

To formalize in Isabelle/HOL sequences of calls and returns of the methods of the `Collection` interface, we introduce for each method definition a corresponding constructor in the following parameterized data type:

$$\textbf{datatype}\ (\alpha, \beta, \gamma)\ \mathit{event} = \mathit{Add}(\alpha, \textbf{bool}) \mid \mathit{AddAll}(\gamma, \alpha\ \mathit{elemlist}) \mid$$
$$\mathit{Remove}(\alpha, \textbf{bool}) \mid \mathit{Iterator}(\beta) \mid \mathit{IteratorNext}(\beta, \alpha) \mid \mathit{IteratorRemove}(\beta) \mid ...$$

The type parameters α, β and γ correspond to (type abstractions of) the Java types `Object`, `Iterator`, and `Collection`, respectively. In general, events specify both the actual parameters and the return value (the last argument of the event) of a call of the specified method. For simplicity, we focus here only on the essential methods of the collection interface, but without much difficulty, all other methods can be added too. For technical convenience, only normal returns from method calls are considered events. The limitation of this is that some programs rely on thrown exceptions, and may exhibit different method behavior based on past method calls that throw exceptions. With extra work, this restriction can be lifted by also considering additional events corresponding to method calls that do not return normally, e.g. by recording the exception that is thrown instead of the return value.

Note that in our definition above, one event is special: namely, the one that corresponds with calls to the `addAll` method, which, roughly, adds all the elements of the argument collection [71]:

> `boolean addAll(Collection c)`
> Adds all of the elements in the specified collection to this collection. 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.) The parameter `c` is the collection containing elements to be added to this collection. Returns true if this collection changed as a result of the call.

The problem here is that the Boolean return value only indicates that the underlying collection has been modified. This information does not suffice to infer from a sequence of events the contents of the underlying collection: the informal specification that in this case *all* elements have been added is ambiguous in that it does not take into account the possible underlying implementation of the receiving collection, e.g., what happens if you want to add all elements of a *list* with duplicates to a *set*? In our formalization, the `addAll` event returns a selection that is consistent with the type of the receiving collection. This selection is represented by the α *elemlist* type which denotes lists of pairs of elements of type α and a Boolean value. Intuitively, instances of this type represent the contents of the argument filtered by the receiving collection, where each Boolean is a status flag whether the paired element is considered to be included or not.

Note that this return type is a *refinement* of the Boolean returned by the `addAll` method, which returns true if and only if the element list contains a pair $(o, \textbf{true})$, for some object o. The requirement that the first component of the pairs in such a list corresponds to the content of the added collection will be stated in the contract of the `addAll` method (see the next section). The α *elemlist* data type is defined as follows:

$$\textbf{datatype}\ \alpha\ \mathit{elemlist} = \mathit{Nil} \mid \mathit{Cons}(\alpha, \textbf{bool}, \alpha\ \mathit{elemlist}).$$

It introduces a polymorphic type, a constant $\mathit{Nil} : \alpha\ \mathit{elemlist}$ and a 3-ary function symbol $\mathit{Cons} : \alpha \times \textbf{bool} \times \alpha\ \mathit{elemlist} \Rightarrow \alpha\ \mathit{elemlist}$. The use of the names *Nil* and *Cons* is standard for sequences.

An iterator provides a view of the elements that the collection contains. Iterators are obtained by calling the `iterator` method of the `Collection` interface. This method returns an object of a so-called inner class (which implements the `Iterator` interface) of the surrounding collection. Objects of inner classes have access to the internal state of the surrounding class. Iterator objects exploit this property to access the elements of the collection. It is possible to obtain multiple iterators, each with their own local view on a collection. Thus, we model iterators as sub-objects of their owning collection: method calls to sub-objects are registered in the history of the associated owning object. The methods of the iterator interface are represented by corresponding events, e.g., $\mathit{IteratorNext}(\beta, \alpha)$ and $\mathit{IteratorRemove}(\beta)$ represent the `Iterator#next()` and `Iterator#remove()` methods of the iterator β, respectively. As a sequence of events, the history of a collection, as defined below, thus includes the calls and returns of the methods of its iterators.

Finally, we introduce the type history as a recursive datatype:

$$\textbf{datatype}\ (\alpha, \beta, \gamma)\ \mathit{history} = \mathit{Empty} \mid \mathit{Event}((\alpha, \beta, \gamma)\ \mathit{event}, (\alpha, \beta, \gamma)\ \mathit{history})$$

As above, the type parameters α, β and γ correspond to (type abstractions of) the Java types `Object`, `Iterator` and `Collection`. Here the data type *history* uses the constructors *Empty* and *Event*: either the history is empty, or it consists of an event at its head and another history as its tail. To add a new event to an old history, the new event will become the head in front and the old history will be its tail.

6.3.1 History abstractions

Abstractions of a history are used to map the history to a particular value. Instead of dealing with a specific history representation, we use abstractions to reason about histories. Since clients of an interface are oblivious to the implementation of the interface, clients cannot know the exact events that comprise a history, only the value of our abstractions. In this sense, we could consider two histories observationally equivalent whenever the value of all our abstractions is the same. Since the contracts are specified in JML, and the verification of clients is based on those contracts, client verification can be done within KeY by leaving histories uninterpreted, thus at the level of KeY one cannot know the internal structure of the history. From this point of view, it is fair to say that we use *abstract* data types on the specification level in JML, and use *algebraic* data types in Isabelle/HOL with a fixed representation to realize the abstract type.

