Integrating ADTs in KeY and their application to history-based reasoning about collection

Bian, J.; Hiep, H.A.; Boer, F.S. de; Gouw, C.P.T. de

Formal Methods in System Design (2022) 61:63–89
https://doi.org/10.1007/s10703-023-00426-x

ORIGINAL ARTICLE

Integrating ADTs in KeY and their application to history-based reasoning about collection

Jinting Bian[1,3] · Hans-Dieter A. Hiep[1,3] · Frank S. de Boer[1,3] · Stijn de Gouw[1,2]

Received: 14 April 2022 / Accepted: 16 April 2023 / Published online: 9 May 2023

Abstract
We discuss integrating abstract data types (ADTs) 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 this new 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. *Open Science.* Includes video material (Bian and Hiep in FigShare, 2021. https://doi.org/10.6084/m9.figshare.c.5413263) and a source code artifact (Bian et al. in Zenodo, 2022. https://doi.org/10.5281/zenodo.7079126).

Keywords Formal verification · Abstract data types · Program correctness · Java collection framework · KeY · JML · Isabelle/HOL

1 Introduction

The overall aim of this paper is to put formal methods to work by the verification of software libraries which are the building blocks of millions of programs, and which run on the devices of billions of users every day. Our research agenda is to verify heavily used software libraries, such as the Java collection framework, since the verification effort weighs up against the potential impact of errors. In [15] the use of formal methods led to the discovery of a major

✉ Hans-Dieter A. Hiep
hdh@cwi.nl

Jinting Bian
j.bian@cwi.nl

Frank S. de Boer
frb@cwi.nl

Stijn de Gouw
sdg@ou.nl

1 Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

2 Open University, Heerlen, The Netherlands

3 Leiden Institute of Advanced Computer Science (LIACS), Leiden, The Netherlands

flaw in the design of TimSort, the default sorting method in many widely used programming languages, such as Java and Python, and platforms, such as Android. An improved version of TimSort was proven correct with the state-of-the-art domain-specific theorem prover KeY [1]. The correctness proof of [15] convincingly illustrates the importance and potential of formal methods as a means of rigorously validating widely used software and improving it. In [21] this line of research has been further successfully extended by the verification of the basic methods of (a corrected version of) the `LinkedList` implementation of the Java collection framework, laying bare an integer overflow bug, using again the KeY theorem prover.

KeY is tailored to the verification of Java programs. In a proof system based on sequent calculus, KeY symbolically executes fragments of the loaded program which are represented by modal operators of the underlying dynamic logic. KeY uses the Java Modeling Language [10], JML for short, for the specification of class invariants, method contracts, and loop invariants: these are annotations added to original Java programs. This specification language is intrinsically *state-based* and as such is not directly suitable for the specification of state-hiding *interfaces*. As such, our work described in [21] excludes the specification and verification of the `Collection#addAll(Collection)` method implemented by `LinkedList`, which adds all the elements of the collection that is passed as parameter. Here, the difficulty lies in giving a specification of the interface, that works for all possible implementations (including `LinkedList` itself).

In recent previous work [20] the concept of a *history* as a sequence of method calls and returns has been introduced as a general methodology for specifying interfaces and verifying clients and implementations of interfaces. As a proof-of-concept, using the KeY theorem prover, this methodology has been applied to the core methods of Java's `Collection` interface and uses an encoding of histories as Java objects on the heap. That encoding, however, 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 (with KeY). Furthermore, for recursively defined pure methods we also need to verify their *termination* and *determinism* [30]. 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 the 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*, that 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.

This paper avoids this complexity by instead modeling histories as *Abstract Data Types*, ADTs for short. Elements of abstract data types are not present on the heap, avoiding the need to use dependency contracts for proving that heap modifications affect their properties. Since KeY has limited support for user-defined abstract data types, we introduce a general *workflow* which integrates the domain-specific theorem prover KeY and the general-purpose theorem prover Isabelle/HOL [33] 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 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 an axiomatic semantics to programs and specifications. A general-purpose theorem prover, in contrast, is oblivious to the intricate details of programs and its specifications in question: e.g. it is not needed to formalize the semantics of Java nor of JML in our general-purpose theorem prover Isabelle/HOL. Our set-up thus differs from other approaches, such as in the Bali [32, 37] and LOOP [22, 25] projects, that *embed* the semantics of the programming language and specification language within the general-purpose theorem prover.

