Universiteit Leiden
The Netherlands

Reasoning about object-oriented programs: from classes to interfaces
Bian, J.

Reasoning about object-oriented programs: from classes to interfaces

Jinting Bian

Reasoning about object-oriented programs: from classes to interfaces

Jinting Bian

Reasoning about object-oriented programs: from classes to interfaces

Jinting Bian

ISBN 978-94-6496-122-5

Cover design: Zhiyu Gu

Reasoning about object-oriented programs: from classes to interfaces

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden,

op gezag van rector magnificus prof.dr.ir. H. Bijl,

volgens besluit van het college voor promoties

te verdedigen op dinsdag 21 mei 2024

klokke 11:15 uur

door Jinting Bian

geboren te Taiyuan,China

in 1994

Promotores:	Prof. dr. F.S. de Boer
	Prof. dr. M.M. Bonsangue
Promotiecommissie:	Prof. dr. R.V. van Nieuwpoort
	Prof. dr. H.C.M. Kleijn
	Prof. dr. M.-C. Jakobs (Ludwig-Maximilians Universität)
	Prof. dr. M. Sirjani (Malardalen University)
	Dr. E. Poll (Radboud Universiteit)
	Dr. A.W. Laarman

Jinting Bian was financially supported through the China Scholarship Council (CSC) to participate in the PhD programme of Leiden University. Grant number 202007720094.

The research in this thesis was performed at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science at Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

Chapter 1

Introduction

1.1 Background

Formal verification provides mathematically precise proof of the correctness of software with respect to formal logic-based specifications of the intended behavior. Formal verification can fully guarantee the correctness of software (as opposed, for instance, to testing) but can be challenging in practice, as it typically requires significant effort. However, without a formal analysis supported by verification tools, software that is used by billions of users every single day may contain bugs, that is, programming errors caused by humans. Indeed, in the TimSort case study [1], it is demonstrated that the formal specification and verification of programs can very well pay off: in an attempt to prove the correctness of the Timsort sorting algorithm [2], a flaw was found in its implementation which under certain circumstances could result in a crash. TimSort is the default sorting algorithm used e.g. in Java, Python, and Android, hence the number of affected devices is well into the billions. The correctness proof of [1] convincingly illustrates the importance and potential of formal methods as a means of rigorously verifying widely used libraries and improving them. For this result, the verification tool KeY, a semi-interactive theorem prover for Java programs, was used to mechanically verify the fixed version, which is therefore now guaranteed to be free of crashes and has been adopted in the most recent version of Python, Java, and Android. KeY is tailored to the verification of Java programs. It models the many programming features of the Java language present in real-world programs. KeY uses the Java Modeling Language [3], JML for short, for the specification of class invariants, method contracts, and loop invariants. JML supports the design by contract paradigm, it provides a way to write specifications that are easy to understand for Java programmers by embedding specifications as Java comments alongside the program.

The core work of this thesis is to extend the aforementioned successful applications of formal methods to a systematic verification of object-oriented programs. Our main focus is given to Java. Java is a widely used programming language that is designed with object-oriented programming principles at its core. As per the TIOBE

Programming Community Index,[1] Java consistently ranks among the top 5 most popular programming languages worldwide. One of the key reasons behind Java's enduring popularity is its extensive range of libraries, which can be employed to address virtually any type of programming challenge. The Java Collection Framework is among the most heavily used software libraries in practice [4]. It provides basic data structures and search and sorting algorithms (including Timsort) and is among the most widely used libraries. We direct specification and verification efforts towards the Java Collection Framework, and we also use it as a source of motivating examples throughout this thesis.

Next to the Timsort case, another issue related to the Java Collection Framework was revealed in [5]. `LinkedList` is the only `List` implementation in the Collection Framework that allows collections of unbounded size. It turns out that Java's linked list implementation does not correctly take the Java integer overflow semantics into account. It is exactly for large lists ($\geq 2^{31}$ items), that the implementation breaks in many interesting ways, and sometimes even in ways oblivious to the client. This basic observation gave rise to several test cases that show that Java's `LinkedList` class breaks the contract of 22 methods out of a total of 25 methods of the `List` interface! While the above illustrates the importance of formal methods as a means of putting widely used state-of-the-art software libraries to the test and improving them, it is clear that without a formal analysis supported by verification tools, libraries that are used by billions of users every single day are bound to contain bugs.