The abstraction *multiset* can be recursively defined to compute the multiplicity of an object given a particular history. Intuitively it represents the ‘contents’ of a collection at a particular instant.

$$\begin{aligned}
&\textbf{fun } \mathit{multiset} : (\alpha, \beta, \gamma)\ \mathit{history} \times \alpha \Rightarrow \textbf{int}\\
&\quad \mathit{multiset}(\mathit{Empty}, x) = 0\\
&\quad \mathit{multiset}(\mathit{Event}(\mathit{Add}(y, b), h), x) = \mathit{multiset}(h, x) + (x = y \wedge b\ ?\ 1 : 0)\\
&\quad \mathit{multiset}(\mathit{Event}(\mathit{AddAll}(y, xs), h), x) = \mathit{multiset}(h, x) + \mathit{multisetEl}(xs, x)\\
&\quad \mathit{multiset}(\mathit{Event}(\mathit{Remove}(y, b), h), x) = \mathit{multiset}(h, x) - (x = y \wedge b\ ?\ 1 : 0)\\
&\quad \mathit{multiset}(\mathit{Event}(\mathit{IteratorRemove}(i), h), x) =\\
&\qquad\qquad \mathit{multiset}(h, x) - (\mathit{last}(h, i) = \mathit{Some}(x)\ ?\ 1 : 0)\\
&\quad \mathit{multiset}(\mathit{Event}(e, h), x) = \mathit{multiset}(h, x)
\end{aligned}$$

and e is any event not specified above leave the *multiset* unchanged.

The function *multisetEl* is defined as follows: given an element list and an element, it computes the multiplicity of pairings of that element with **true**, intuitively representing the 'contents' of a filtered sequence.

$$\begin{aligned}
&\textbf{fun } \mathit{multisetEl} : \alpha\ \mathit{elemlist} \times \alpha \Rightarrow \textbf{int}\\
&\quad \mathit{multisetEl}(\mathit{Nil}, x) = 0\\
&\quad \mathit{multisetEl}(\mathit{Cons}((y, b), t), x) = \mathit{multisetEl}(t, x) + (x = y \wedge b\ ?\ 1 : 0)
\end{aligned}$$

Similarly, *occurs* is defined as follows: given an element list, it computes the multiplicity of elements occurring on the left in each pair that is in the element list, regardless of the Boolean status flag.

A call to the `iterator()` method should return a new iterator sub-object. We use the abstraction *iterators* to collect all previously returned iterators and store them in a set. If we are to ensure that a new iterator is returned then the newly created iterator must not be in this set.

The `Iterator#remove()` method does not carry any arguments from which we can infer what element of the collection is to be removed: this element is only retrieved by searching the past history. Each iterator sub-object can be associated with an element that it has returned by a previous call to its `next()` method (if it exists). To that end, we define the partial function *last* below:

$$\begin{aligned}
&\textbf{fun } \mathit{last} : (\alpha, \beta, \gamma)\ \mathit{history} \times \beta \Rightarrow \alpha\ \mathit{option}\\
&\quad \mathit{last}(\mathit{Empty}, i) = \mathit{None}\\
&\quad \mathit{last}(\mathit{Event}(\mathit{IteratorNext}(j, x), h), i) = (i = j\ ?\ \mathit{Some}(x) : \mathit{last}(h, i))\\
&\quad \mathit{last}(\mathit{Event}(e, h), i) = (\mathit{modify}(e)\ ?\ \mathit{None} : \mathit{last}(h, i))
\end{aligned}$$

where in the final clause e is any event different from *IteratorNext*.

We use the α *option* type to model this as a partial function, because not all iterators have a last element (e.g., a newly created iterator). We cannot use **null**, since a collection could contain such objects and that reference is not available in our Isabelle theory. We also define the *modify* abstraction recursively: it is **true** for those and only those events that represent a modification of the collection (e.g. successfully adding or removing elements).

The abstraction *visited* tracks the multiplicities of the elements already seen. Intuitively, a call on method `Iterator#next()` will increase the *visited* multiplicity of the returned object by one and leave all other element multiplicities the same. We also define *size* that takes a history and gives the number of elements contained by the collection, *iteratorSize* of a history and an iterator which computes the total number of elements already seen by the iterator, and the attribute *objects* that collects all elements that occur in the history in a set. The abstraction *hasNext* models the outcome of the `Iterator#hasNext()` method. That method returns true if and only if the iterator has a next element. If the iterator has not yet seen all elements that are contained in its owner, it must have a next element that can be retrieved by a call to `Iterator#next()`. We define *hasNext* to be true if and only if *iteratorSize* is less than *size*.

What happens when using an iterator if the collection it was obtained from is modified after the creation of the iterator? A `ConcurrentModificationException` is thrown in practice. To ensure that the iterator methods are only called when the backing collection is not modified in the meantime, we introduce the notion of validity of an iterator as below. If the backing collection is modified, all iterators associated with that collection will be invalidated.

We introduce the following abstraction:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{isIteratorValid} : (\alpha,\beta,\gamma)\ \mathit{history} \times \beta \Rightarrow \textbf{bool} \\
\quad \mathit{isIteratorValid}(\mathit{Empty}, i) \leftrightarrow \textbf{false} \\
\quad \mathit{isIteratorValid}(\mathit{Event}(\mathit{Iterator}(y), h), i) \leftrightarrow \\
\qquad\qquad (y = i\ ?\ \textbf{true} : \mathit{isIteratorValid}(h, i)) \\
\quad \mathit{isIteratorValid}(\mathit{Event}(\mathit{IteratorNext}(y, x), h), i) \leftrightarrow \\
\qquad\qquad ((y = i \rightarrow \mathit{hasNext}(h, y)) \wedge \\
\qquad\qquad (\mathit{visited}(h, y, x) < \mathit{multiset}(h, x)) \wedge \mathit{isIteratorValid}(h, i)) \\
\quad \mathit{isIteratorValid}(\mathit{Event}(\mathit{IteratorRemove}(y), h), i) \leftrightarrow \\
\qquad\qquad ((y = i) \wedge (\exists w.\ \mathit{last}(h, y) = \mathit{Some}(w) \wedge \\
\qquad\qquad (0 < \mathit{visited}(h, y, w))) \wedge \mathit{isIteratorValid}(h, i)) \\
\quad \mathit{isIteratorValid}(\mathit{Event}(e, h), i) = (\neg \mathit{modify}(e)\ \wedge \mathit{isIteratorValid}(h, i))
\end{array}
$$