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 [20]. Although the previous approach works *in principle*, with our new approach we can *practically* give a specification of the `addAll` method and verify 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.

This paper is an extended version of the conference paper [5]: in this extended paper, we improved the accompanying artifact [6], we give more details on how we designed our specification of the `Collection` interface, and describe in more detail the steps needed to verify a number of complex example clients. In this paper we shall introduce and informally explain concepts as they appear on-the-fly, but for reproducing the proofs underlying the results it is helpful if the reader is familiar with KeY, JML, and Isabelle/HOL. Still, also to the non-specialist, this paper may be interesting as it shows what features of a specification and verification system we need in order to reason about real-world programs.

Related work The Java collection framework is among the most heavily-used software libraries in practice [11], and various case studies have focused on verifying parts of that framework [3, 23, 24]. Knüppel et al. [27] specified and verified several classes of the Java collection framework with standard JML state-based annotations, and found that specification was one of the main bottlenecks. One source of the complexity concerns framing: specifying and reasoning about properties and locations that do *not* change.

In state-based approaches, including the work by Knüppel et al. [27], (dynamic) frames [38] 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 [34] approaches for Java [16]. 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 [2]. 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 (by means of an interface), which raises the question: how to verify clients that make use of interfaces?

The history-based approach in this paper (contrary to our previous work on histories [20]) 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. Additionally, since we model such histories as elements of an ADT *separate from the sorts used by Java* in this paper, 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 arises in the approach of [20], which used an encoding of histories as ordinary Java objects living on the heap. We refer to the previous approach [20] as the *executable* history-based (EHB) approach, and the new approach presented in this paper as the *logical* history-based (LHB) approach.

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 [29] and Why3 [17], 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 to verify 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.

2 Integrating abstract data types in KeY

Abstract data types were introduced in 1974 by Barbara Liskov and Stephen Zilles [31] 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 lead 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 the textbook [35].

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" [28]. There are extensions of JML to support user-defined types on the specification level, e.g. model classes [12], but KeY does not implement them.

However, KeY does extend JML in an important way: a number of built-in abstract data types at the specification level are provided [1, 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 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 [7] that provides a mechanism for defining so-called *algebraic data types*,

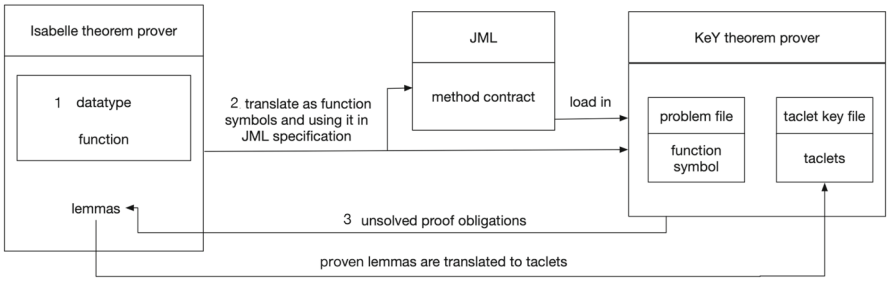

Fig. 1 The workflow of integrating ADTs in KeY

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 [36], 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 Fig. 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 correctness of Java 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. These definitions can not refer to Java types directly, but instead are defined using polymorphic type parameters, thereby abstracting away from Java types.
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):

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

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 Sect. 3.

Step 2. Using ADTs in JML specifications. The dynamic logic underlying KeY is a multi-sorted logic. To declare new data types and functions, we may introduce sorts and function symbols. The behavior of these function symbols are encoded as proof rules, which we formulate using an extensible formalism called *taclets* [18, 19]. 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 by 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 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 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 following from the underlying definition in Isabelle/HOL.

There are two ways in "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 $Some(o) = None$ 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 $Some(o) \neq None$. 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

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

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

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 2.

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

Listing 2 Adding a taclet to KeY that expresses the distinctness of constructors.

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 13 later on).

Another example shown below expresses 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 which 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 scope of this article, we instead refer to the KeY book [1].