Our work offers several significant advantages in real-world program correctness and verification. Firstly, this thesis is based on extensive use of the KeY theorem prover, a tool specifically designed for asserting the correctness of Java programs, enhancing the accuracy and reliability of our verifications. Secondly, our work operates on a large scale with a focus on general-purpose libraries. Thirdly, we focus on directly verifying unaltered Java code from the standard library, ensuring that our methods are directly applicable to real-world programming scenarios. Lastly, our work encompasses both classes and interfaces within the standard library, thereby providing a thorough and robust examination of Java's foundational components. This thesis shows that formal verification of actual code can be done in practice.

Related work Here we provide a general related work of this thesis, more specific related work is described in each individual chapter.

Throughout the history of computer science, a major challenge has been how to assert that software is free of bugs and works as intended. Software bugs can lead to serious negative impacts on any software system. From a user's perspective, these bugs can interfere with program functionality and undermine the overall user experience. Additionally, they may create security vulnerabilities that can leave the system open to malicious activity. According to new research in 2022 commissioned by the Consortium for Information and Software Quality (CISQ), the economic burden of poor software quality in the United States has escalated to at least $2.41 trillion, marking an increase of almost 16% from the 2020s $2.08 trillion [6]. This

[1] https://www.tiobe.com/tiobe-index/

substantial amount does not even include an estimated $1.52 trillion in "technical debt" (TD) — accumulated software vulnerabilities in applications, networks, and systems that are ignored for now but will need to be paid eventually.

Using informal root cause analysis [7], one could find from a system failure its root causes, which may include bugs caused by programming errors. But, root cause analysis can only be applied *after* a failure has happened. To prevent certain failures from happening in the first place, program correctness is of the utmost importance. Although establishing program correctness can be an expensive activity [8], it is still worthwhile for critical software programs, such as the standard library that all programs rely on.

It can be empirically established that Java libraries, and Java's Collection Framework in particular, are heavily used and have many implementations [4]. Various case studies have focused on verifying parts of that framework [9, 10, 11]. Numerous approaches have been proposed to specify and check Java programs to ensure program reliability for run-time verification. Most of these approaches are based on communication histories (or traces), which are defined as finite sequences of messages. In particular, in [12], the tool environment prototyped for JML has pioneered runtime assertion checking by integrating communication histories, offering a nuanced understanding of method interactions and invocations. A specialized approach [13] extending this work for JML employs runtime assertion checking in multithreaded applications, using e-OpenJML as a tooling environment. Furthermore, the development of the MOP framework [14] showcases the feasibility of a generic and efficient approach to runtime verification, indicating its potential utility beyond just the Java Collection Framework. Explicit method call sequences in Java programs have also been studied in [15], ensuring that methods are invoked in the intended order and manner, strengthening the reliability of the overall system. LARVA [16], a tool also mainly developed for run-time verification, was extended in e.g. [17] to optimize away checks at run-time that can be established statically. But, there, static guarantees are limited by expressivity (no fully-fledged theorem prover is used) and interfaces are not handled by the static analysis. The work [18] by Welsch and Poetzsch-Heffter focuses on Java libraries. They reason about the backward compatibility of Java libraries in a formal manner using histories to capture and compare the externally observable behavior of two libraries. In [18], however, two programs are compared, and not a program against a formal specification. The Java PathFinder is an extensible software framework that performs model checking directly on Java bytecode, originally developed at NAS Ames Research Center. Works using Java PathFinder [19, 20] mostly focus on checking concurrency defects and unhandled exceptions in Java programs.