where in the last clause, again e is any event not specified above: for those events we first check if the collection was modified then we leave *isIteratorValid* the same as for its tail. Note that calling the `Iterator#remove()` method invalidates all other iterators, but leaves the iterator on which that method was called valid.

Finally, the abstraction *isValid* is a global invariant of the `Collection` interface and is used only in Isabelle/HOL. We say a history is valid if all the conditions on the history as specified by the method contracts are satisfied (see next section). The sort of histories that are imported in KeY comprises only the valid histories, i.e. the subtype of histories for which this global invariant holds. Validity of histories is defined recursively over the history data type as follows (but we only focus on the definition of validity for the most important events, for the full definition we refer the reader to the artifact [72]):

```
fun $isValid : (\alpha, \beta, \gamma)\ history \Rightarrow \mathbf{bool}$
    $isValid(Empty) \leftrightarrow \mathbf{true}$
    $isValid(Event(Add(y, b), h)) \leftrightarrow (multiset(h, y) = 0 \rightarrow b) \wedge isValid(h)$
    $isValid(Event(AddAll(xs, b), h)) \leftrightarrow$
        $(\forall y.\ multiset(h, y) = 0 \rightarrow multisetEl(xs, y) > 0) \wedge$
        $(b \leftrightarrow \exists y.\ multisetEl(xs, y) > 0) \wedge isValid(h)$
    $isValid(Event(Remove(y, b), h)) \leftrightarrow (b \leftrightarrow multiset(h, y) > 0) \wedge isValid(h)$
    $isValid(Event(Iterator(x), h)) \leftrightarrow x \notin iterators(h) \wedge isValid(h)$
    $isValid(Event(IteratorNext(x, y), h)) \leftrightarrow x \in iterators(h) \wedge$
        $isIteratorValid(Event(IteratorNext(x, y), h)) \wedge isValid(h)$
    $isValid(Event(IteratorRemove(x), h)) \leftrightarrow x \in iterators(h) \wedge$
        $isIteratorValid(Event(IteratorRemove(x), h)) \wedge isValid(h)$
```

Intuitively, the clauses of the *isValid* predicate capture the following conditions which are based on the Javadoc descriptions:

- *Add*: If one adds an element to the receiver, it must return true if it was not yet contained before.
- *AddAll*: All elements of the argument that are not contained in the receiver should be added, and the return value must be true whenever one such add succeeds.
- *Remove*: An element is removed (the return value must be true) if and only if it was contained.
- *Iterator*: The returned iterator sub-object is an object that is not returned by a previous call to `Collection#iterator()`.
- *IteratorNext*: The method is only called on sub-objects returned before by a previous call to `Collection#iterator()`, and the iterator should remain valid. By definition of *isIteratorValid*, we also know that it implies the attribute *isIteratorValid*(h), i.e. that the iterator must be valid before the method `Iterator#next()` is called.
- *IteratorRemove*: Similar to above.

6.3.2 Method contracts of `Collection`

We are now able to formulate method contracts of the methods of the interface, making use of histories and abstractions. Every instance of the `Collection` interface has an associated history, which we specify by using a *model method* in JML, as shown in Listing 6.5. The model method has as a return type the sort corresponding to the histories we defined earlier in Isabelle/HOL. We also specify the owner of an iterator using a model method, see Listing 6.6. This allows us to refer to the history of the owning collection in the specification of methods of the iterator.

```
public interface Collection {
/*@ model_behavior
  @ requires true;
  @ model history history();
  @*/
...
}
```

Listing 6.5: The `history()` model method in JML.

```
public interface Iterator {
/*@ model_behavior
  @ requires true;
  @ model Collection owner();
  @*/
...
}
```

Listing 6.6: The `owner()` model method in JML.

The `history()` model method here returns an element of an abstract data type: these elements are independent of the heap, meaning that heap modifications do not affect the value returned by the model method before the heap modifications took place, thus eliminating the need to apply dependency contracts for lifting abstractions of the history to updated heaps as was required in the EHB approach [53].

As a guiding principle, our contracts are specified in terms of the history abstractions only. This principle ensures that interfaces are specified up to observational equivalence, thus leaving more room on the side of an implementor of an interface to make choices on how to implement a method. For example, the `add` method can be implemented in terms of calling the `addAll` method of the same implementation supplied with a singleton collection wrapping the argument. Another example would be implementing the `addAll` method by iterating over the supplied collection and for each object calling the `add` method of the same implementation.

Method contract of the `add()` method.

We have specified this method in terms of the *multiset* of the new history (after the method call) and the old history (prior to the method call, referred to in the postcondition with `\old`).

```
/** 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 \dl_multiset(history(),o) ==
      \dl_multiset(\old(history()), o) + ((\result == true) ? 1 : 0);
    @ ensures (\forall Object o1; o1 != o; \dl_multiset(history(),o1) ==
      \dl_multiset(\old(history()), o1));
    @ ensures \dl_multiset(history(),o) > 0;
    @ ensures \result == false ==>
      (\forall Iterator it; it.owner() == this;
14           \dl_isIteratorValid(\old(history()), it) ==>
15           \dl_isIteratorValid(history(), it));
16       @ ensures (\forall Iterator it;
17        \old(it.owner()) == this; it.owner() == this);
18       @*/
19   boolean add(Object o);
```

Listing 6.7: The specification of the `add()` method.