3 History-based specification of collection

As a particular case study of working with abstract data types in KeY, we will employ ADTs to support history-based reasoning [20]. In this section we will motivate our approach, and give specifications of the `Collection` interface in terms of histories. In Sect. 4, we will illustrate the use of these specification in the verification of the correctness of clients of the `Collection` interface.

Listing 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 [8], by means of pre- 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 [23, 27]. 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. [38]) 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 [20, 26]: by modeling the interface using its history, no longer do we need ad hoc abstractions at the level of the interface. Further, since histories are modeled using ADTs of which its 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]. [8]

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 behavioural subtype hierarchy [?]. 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 3 The `Collection` interface.

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

Listing 4 The `Iterator` interface.

In order 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:

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

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 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 [8]:

> `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.) 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 which 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:

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

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.

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 of 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 are the same.

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.

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

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.

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

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 returns 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:

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

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 leaves 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:

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

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 is 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 [6]):

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

Intuitively, the clauses of the *isValid* predicate captures 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 *isIteratorValid*(h), i.e. that the iterator must be valid before the method `Iterator#next()` is called.
- *IteratorRemove*: Similar to above.

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 its 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 5. The model method has as return type the sort corresponding to the histories we defined earlier in Isabelle/HOL. We also specify the owner of an iterator by the means of a model method,

see Listing 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 5 The `history()` model method in JML.

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

Listing 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 [20].

As a guiding principle, we refrain from refering to constructors of the history in the pre- and postcondition specifications of interface methods. Thus, 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 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 call the `add` method of the same implementation.

3.2.1 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;
        \dl_isIteratorValid(\old(history()), it) ==>
        \dl_isIteratorValid(history(), it));
     @ ensures (\forall Iterator it;
      \old(it.owner()) == this; it.owner() == this);
     @*/
boolean add(Object o);
```

Listing 7 The specification of the `add()` method.

In Listing 7, lines 1–5 show the informal Javadoc of the `add` method [8]. 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 make 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.

3.2.2 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 8 shows the interface specification of the `addAll` method. Lines 1–7 show the informal Javadoc of the `addAll` method.

```
/** 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. **/
  * @ ensures (\exists elemlist el;
      (\forall Object o;
        \dl_occurs(el,o) == \dl_multiset(c.history(),o) &&
        \dl_multiset(history(),o) ==
        \dl_multiset(\old(history()),o) + \dl_multisetEl(el,o)));
    @ ensures (\forall Object o;
      \dl_multiset(c.history(),o) == \dl_multiset(\old(c.history()),o));
    @ ensures (\forall Object o;
     \dl_multiset(c.history(),o) > 0 ==>\dl_multiset(history(),o) > 0);
    @ ensures \result == false ==>
     (\forall Iterator it; it.owner() == this;
     \dl_isIteratorValid(\old(history()), it) ==>
     \dl_isIteratorValid(history(), it));
    @ ensures (\forall Iterator it;
     \old(it.owner()) == this; it.owner() == this);
    @*/
boolean addAll(Collection c);
```

Listing 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.

3.2.3 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, a `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 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 invalidating all other iterators.

The specification of this method is illustrated in Listing 9.