The programming errors discovered in the Timsort and `LinkedList` case studies are hard to discover at run-time due to their heap size requirements, necessitating a static approach to analysis. Static verification of the Collection Framework was already initiated more than two decades ago, see e.g. the work by Huisman *et al.* [10, 21]. A significant complication in static verification is that it requires formal specifications. Two well-known approaches in this domain come from Huisman [10] and Knüppel *et al.* [11]. Huisman [10] presented the specification and

modular verification of Java's `AbstractCollection` class identifying a problem not documented in the informal documentation: unexpected behavior occurs when a collection contains a reference to itself. Knüppel et al. [11] specified and verified several classes of the Java Collection Framework with standard JML state-based annotations; they found that specification was one of the main bottlenecks. However, these approaches are not complete nor demonstrate the verification of various clients and implementations. Generally speaking, while existing research mainly focuses on class specification and verification, there seems to be no obvious strategy in specifying Java interfaces so that its clients and its implementations can be verified statically using a theorem prover.

1.2 Thesis outline

The thesis is organized into eight chapters. Figure 1.1 provides an overview of the thesis structure.

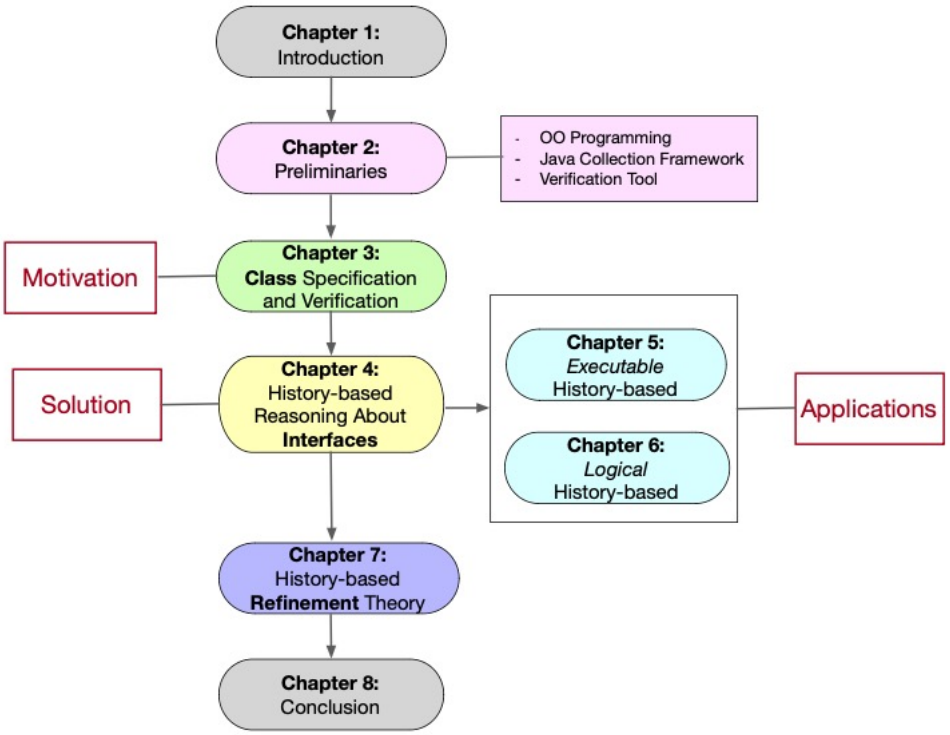

Figure 1.1: Thesis structure.

In this introductory chapter, we provide a brief overview of the background and related work, outline the main challenges, and highlight the key contributions of this thesis. Chapter 2 presents the background knowledge, offering an initial survey of several key concepts and introduces the theorem provers used for verification.

In Chapter 3, we focus on class specification and verification. We present the methodology for analyzing the `java.util.LinkedList` class to gain a deeper understanding of its behavior. We emphasize the crucial role of precise specifications using JML. To validate the program's behavior against these specifications, we take a formal approach supported by the KeY tool, which uniquely allows for comprehensive reasoning about Java programs.

However, we leave out some methods that take an interface type as a parameter, as these present challenges we can not solve yet. Motivated by this challenge, we focus on investigating a history-based approach to writing interface specifications in the state-based specification language JML by referring to histories and their attributes to describe the intended behavior of implementations The content related to interface specification and verification is present in Chapters 4, 5, and 6.