In Listing 7, lines 1–5 show the informal Javadoc of the `add` method [71]. The JML specification (lines 7–17) covers all information present in the Javadoc. More explanation about the specification is given below:

- *On lines 7–8*: This clause ensures that the collection contains the specified element after the `add` method call (as described in the informal Javadoc). If the collection changed as a result of the call, the result is true and the *multiset* will be incremented accordingly. Otherwise, the *multiset* will remain unchanged. Note that the value of `\result` is underspecified, leaving room for multiple implementations of the collection interface. Indeed, the difference between the refinements `List` and `Set` of the `Collection` interface makes a distinction between the behavior of `add(Object)`: lists always allow the addition of new elements, whereas sets only add unique elements. So, for the `List` interface, the `\result` is unconditionally true. For the `Set` interface, the `\result` is true if and only if the multiplicity of the object to add is zero before execution of the `add` method.
- *On lines 9–10*: For each object different from the object to be added, the multiplicity does not change. The Javadoc does not explicitly cover this. However, this makes more precise how the collection may change by the call: no other objects may be added, other than the one in the parameter.
- *On line 11*: The call to the `add` method guarantees that the multiplicity of the object to add is positive. This formalizes the informal Javadoc property that the collection will contain the specified element after returning.

The last two postconditions in the contract of `add` are not related to the Javadoc description, but rather specify two properties related to our formalization of iterators as sub-objects. On lines 12–15, the specification is a direct translation of the *isIteratorValid* definition in Isabelle/HOL. If the collection remains unchanged, all iterators related to the collection are still valid, otherwise, the iterators will be invalidated due to the successful adding of elements to the collection. On lines 16–17, it specified that a call to the `add` method does not affect ownership of iterators of the collection.

Method contract of the `addAll()` method.

Consider modeling the `addAll()` method: how can we represent an invocation of this method in a history? We can not simply record the argument instance, since that instance may be modified over time. Could we instead take a snapshot of its history, and embed that in the event corresponding to `addAll`? No, it turns out that such a nested history snapshot leads to difficulty in defining the *multiset* function that represents the contents of a collection: the receiver of the `addAll` method, being a concrete implementation, is underspecified at the level of `Collection`. A snapshot of the history of the argument merely allows us to retrieve the contents of the argument at that time, but not how the receiving collection deals with those individual elements.

Listing 6.8 shows the interface specification of the `addAll` method. Lines 1–7 show the informal Javadoc of the `addAll` method. Lines 8–23 show the postconditions of the `addAll` method.

```
 1  /** Adds all of the elements in the specified collection to this
 2     * collection (optional operation).
 3     * The behavior of this operation is undefined if the specified
 4     * collection is modified while the operation is in progress.
 5     * This implies that the behavior of this call is undefined if the
 6     * specified collection is this collection, and this collection is
 7     * nonempty. **/
 8     * @ ensures (\exists elemlist el;
 9          (\forall Object o;
10            \dl_occurs(el,o) == \dl_multiset(c.history(),o) &&
11            \dl_multiset(history(),o) ==
12            \dl_multiset(\old(history()),o) + \dl_multisetEl(el,o)));
13      @ ensures (\forall Object o;
14        \dl_multiset(c.history(),o) == \dl_multiset(\old(c.history()),o));
15      @ ensures (\forall Object o;
16       \dl_multiset(c.history(),o) > 0 ==>\dl_multiset(history(),o) > 0);
17      @ ensures \result == false ==>
18       (\forall Iterator it; it.owner() == this;
19       \dl_isIteratorValid(\old(history()), it) ==>
20       \dl_isIteratorValid(history(), it));
21      @ ensures (\forall Iterator it;
22       \old(it.owner()) == this; it.owner() == this);
23      @*/
24  boolean addAll(Collection c);
```

Listing 6.8: The use of *multiset* and *elemlist* in the specification of `addAll`.

- *On lines 8–12*: The ensures clause shows how the multiplicities of elements of the argument collection are related to that of the receiving collection. Here, `\dl_multiset(c.history(),o)` and `\dl_multiset(history(),o)`, defined above, denote the multiplicity of an element *o* in the argument and receiving collection, respectively. The list *el* associates a status flag with each occurrence of an element of the argument collection. This flag indicates whether the *receiving* collection's implementation actually *does* add the supplied element (e.g., a `Set` filters out duplicate objects but a `List` does not). Consequently, the multiplicity of the elements of the receiving collection is updated by how many times the object is actually added, denoted by `\dl_multisetEl(el,o)` (also defined above). The existential quantification of this list allows both for abstraction from the particular enumeration order of the argument collection and the implementation of the receiving collection as specified by the association of the Boolean values.

- *On lines 13–14*: The multiplicity of the elements of the argument collection will not change due to this method call. The Javadoc does not explicitly state this, but this property is needed to reason about unchanged contents of the supplied argument collection.
- *On lines 15–16*: If there are some objects in the argument collection that are not yet added to this collection, then the multiplicity of those objects must be positive after the method returns. This formalizes the informal Javadoc that all of the elements in the specified collection need to be added to this collection.

The postconditions on lines 17–20 and lines 21–22 have the same meaning as the last two postconditions of the `add` method. On lines 17–20, the specification is a direct translation of the *isIteratorValid* definition in Isabelle/HOL. If the collection remains unchanged, all iterators related to the collection are still valid, otherwise, the iterators will be invalidated due to the successful adding of elements to the collection. On lines 21–22, it specified that a call to the `add` method does not affect ownership of iterators of the collection.