```
 /** Removes from the underlying collection the last element returned by
   * this iterator (optional operation).
   * This method can be called only once per call to next().
   * The behavior of an iterator is unspecified if the underlying
   * collection is modified while the iteration is in progress in any way
   * other than by calling this method. **/
  /* @ ...
     @ requires \dl_last(owner().history(),this) != \dl_None;
     @ ensures (\exists Object o;
          \dl_last(\old(owner().history()),this) == \dl_Some(o);
          \dl_multiset(owner().history(),o) ==
          \dl_multiset(\old(owner().history()),o) - 1);
     @ ensures (\exists Object o;
         \dl_last(\old(owner().history()),this) == \dl_Some(o);
         (\forall Object o1; o1 != o;
             \dl_multiset(owner().history(),o1) ==
             \dl_multiset(\old(owner().history()),o1)));
    @*/
void remove();
```

Listing 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 [6]. 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, that we used to close the proof obligations generated by KeY.

4 History-based client-side verification of collection

In this section, we will describe several case studies which we perform to show the feasibility and usability of our history-based reasoning approach supported by ADTs. Section 4.1 provides an example that we have verified with both the EHB approach [20] and the LHB approach described in this paper. This case supports our claim that the LHB approach yields a significant improvement of the total proof effort when compared to the EHB approach. As such, we are now able to verify more complex examples: the examples in Sect. 4.2 demonstrate reasoning about iterators, and, advancing further, we will verify binary methods in Sect. 4.3. Finally, proof statistics for all case studies are in Sect. 4.4.

We focus in this paper on the verification of client-side programs. Clients of an interface are, in principle, oblivious of 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.

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 [20] 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 10 Adding an object and if successful removing it again, leaves the contents of a `Collection` the same.

The client code and its contract is given in Listing 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 remains 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 $multiset(h', y) = multiset(h, y) - 1$ if the return value was **true**, and $multiset(h', y) = multiset(h, y)$ otherwise. Further, it ensures

the return value is **true** if $multiset(h, y) > 0$. Also, $multiset(h', x) = multiset(h, x)$ 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 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 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 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 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 [14] 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 does 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 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 auto-

[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.

Table 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

Name	Nodes	Branches	I.step	Q.inst	Contract	Dep	Loop inv	Time
Multiset	54,857	1053	52	476	39	0	0	72 min

matically 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 symbol *multiset* can be resolved using the contracts of the methods `add(Object)` and `remove(Object)` only, since these contract 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.

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 discuss 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 [20], we did verify a client (shown in Listing 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, 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 12 Iterating over the collection. Why does it terminate?

The main term needed to show 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 [1]. Following our workflow in Sect. 2, we are stuck in 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:

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

According to the definition of *iteratorSize*, it only adds 1 when executing the `next()` method, but the definition of *isIteratorValid* in Sect. 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 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 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 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 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 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 15 Taclet for showing the equality between *size* and *iteratorSize*.

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 [9] 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 properties of such an object have changed.

In the example given in Listing 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 not 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, that 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 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 mutation of two collections. Two collections are considered equivalent whenever they have the same multiplicities for all elements. The example shown in Listing 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 [13], 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 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 18. These ensures clauses need to be additionally mentioned in the collection's method specifications.[2]