Specifically, in Chapter 4, we outline the challenges of proving client code correct with respect to arbitrary implementations and describe a practical specification and verification effort of part of the `Collection` interface using KeY. We explore two history-based approaches: the *executable* history-based approach (the EHB approach) and the *logical* history-based approach (the LHB approach).

In Chapter 5 and Chapter 6, we apply these history-based reasoning approaches to Java's `Collection` interface and use histories to prove the correctness of several clients. As a more advanced case study, we have also verified clients using the LHB approach, which operates on multiple objects. This significantly improves the state-of-the-art of history-based reasoning.

In Chapter 7, we focus on hierarchical structures in object-oriented programs. We investigate reasoning about behavioral subtyping based on histories, which enables us to formally verify the class and interface hierarchy.

At last, Chapter 8 summarizes the contributions of the thesis and points out directions for future research.

1.3 Main challenges and contributions

This thesis demonstrates formal methods for systematically verifying real object-oriented programs. The goal of this thesis is to develop techniques that are capable of specifying and verifying interface/class hierarchies, such as those present in Java's collection framework, and to convincingly argue about the usefulness of these techniques by means of case studies. To achieve this aim, in this thesis, we tackle several challenging problems that hamper the verification of object-oriented programs.

When reasoning about object-oriented programs, the ultimate goal is to ensure that program execution does not result in unexpected behavior. This requires a precise specification of what behavior one expects and further requires a convincing argument that all possible executions of the program exhibit that behavior. Focusing on this challenge, in Chapter 3, we give a tutorial on using the KeY theorem prover to demonstrate formal verification of state-of-the-art, real software. In sufficient detail for a beginning user of JML and KeY, the specification and verification of part of a corrected version of the `java.util.LinkedList` class of the Java Collection framework is explained. We provide technical details on how we use the KeY theorem prover, and we give more detail concerning the production of the proofs. This tutorial consists of artifacts and video materials. The collection of video material consists of screen recordings of interactive proof sessions with the KeY theorem prover that shows recordings of interactive sessions. Each video displays how to create a proof for part of one method contract, or proofs of several method contracts.

The material in Chapter 3 is based on a conference paper, an artifact, and a collection of video materials:

- Hiep, H.A., **Bian, J.**, de Boer, F.S., de Gouw, S. (2020). A Tutorial on Verifying LinkedList Using KeY. Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY, 221-245.
- **Bian, J.**, Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Video Material (2020).
  Available at: https://doi.org/10.6084/m9.figshare.c.4826589.v2
- Hiep, H.A., **Bian, J.**, de Boer, F.S., de Gouw, S.: A Tutorial on Verifying LinkedList using KeY: Proof Files (2020).
  Available at: https://doi.org/10.5281/zenodo.3613711

In addition to the verification of class types, some of the methods in libraries contain an interface type as a parameter, which generates the next challenge. For example, as shown in Listing 1.1, the `addAll` method in `java.util.LinkedList` class of the Java Collection framework takes two arguments, the second one is of the `Collection` type:

```
public boolean addAll(int index, Collection c){
 ...
 Object[] a = c.toArray();
 ...
}
```

Listing 1.1: The `addAll` method in `LinkedList`.

Following the design by contract paradigm, verification of `LinkedList`'s `addAll` method requires a contract for each of the other methods called, including the `toArray` method in the `Collection` *interface*. How can we specify interface methods, such as `Collection.toArray`? More generally, libraries employ the technique of *programming to interfaces*, which is one of the most important principles in software engineering. Interfaces in Java abstract from the state and other internal implementation details. However, tool-supported programming logic and specification languages are predominantly state-based which as such cannot be used for Java interfaces. Furthermore, state-based specifications may expose the internal representation of a data structure. Different implementations of, for example, a stack would give rise to different specifications. Consequently, state-based specifications are not robust under changes to the underlying representation.