Method contract of the `Iterator#remove()` method.

Next, we consider the following use case: iterating over the elements of a collection. The question arises: what happens when using an iterator when the collection it was obtained from is modified after its creation? In practice, an exception named `ConcurrentModificationException` is thrown. To ensure that the iterator methods are only called when the backing collection is not modified in the meantime, we introduce the notion of the validity of an iterator. As already discussed above, we record the events of the iterators in the history of the owning collection, alongside other events that signal whether that collection is modified, so that indeed we *can* define a recursive function that determines whether an iterator is still valid. Another complex feature of the iterator is that it provides a parameterless `Iterator#remove()` method, producing no return value. Its intended semantics is to delete from the backing collection the element that was returned by a previous call to `Iterator#next()`, and invalidate all other iterators.

The specification of this method is illustrated in Listing 6.9.

```
1  /** Removes from the underlying collection the last element returned by
2    * this iterator (optional operation).
3    * This method can be called only once per call to next().
4    * The behavior of an iterator is unspecified if the underlying
5    * collection is modified while the iteration is in progress in any way
6    * other than by calling this method. **/
7   /* @ ...
8      @ requires \dl_last(owner().history(),this) != \dl_None;
9      @ ensures (\exists Object o;
10             \dl_last(\old(owner().history()),this) == \dl_Some(o);
11             \dl_multiset(owner().history(),o) ==
12             \dl_multiset(\old(owner().history()),o) - 1);
13       @ ensures (\exists Object o;
14           \dl_last(\old(owner().history()),this) == \dl_Some(o);
15           (\forall Object o1; o1 != o;
16               \dl_multiset(owner().history(),o1) ==
17               \dl_multiset(\old(owner().history()),o1)));
18       @*/
19 void remove();
```

Listing 6.9: Part of the specification of the `remove` method on `Iterator`.

- *On line 8*: Here we use the *last* property to capture the return value of a previous call to `Iterator#next()`. This formalizes the informal Javadoc that the `remove()` method can be called only once per call to `next()`: after the remove method returns, the *last* property gives back *None* as can be seen from the definition in the previous section. Thus calling `remove()` twice after one call to `next()` is not allowed.
- *On lines 9–12*: The object that was last returned by `next()` is removed from the owning collection.
- *On lines 13–17*: This postcondition is not explicitly covered by the informal Javadoc, but this specifies that no other object may be removed, other than the object that was returned by the previous call to `next`.

For the full Isabelle/HOL theory and method contracts of our case study, we refer the reader to the artifact accompanying this paper [72]. This artifact includes the translation of the theory to a signature that can be loaded in KeY (version 2.8.0) so that its function symbols are available in the JML specifications we formulated for `Collection` and `Iterator`. It also includes the taclets we imported from Isabelle, which we used to close the proof obligations generated by KeY.

6.4 History-based client-side verification

In this section, we will describe several case studies that we perform to show the feasibility and usability of our history-based reasoning approach supported by ADTs. Section 6.4.1 provides an example that we have verified with both the EHB approach [53] and the LHB approach described in this paper. This case supports our claim that the LHB approach yields a significant improvement in the total proof effort when compared to the EHB approach. As such, we are now able to verify more complex examples: the examples in Section 6.4.2 demonstrate reasoning about iterators, and, advancing further, we will verify binary methods in Section 6.4.3. Finally, proof statistics for all case studies are in Section 6.4.4.

We focus in this paper on the verification of client-side programs. Clients of an interface are, in principle, oblivious to the implementation of the interface. Hence, every property that we verify of a client of an interface should hold for any correct implementation of that interface.

6.4.1 Significant improvement in proof effort

Using ADTs instead of encoding histories as Java objects results in significantly lower effort in defining functions for use in contracts and giving correctness proofs. This can be best seen by revisiting an example of our EHB work [53] and comparing it to the proof effort required in the LHB approach using ADTs.

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

Listing 6.10: Adding an object and if successful removing it again, leaves the contents of a `Collection` the same.

The client code and its contract are given in Listing 6.10, which has the same contract as in previous work, except we now use the imported functions we have defined in Isabelle instead of using pure methods and their dependency contracts.

In both the previous and current work, we specify the behavior of the client by ensuring that the 'contents' of the collection remain unmodified: we do so in terms of the multiset of the old history and the new history (after the `add_remove` method). During verification, we make use of the contracts of methods `add(Object)` and `remove(Object)`. These contracts specify their method behavior also in terms of the old and new history, relative to each call. Let h be the old history (before the call) and h' be the new history (after the call). Let y be the argument, the `remove` method contract specifies that $\mathit{multiset}(h', y) = \mathit{multiset}(h, y) - 1$ if the return value was **true**, and $\mathit{multiset}(h', y) = \mathit{multiset}(h, y)$ otherwise. Further, it ensures the return value is **true** if $\mathit{multiset}(h, y) > 0$. Also, $\mathit{multiset}(h', x) = \mathit{multiset}(h, x)$ holds for any object $x \neq y$. In similar terms, a contract is given for `add` that specifies that the multiplicity of the argument is increased by one, in the case that **true** is returned, and that regardless of the return value the multiplicity of the argument is positive after `add`.

We need to show that the multiplicity of the object y after the `add` method and the `remove` method is the same as before executing both methods. At this point, we can see a clear difference in the verification effort required between the two approaches. In the EHB approach, multiplicities are computed by a pure Java method `Multiset` that operates on an encoding of the history that lives on the heap. Since Java methods may diverge or use non-deterministic features, we need to show that the pure method behaves as a function: it terminates and is deterministic. Moreover, since we deal with the effects of the heap, we also need to show that the computation of this pure method is not affected by calls of `add` or `remove`, which requires the use of an accessibility clause of the multiset method.