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

Listing 18 Additional specification clauses needed to prevent potential aliasing.

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 iff 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.

4.4 Proof statistics

The proof statistics of all the use cases discussed in this article are given in Table 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 2 h. 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.

The rows marked † come from the EHB approach (encoding histories as Java objects [20]). The non-marked rows, i.e. the LHB approach, are part of the accompanying artifact [6]. Compared with the artifact [6] for our conference paper [5], 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 [4].

5 Conclusion

We showed how ADTs externally defined in Isabelle/HOL can be used in JML specifications and KeY proofs, and we applied this technique for specifying and verifying an important

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

Table 2 Summary of proof statistics

Name	Nodes	Branches	I.step	Q.inst	Contract	Dep	Loop inv	Time (min)
add_remove†	3936	79	44	5	2	23	0	11
add_remove	1514	15	12	7	2	0	0	1
iter_only†	8549	58	53	0	4	12	1	15
iter_only	6549	18	0	9	3	0	1	2
iter_remove	10,353	24	20	0	4	0	1	4
all_contains	23,900	94	187	40	5	0	2	40
compare_two	44,481	199	544	93	8	0	1	100

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

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 powerful enough to deal with binary methods and sub-objects such as iterators. Sub-objects require a notion of ownership as their behavior depend 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 paper [21].

Compared to modeling the history as an ordinary Java class (the so-called EHB approach) [20], the modeling of histories in this paper as an external ADT with functions (the so-called LHB approach) offers numerous benefits. In the latter we avoid pure methods that rely on the heap, which give rise to additional proof obligations every time these pure methods are used in JML specifications. Also we significantly simplified reasoning about properties of user-defined functions themselves. For example, in our case study, we reduced proofs from the previous paper (in KeY) about *multiset* modeled as a pure method from 72 min of work, to a fully automated verification in Isabelle/HOL with *multiset* modeled as a function.

This work has opened 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 use collections. Our most complex example, a binary method, takes more than 100 min to verify: it is hard to imagine that it can be done with the EHB approach. Further, while KeY is tailored for proving properties of concrete Java programs, Isabelle/HOL has more powerful facilities for general theorem proving. Our approach allows leveraging Isabelle/HOL to guarantee, for example, meta-properties such as the consistency of axioms about user-defined ADT functions. Using KeY alone, this was problematic or even impossible.

The main contribution of this paper is to provide a technique for integrating ADTs, defined in the general-purpose theorem prover Isabelle/HOL, in the domain-specific theorem prover KeY. We describe how data types, functions and lemmas can be imported into KeY from Isabelle/HOL. From the practical perspective, an automatic tool that imports Isabelle/HOL theories into KeY based on our work could be implemented, but such work is beyond the scope of this paper and we leave it as future work.

In this paper we have seen an application of our technique to the case of history-based reasoning. Our LHB approach is not only useful for reasoning about the Java collection framework, it is a general method that can also be applied to other libraries and their interfaces. We foresee that our technique can be extended to other common data types, such as trees and graphs, which provides a fruitful direction for future work.

A further next step is to continue work in the history-based specification of interfaces and its application to the Java Collection Framework. We can develop more advanced client-side use cases involving `addAll` but also the methods `removeAll` and `containAll` using our method. Moreover, we want to work on a general history-based refinement theory which allows to formally verify that a class *implements* a given interface, and, more specifically, that inherited methods are correct with respect to refinements of overridden methods.

References

1. Ahrendt W, Beckert B, Bubel R, Hähnle R, Schmitt PH, Ulbrich M (eds) (2016) vol 10001. LNCS. Springer, Berlin
2. Banerjee A, Naumann DA, Nikouei M (2018) A logical analysis of framing for specifications with pure method calls. ACM Trans Program Lang Syst 40(2):1–90
3. Beckert B, Schiffl J, Schmitt PH, Ulbrich M (2017) Proving JDK's dual pivot quicksort correct. In: 9th Conference on verified software, theories, tools, and experiments (VSTTE), volume 10712 of LNCS. Springer, pp 35–48
4. Bian J, Hiep HA (2021) Integrating ADTs in KeY and their application to history-based reasoning: video material. FigShare. https://doi.org/10.6084/m9.figshare.c.5413263
5. Bian J, Hiep HA, de Boer FS, de Gouw S (2021) Integrating ADTs in KeY and their application to history-based reasoning. In: Huisman M, Păsăreanu C, Zhan N (eds) Formal methods. Springer, Cham, pp 255–272
6. Bian J, Hiep HA, de Boer FS, de Gouw S (2022) Integrating ADTs in KeY and their application to history-based reasoning about collection: proof files. Zenodo. https://doi.org/10.5281/zenodo.7079126