Addressing this challenge, in Chapter 4, we show the feasibility of an approach that overcomes this limitation, by integrating history-based reasoning with existing specification and verification methods (in the KeY theorem prover). The formal semantic justification of this approach is provided by the fully abstract semantics for Java introduced in [22] which characterizes exactly the minimal information about a method implementation in a class in a Java library that captures its external use. This minimal information consists of *histories* (also called *traces*) of method calls and returns, and provides a formal semantic justification of the basic observation that

such histories completely determine the concrete state of any implementation and thus can be viewed as constituting the generic abstract state space of an interface. This observation naturally leads to the development of a history-based specification language for interfaces.

The material of Chapter 4 is based on a journal paper and a conference paper:

- **Bian, J.**, Hiep, H.A., de Boer, F.S., de Gouw, S. (2023). Integrating ADTs in KeY and their application to history-based reasoning about collection. Formal Methods in System Design, 1-27.
- Hiep, H.A., **Bian, J.**, de Boer, F.S., de Gouw, S. (2020). History-based specification and verification of Java collections in KeY. In Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings 16 (pp. 199-217). Springer International Publishing.

To implement history-based reasoning, we first embed histories and attributes in the KeY theorem prover [23] by encoding them as Java objects, thereby avoiding the need to change the KeY system itself. Interface specifications can then be written in the state-based specification language JML [24] by referring to histories and their attributes to describe the intended behavior of implementations. We call this approach the *executable* history-based approach (the EHB approach). In Chapter 5, we use this approach to reason about Java's Collection interface in proving client code correct with respect to arbitrary implementations and describe a practical specification and verification effort of the part of the `Collection` interface using KeY. We provide source and video material of the verification effort to make the construction of the proofs fully reproducible.

The material of Chapter 5 is based on a conference paper, an artifact, and a collection of video materials:

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

However, the encoding of the EHB approach made use of pure methods in its specification and thus required extensive use of so-called *accessibility* clauses, which express the set of locations on the heap that a method may access during its execution. These accessibility clauses must be verified (with KeY). Furthermore, for recursively defined pure methods we also need to verify their *termination* and *deter-*

minism [25]. Essentially, the associated verification conditions boil down to verifying that the method under consideration computes the same value starting in two heaps that are different except for the locations stated in the accessibility clause. To that end, one has to symbolically execute the method more than once (in two different heaps) and relate the outcome of the method starting in different heaps to one another. After such proof effort, accessibility clauses of pure methods can be used in the application of *dependency contracts*, which are used to establish that the outcome of a pure method in two heaps is the same if one heap is obtained from the other by assignments outside of the declared accessible locations. The degree of automation in the proof search strategy with respect to pure methods, accessibility clauses, and dependency contracts turned out to be rather limited in KeY. So, while the methodology works in principle, in practice, for advanced use, the pure methods were a source of large overhead and complexity in the proof effort.

Facing this challenge, we introduce an innovative strategy: 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 to prove that heap modifications affect their properties. We term this novel approach the *logical* history-based (LHB) approach. In Chapter 6, we employ the LHB approach to verify several example client use cases of the interface and prove the correctness of a client that operates on multiple objects. This significantly improves the state-of-the-art of history-based reasoning. We have supplied video material along with a source code artifact for a more comprehensive understanding of our LHB approach.

The material in Chapter 6 is based on a conference paper, a journal paper, an artifact, and a collection of video materials:

- **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.
  Available at: 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.
  Available at: https://doi.org/10.5281/zenodo.7079126

The use of hierarchy in object-oriented design is foundational to organizing, structuring, and modeling real-world entities and their relationships. Type hierarchies support the *programming to interfaces principle* in object-oriented design by allowing the declaration of new subtypes that inherit properties and behaviors from their supertypes while also providing the flexibility to add or override specific features

as needed. In Chapter 7, we discuss the development of a general history-based refinement approach, which allows us to formally verify behavioral subtyping in class and interface hierarchies. Our primary focus is on the Java Collection Framework; however, we extend our discussion to demonstrate the practical utility of our history-based refinement theory through a banking example.