To make this explicit, Listing 6.11 shows a concrete example of a proof obligation from KeY that arose in the EHB approach.

```
...
History.Multiset(h,y)@heap2 + 1 = History.Multiset(h,y)@heap1,
History.Multiset(h,y)@heap1 = History.Multiset(h,y)@heap + 1,
...
==>
History.Multiset(h,y)@heap2 = History.Multiset(h@heap,y)@heap2
```

Listing 6.11: Simplified proof obligation with histories as Java objects showing evaluation of the multiset function as a pure (Java) method in various heaps.

Informally, the proof obligation states that we must establish that the multiplicity of `y` after adding and removing object `y` (resulting in the heap named `heap2`) is equal to the multiplicity of `y` before both methods were executed (in the heap named `heap`). So we have to perform proof steps relating the result/behavior of the multiset method in different heaps. In practice, heap terms may grow very large (i.e. in a different, previous case study [73] we encountered heap terms that were several pages long) which further complicates reasoning.

By contrast, in the LHB approach of this paper, we model *multiset* as a function without any dependency on the heap, and so we do not have to perform proof steps to relate the behavior of *multiset* in different heaps (the interpretation of *multiset* is fixed and does not change if the heap is modified). While the arguments of *multiset* may still depend on the heap (such as the history associated with an interface that lives on the heap), when we evaluate the argument to a particular value (such as an element of the history ADT) the behavior of the *multiset* function when given such values do not depend on the heap.[1] Moreover, by defining the function in Isabelle/HOL, we make use of its facilities to show that the function is well-defined (terminating and deterministic). These properties are verified fully automatically in Isabelle: contrary to the proofs of the same properties given in KeY in the EHB approach. Thus, the LHB approach significantly reduces the total verification effort required.

More specifically, the proof statistics that show how to verify the `Multiset` pure method is terminating and deterministic and satisfies its equational specification in our EHB approach is shown in Table 6.1. This (partially manual) effort in KeY is eliminated in the LHB approach since the proof can be done automatically using Isabelle/HOL: these properties follow automatically from the function definition and the characteristic theorems of the underlying data type definitions.

Furthermore, comparing the verification of the `add_remove` method in both approaches, it can be immediately seen that we no longer have to apply any dependency contract in the LHB approach. The EHB approach was studied in the context of a simpler definition for histories (without modeling the `addAll` event), thus favoring the LHB approach even more. Moreover, the proof obligations involving the function

[1]This can be compared to the expression $x + y$ in Java where x and y are fields: the value of x and y depends on the heap but the meaning of the '+' operation does not.

symbol *multiset* can be resolved using the contracts of the methods `add(Object)` and `remove(Object)` only since these contracts specify that the multiplicity of the argument is first increased by one and then decreased by one. Thus, this example client can be verified without importing any lemma from Isabelle/HOL.

Name	Nodes	Branches	I.step	Q.inst	Contract	Dep.	Inv.	Time
Multiset	54,857	1,053	52	476	39	0	0	72 min

Table 6.1: Proof statistics of verifying termination, determinacy, and equational specification of the `Multiset` pure method in the EHB approach. The required effort for a single pure method is large.

6.4.2 Reasoning about `Iterator`

In this subsection, we will illustrate the benefits of our LHB approach in the verification of client-side examples that work with iterators. We model iterators as sub-objects so that their history is recorded by the associated owning collection. As we discussed above, iterators require special treatment because their behavior relies on the history of other objects, in our case the enclosing collection that owns the iterator.

In the EHB approach [53], we did verify a client (shown in Listing 6.12) of iterator and showed its termination: but we did not verify the pure methods (termination, determinism, equational specification) used in the specification that modeled the behavior of iterators. The EHB approach was not practical in this respect, since we need many abstractions: such as *size*, *iteratorSize*, *isValid*, *isIteratorValid* and its supporting functions *last*, *hasNext*, and *visited*. The large number of abstractions needed to model the behavior of iterators shows a verification bottleneck we encountered in the EHB approach: modeling these as pure methods and verifying their properties takes roughly the same effort as required for *multiset*, per function! In the LHB approach, we have defined these abstractions in Isabelle/HOL, and thus eliminated the need to show termination, and determinism and that they satisfy their equational specification within KeY.

```
public static void iter_only(Collection x) {
    Iterator it = x.iterator();
    /*@ ...
      @ decreasing \dl_size(it.owner().history()) −
             \dl_iteratorSize(it.owner().history(),it); @*/
    while (it.hasNext()) it.next();
}
```

Listing 6.12: Iterating over the collection. Why does it terminate?

The main term needed to show the termination of the client of iterator is given in the `decreasing` clause in JML. For the decreasing term, it has to be shown that it is strictly decreasing for each loop iteration and that it evaluates to a non-negative value in any state satisfying the loop invariant [23]. Following our workflow in Section 6.2, we are stuck in a proof situation of the verification conditions involving the decreasing term, since the function behaviors of *size* and *iteratorSize* are not defined in KeY. We thus formulate the lemma below:

$$\textbf{lemma}\ \mathit{sizeCompare} :: \mathit{isValid}(h) \Rightarrow \mathit{isIteratorValid}(h, it) \Rightarrow \\ \mathit{size}(h) \geq \mathit{iteratorSize}(h, it)$$

According to the definition of *iteratorSize*, it only adds 1 when executing the `next()` method, but the definition of *isIteratorValid* in Section 6.3.1 indicates that this method is only executed under the condition that *size* is larger than *iteratorSize*, so this lemma can be proven in Isabelle/HOL. The next step we take is translating the above lemma to a taclet named `sizeCompare` as shown in Listing 6.13. We can now apply this taclet to close the verification condition showing that the loop invariant implies that the decreasing term is not negative.

```
\axioms {
  sizeCompare {
    \schemaVar \term history h;
    \schemaVar \term Iterator it;
    \assumes(isIteratorValid(h,it) = TRUE ==>)
    \add(size(h) >= iteratorSize(h,it) ==>)
  }; }
```

Listing 6.13: Adding a taclet to KeY that expresses the relationship between *size* and *iteratorSize*.