7. Biendarra J, Blanchette JC, Desharnais M, Panny L, Popescu A, Traytel D (2016) Defining (co)datatypes and primitively (co)recursive functions in Isabelle/HOL. Available at: https://isabelle.in.tum.de/doc/datatypes.pdf
8. Bloch J, Gafter N (2010) Collection (Java Platform SE 7). Available at: https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
9. Bruce KB, Cardelli L, Castagna G, Eifrig J, Smith SF, Trifonov V, Leavens GT, Pierce BC (1995) On binary methods. Theory Pract Object Syst 1(3):221–242
10. Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Leino KRM, Poll E (2005) An overview of JML tools and applications. Int J Softw Tools Technol Transf 7(3):212–232
11. Costa D, Andrzejak A, Seboek J, Lo D (2017) Empirical study of usage and performance of Java collections. In: 8th Conference on performance engineering. ACM, pp 389–400
12. Darvas A, Müller P (2007) Faithful mapping of model classes to mathematical structures. In: 2007 Conference on specification and verification of component-based systems (SAVCBS). ACM, pp 31–38
13. de Boer M, de Gouw S, Klamroth J, Jung C, Ulbrich M, Weigl A (2022) Formal specification and verification of JDK's identity hash map implementation. In: ter Beek MH, Monahan R (eds) Integrated formal methods—17th international conference, IFM 2022, Lugano, Switzerland, June 7–10, 2022, proceedings, volume 13274 of lecture notes in computer science. Springer, pp 45–62

14. de Gouw S, de Boer FS, Rot J (2014) Proof pearl: the key to correct and stable sorting. J Autom Reason 53(2):129–139
15. de Gouw S, Rot J, de Boer FS, Bubel R, Hähnle R (2015) OpenJDK's Java.utils.collection.sort() is broken: the good, the bad and the worst case. In: 27th Conference on computer aided verification (CAV), volume 9206 of LNCS. Springer, pp 273–289
16. Distefano D, Parkinson MJ (2008) jStar: towards practical verification for Java. In: 23rd Conference on object-oriented programming, systems, languages, and applications (OOPSLA). ACM, pp 213–226
17. Filliâtre J-C, Paskevich A (2013) Why3: where programs meet provers. In: 22nd European symposium on programming, volume 7792 of LNCS. Springer, pp 125–128
18. Giese M (2004) Taclets and the KeY prover. Electron Notes Theor Comput Sci 103:67–79
19. Habermalz E (2000) Ein dynamisches automatisierbares interaktives Kalkül für schematische theorie spezifische Regeln. PhD thesis, University of Karlsruhe
20. Hiep HA, Bian J, de Boer FS, de Gouw S (2020) History-based specification and verification of Java collections in KeY. In: 16th International conference on integrated formal methods. Springer, pp 199–217
21. Hiep HA, Maathuis O, Bian J, de Boer FS, van Eekelen MCJD, de Gouw S (2020) Verifying OpenJDK's LinkedList using KeY. In: 26th Conference on tools and algorithms for the construction and analysis of systems (TACAS), volume 12079 of LNCS. Springer, pp 217–234
22. Huisman M (2001) Reasoning about Java programs in higher order logic using PVS and Isabelle. PhD thesis, University of Nijmegen
23. Huisman M (2002) Verification of Java's AbstractCollection class: a case study. In: 6th Conference on mathematics of program construction, volume 2386 of LNCS. Springer, pp 175–194
24. Huisman M, Jacobs B, van den Berg J (2001) A case study in class library verification: Java's Vector class. Int J Softw Tools Technol Transf 3(3):332–352
25. Jacobs B, Van Den Berg J, Huisman M, van Berkum M, Hensel U, Tews H (1998) Reasoning about Java classes: preliminary report. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp 329–340
26. Jeffrey A, Rathke J (2005) Java Jr: fully abstract trace semantics for a core Java language. In: Programming languages and systems (PLS), volume 3444 of LNCS. Springer, pp 423–438
27. Knüppel A, Thüm T, Pardylla C, Schaefer I (2018) Experience report on formally verifying parts of OpenJDK's API with KeY. In: F-IDE 2018: formal integrated development environment, volume 284 of EPTCS. OPA, pp 53–70
28. Leavens GT, Cheon Y (2006) Design by contract with JML. Available at: http://www.cs.utep.edu/cheon/cs3331/data/jmldbc.pdf
29. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Clarke EM, Voronkov A (eds) Logic for programming, artificial intelligence, and reasoning. Springer, Berlin Heidelberg, pp 348–370
30. Leino KRM, Müller P (2008) Verification of equivalent-results methods. In: 17th European symposium on programming, volume 4960 of LNCS. Springer, pp 307–321
31. Liskov B, Zilles S (1974) Programming with abstract data types. ACM SIGPLAN Not 9(4):50–59
32. Nipkow T (1999) Embedding programming languages in theorem provers. In: International conference on automated deduction. Springer, pp 398–398
33. Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic, volume 2283 of LNCS. Springer
34. Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: 17th Symposium on logic in computer science (LICS). IEEE, pp 55–74
35. Sannella D, Tarlecki A (2012) Foundations of algebraic specification and formal software development. Monographs in theoretical computer science. Springer, Berlin
36. Traytel D, Popescu A, Blanchette JC (2012) Foundational, compositional (co)datatypes for higher-order logic: category theory applied to theorem proving. In: 27th Symposium on logic in computer science (LICS). IEEE, pp 596–605
37. von Oheimb D (2001) Hoare logic for Java in Isabelle/HOL. Concurr Comput Pract Exp 13(13):1173–1214
38. Weiß B (2011) Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. PhD thesis, Karlsruhe Institute of Technology