The material in Chapter 7 is based on a paper to be submitted and an artifact:

- **Bian, J.**, Hiep, H.A., de Boer, F.S. History-based Reasoning about Behavioral Subtyping. (To be submitted.)
- **Bian, J.**, Hiep, H. A., de Boer, F. S. (2024). History-Based Reasoning about Behavioral Subtyping: Proof files.
  Available at: https://doi.org/10.5281/zenodo.10998227

The theoretical part of this thesis was developed in close collaboration with my co-authors. The implementation, which includes using KeY for the formal verification of the Java Collection Framework and creating video material of the interactive sessions, was mainly done by myself. Actually, my whole thesis derived from the challenge of verifying real Java programs. The actual work did not always proceed in a top-down manner from theory to implementation; rather, problems encountered during implementation led to further theoretical developments. For example, with respect to verification of interfaces using the executable history-based approach, we found that this approach is rather limited in KeY. This led to the development of the logical history-based reasoning approach, which I then applied in developing the history-based refinement theory.

Chapter 2

Preliminaries

2.1 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that utilizes "objects" to structure applications and computer programs. In OOP, classes act as user-defined data types that serve as blueprints for creating unique instances called objects. These objects can represent real-world entities or abstract concepts. Each class defines both fields and methods, where fields represent the state of an object, and methods describe its behaviors.

Fields in a class are variables that store data related to instances of the class, and they define the state of an object. Class fields, on the other hand, belong to the class itself. Methods are functions encapsulated within the class, enabling code reusability and defining the behaviors associated with an object. Each method within a class begins with a reference to an instance object, making them instance methods.

A method signature consists of the method name coupled with its parameter list. An interface in OOP only contains method declarations, specifying what methods a class must implement but not how to implement them. This allows multiple classes to implement the same interface in different ways, giving programmers the flexibility to change implementations without affecting the code that relies on these interfaces.

The basic features of object-oriented programming include encapsulation, data abstraction, inheritance, and polymorphism, all of which are consistently adhered to by object-oriented programs. *Encapsulation* involves using classes to encapsulate both fields and methods, ensuring that the internal state of these objects is shielded from the outside world—a principle known as information hiding. Methods serve as the public interface through which interaction with object data is controlled, preserving data integrity and restricting unauthorized access. *Data abstraction* allows for the creation of abstract classes or interfaces that define the essential feature of a data type, without revealing the underlying implementation details. This enables users to interact with objects at a higher, more abstract level, which facilitates more comfortable and straightforward usage. *Inheritance* is widely used in object-oriented libraries to promote code reusability and to establish a relationship between super-

type and subtype. A subtype inherits fields (if both types are classes) and methods from a supertype, and it also has the ability to override or extend these inherited features. This makes it easy to add functionalities to existing classes or interfaces without modifying them. *Polymorphism* enables objects to be treated as instances of their supertype, leading to simpler code and fewer errors. In object-oriented programs, polymorphism is often implemented through method overriding in subtypes, allowing for different implementations under the same method name.

The *programming to interfaces* principle [26] is one of the most important principles in OOP. This principle is supported by the feature of encapsulation and data abstraction, allowing developers of client code to focus on essential functionalities rather than the need to consider irrelevant implementation details. By focusing on what an object should do rather than how it should do it, this principle enables a higher level of abstraction and decoupling in software systems. Furthermore, when combined with the *design by contract* [27] principle, programming to interface becomes a crucial strategy for managing the complexity of software today. This combination ensures a structured approach to software design, where the focus is on fulfilling clearly defined contracts, thereby enhancing the reliability and maintainability of software systems.

2.2 Java Collection Framework

The Java Collections Framework is a comprehensive set of classes and interfaces provided by Java for working with collections of objects. It provides basic data structures and is among the most widely used libraries. The Java Collection Framework is illustrated in Figure 2.1.