Advancing further, we want to verify an example that modifies the backing collection through an iterator. Consider the example in Listing 6.14 that makes use of the `Iterator#remove()` method. We iterate over a given collection and at each step we remove the last returned element by the iterator from the backing collection. Thus, after completing the iteration, when there are no next elements left, we expect to be able to prove that the backing collection is now empty.

```
/*@ ...
  @ ensures \dl_size(x.history()) == 0; @*/
public static void iter_remove(Collection x) {
    Iterator it = x.iterator();
    /*@ ...
      @ loop_invariant \dl_iteratorSize(it.owner().history(),it) == 0;
      @ decreasing \dl_size(it.owner().history()); @*/
    while (it.hasNext()) {
        it.next();
        it.remove();
    }
}
```

Listing 6.14: Example 3: Iterating over the collection and removing all its elements.

This example also shows an important aspect of our LHB approach: being able to use Isabelle/HOL to derive non-trivial properties of the functions we have defined. The crucial insight here is that, after we exit the loop, we know that `hasNext()` returned **false**. Following the definition of *hasNext*, we established in Isabelle/HOL the (non-trivial) fact that a valid iterator has no next elements if and only if *iteratorSize* and *size* are equal. Following our workflow, we have proven this fact and imported it into KeY as a taclet, which is shown in Listing 6.15. Since it is a loop invariant that the size of the iterator remains zero (each time we remove an element through its iterator, it is not only removed from the backing collection but also from the elements seen by the iterator), we can thus deduce that finally, the collection must be empty.

```
HasNext_size {
\schemaVar \term history h;
\schemaVar \term Iterator it;
\assumes(isIteratorValid(h,it) = TRUE ==>)
\find(HasNext(h,it) = FALSE)
\replacewith(size(h) = iteratorSize(h,it))
};
```

Listing 6.15: Taclet for showing the equality between *size* and *iteratorSize*.

6.4.3 Reasoning about binary methods

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. 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.

As a client-side verification example, we have verified clients that operate on two collections at the same time. This is interesting, since both collections can be of a different implementation, and can potentially interfere with each other. The technique we applied here is to specify what properties remain *invariant* of histories of all other collections, e.g. that a call to a method of one collection does not change the history of any other collection. Since histories are not part of the heap, that a history remains invariant implies that all its (polymorphic) properties are invariant too. However, if a history contains some reference to an object on the heap, it can still be the case that the properties of such an object have changed.

In the example given in Listing 6.16, we make use of the `addAll` method of the collection, adding elements of one collection to another. Clearly, during the `addAll` call, the collections interfere: collection `x` could obtain an iterator of collection `y` to add all elements of `y` to itself. So, in the specification of `addAll`, we have no history invariance of `y`. Instead, we specify what properties of `y`'s history remain invariant: in this case, its multiset must remain invariant (assuming `x` and `y` are not aliases). In our example, the program first performs such `addAll` and then iterates over the collection `y` that was supplied as an argument. For each of the elements in the argument collection `y`, we check whether `x` did indeed add that element, by calling `contains`. We expect that after adding all elements, all elements must be contained. Indeed, we were able to verify this property.

```
/*@ ...
  @ ensures \result = true; @*/
public static boolean all_contains(Collection x, Collection y) {
    x.addAll(y); Iterator it = y.iterator();
    /*@ ...
      @ loop_invariant (\forall Object o1;
            \dl_multiset(y.history(),o1) > 0 ==>
            \dl_multiset(x.history(),o1) > 0); @*/
    while(it.hasNext()) {
        if (!x.contains(it.next())) { return false; }
    }
    return true;
}
```

Listing 6.16: Using the `addAll` method and checking for inclusion.

The crucial property in this verification is shown as the loop invariant: all objects that are contained in collection `y` are also contained in collection `x`. This can be verified initially: the call to iterator does not change the multisets associated with the histories of `x` and `y`, and after the `addAll` method is called this inclusion is true. But why? As already explained above, in the specification of `addAll`, we state the existence of an element list: this is an enumeration of the contents of the argument collection `y` but for each element also a Boolean flag that states whether `x` has decided to add those elements. Since this flag depends on the actual implementation of `x`, which is inaccessible to us, the contract of `addAll` existentially quantifies the element list. Thus, from the postcondition of `addAll`, for any element that was not yet contained in `x`, at least one of the pairs in the element list with that same element must have a **true** flag associated. Following from the specification of `addAll`, we can deduce that the loop invariant holds initially. From the loop invariant, we can further deduce that the `contains` method never returns **false**, so the then-branch returning **false** is unreachable. Termination of the iterator can be verified as in the previous example. Hence, the overall program returns **true**.

The last example we give is the most complex and realistic one: it is a program that compares two collections. The example involves the mutation of two collections. Two collections are considered equivalent whenever they have the same multiplicities for all elements. The example shown in Listing 6.17 performs a destructive comparison: the collections are modified in the process by removing elements. Thus, we have formulated in the contract that this method returns **true** if and only if the two collections were equivalent *before* calling the method. From this example, it is also possible to build a non-destructive comparison method by first creating a copy of the input collections, e.g. using `IdentityHashMap` (which, in recent work [74], has its correctness verified).

```
/*@ ...
  @ requires x != y;
  @ ensures \result == true <==> (\forall Object o1;
          \dl_multiset(\old(x.history()),o1) ==
            \dl_multiset(\old(y.history()),o1)); @*/
public static boolean compare_two(Collection x, Collection y) {
    Iterator it = x.iterator();
    /*@ ...
      @ loop_invariant \dl_isiteratorValid(it.owner().history(), it);
      @ loop_invariant (\forall Object o1;
              \dl_multiset(\old(x.history()),o1) ==
                \dl_multiset(\old(y.history()),o1) <==>
              \dl_multiset(x.history(),o1) ==
                \dl_multiset(y.history(),o1)); @*/
    while (it.hasNext()) {
        if (!y.remove(it.next())) { return false; }
        else { it.remove(); }
    }
    return y.isEmpty();
}
```

Listing 6.17: A realistic example of a binary method.

We assume the two collections are not aliases. The verification goes along the following lines: it is a loop invariant that the two collections were equivalent at the beginning of the method `compare_two` *if and only if* the two collections are equivalent in the current state. The invariant is trivially valid at the start of the method, and also at the start of the loop since the iterator does not change the multisets of either collection: the call on `x` explicitly specifies that `x`'s multiset values are preserved, but moreover specifies the invariance of properties of histories of any other collection (so also that of `y`). The crucial point is that a call to a method of one collection does not change the properties of other collections, such as the value of its multiset. The same holds for iterators of other collections. We specify that the history remains invariant for all other collections (and thus the history of sub-objects too) and that the owners of all iterators are preserved, as shown in Listing 6.18. These ensure clauses need to be additionally mentioned in the collection's method specifications.[1]

```
/*@ ...
  @ ensures (\forall Collection x; x != this;
      x.history() == \old(x.history()));
  @ ensures (\forall Iterator it; \old(it.owner())== it.owner());
  @*/
```

Listing 6.18: Additional specification clauses needed to prevent potential aliasing.

[1]See, in the artifact, the `LocalCollection` and `LocalIterator` interfaces.

For each element of x, we remove it from y (which does not affect the iteration over x, since the removal of an element of y specifies that the history of any other collection remains unaffected). If that fails, then there is an element in x which is not contained in y, hence x and y are not equivalent, hence they were not equivalent at the start of the program. If removal from y succeeded, we also remove the element from x through its iterator: hence x and y are equivalent if and only if they were equivalent at the start of the loop. At the end of the loop, we know x is empty (a similar argument as seen in a previous example). If y is not empty then it has (and had) more elements than x, otherwise both are empty and thus were also equivalent at the start of the program.

6.4.4 Proof statistics

The proof statistics of all the use cases discussed in this article are given in Table 6.2 below. These proofs were constructed with KeY version 2.8.0. Some of the lemmas proven in Isabelle/HOL can be done automatically, but the overall proof effort in Isabelle/HOL takes about two hours. The time estimates must be interpreted with caution: the reported time is based on the final version of all definitions and specifications and does not include the development of the theory in Isabelle/HOL or specifications in JML, and the time estimates are highly dependent on the user's experience with the tool.

Name	**Nodes**	**Branches**	**I.step**	**Q.inst**	**Contract**	**Dep.**	**Inv.**	**Time**
add_remove†	3,936	79	44	5	2	23	0	11 min
add_remove	1,514	15	12	7	2	0	0	1 min
iter_only†	8,549	58	53	0	4	12	1	15 min
iter_only	6,549	18	0	9	3	0	1	2 min
iter_remove	10,353	24	20	0	4	0	1	4 min
all_contains	23,900	94	187	40	5	0	2	40 min
compare_two	44,481	199	544	93	8	0	1	100 min

Table 6.2: Summary of proof statistics. **Nodes** and **Branches** measure the size of the proof tree, **I.step** counts the number of interactive steps performed by the user, **Q.inst** is the number of quantifier instantiations, Contract is the number of contracts applied, **Dep.** is the number of dependency contracts applied, **Loop inv.** is the number of loop invariants applied, and **Time** is the estimated time of completing the proof in the KeY theorem prover.

The rows marked † come from the EHB approach (encoding histories as Java objects [53]). The non-marked rows, i.e. the LHB approach, are part of the accompanying artifact [72]. Compared with the artifact [72] for our conference paper [75], we have simplified the contracts to make them more readable. For example, instead of adding invariant properties to all pre- and postconditions explicitly in the contracts, we now specify them as interface invariants. This requires more effort during verification, since previously verification conditions that could be automatically closed need to be proven manually (due to the limitations of KeY in its strategy of automatically unfolding partial invariants). We also provide video files (no sound!) that show a recording of the interactive proof sessions [76].

6.5 Summary

In this chapter, we showed how ADTs externally defined in Isabelle/HOL can be used in JML specifications and KeY proofs, and we applied this technique to specifying and verifying an important part of the Java Collection Framework. Our technique enables us to use Isabelle/HOL as an additional back-end for KeY, but also to enrich the specification language. We successfully applied our approach to define an ADT for histories of Java interfaces and specified core methods of the main interface of the Java Collection Framework and verified several client programs that use it. Our method is tailored to support programming to interfaces and is powerful enough to deal with binary methods and sub-objects such as iterators. Sub-objects require a notion of ownership as their behavior depends on the history of other objects, e.g. the enclosing collection and other iterators over that collection. Moreover, we specified the method `Collection#addAll(Collection)` and were able to verify client code that makes use of that method, which solved a problem left open in our previous work [5].