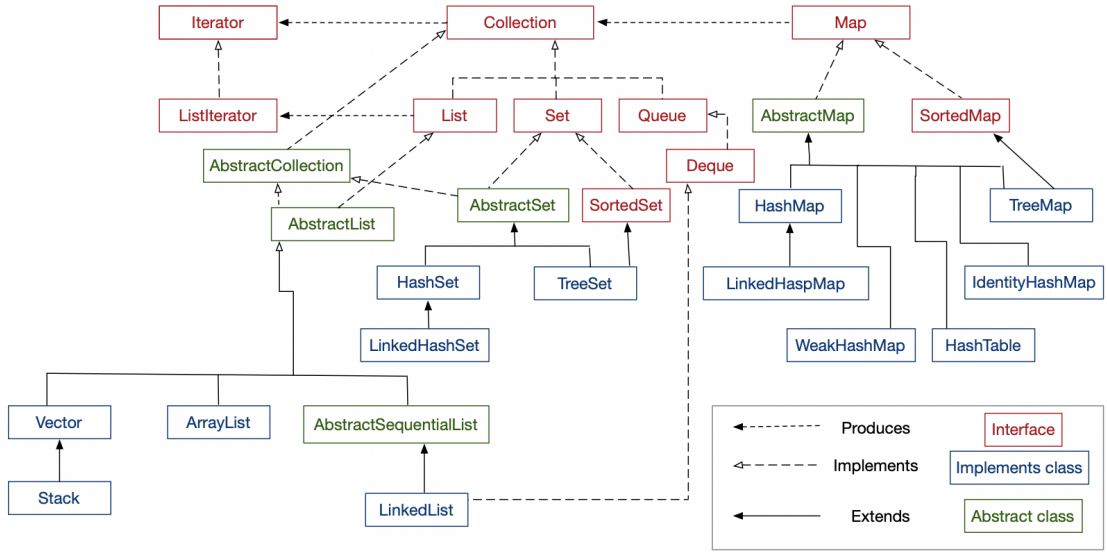

Figure 2.1: Java Collection framework.

The Java Collection Framework has a behavioral subtype hierarchy [28]. The `Collection` interface serves as the topmost type and includes three sub-interfaces: `List`, `Set`, and `Queue`. These sub-interfaces are implemented by some abstract

classes and finally lead down to the concrete implementation classes at the bottom of the hierarchy, such as `ArrayList`, `HashSet`, and `LinkedList`, etc.

The `Collection` interface outlines all the fundamental operations for collections, such as `add`, `remove`, and other methods for querying and manipulating a collection of objects.

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

Listing 2.1: The part of `Collection` interface.

The `iterator` method, declared within the `Collection` interface, returns an object that implements the `Iterator` interface. The `Iterator` interface is another key component of the Java Collection Framework, offering a way to enumerate all the elements in a collection.

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

Listing 2.2: The `Iterator` interface.

In addition to collections, the framework also includes a variety of `Map` interfaces and classes. These `Map` entities are used to store key/value pairs. Even though a `Map` is not considered as a collection, it is fully integrated into the Java Collections Framework.

The `LinkedList` class is one of the most important classes in the Java Collection Framework. This class serves as a particular case study for the formal verification of objection-oriented libraries, as discussed in Section 3. It was introduced in Java version 1.2 as part of the Java Collection Framework in 1998. Figure 2.1 shows how `LinkedList` fits in the type hierarchy of this framework: `LinkedList` implements the `List` interface, and also supports all general `Collection` methods as well as the methods from the `Queue` and `Deque` interfaces. The `List` interface provides positional access to the elements of the list, where each element is indexed by Java's primitive `int` type. Each element in a `LinkedList` is stored as a separate object referred to as a node, containing data and references to the previous and next elements in the list.

2.3 Theorem prover overview

In our setup, we distinguish domain-specific theorem provers from general-purpose theorem provers. 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. In our case study, we use the state-of-the-art KeY theorem prover, as KeY is tailored to the verification of Java programs. A general-purpose theorem prover, in contrast, is oblivious to the intricate details of programs and the specifications in question: e.g. it is not needed to formalize the semantics of Java nor JML in a general-purpose theorem prover. As a general-purpose theorem prover, we choose the Isabelle/HOL theorem prover. In this section, we will introduce both the KeY theorem prover and the Isabelle/HOL theorem prover, offering an overview of both tools.

2.3.1 KeY theorem prover

JML [24] is a specification language for Java that supports the design-by-contract paradigm. Specifications are embedded as Java comments alongside the program. A method precondition in JML is given by a `requires` clause and a postcondition is given by `ensures`. JML also supports class invariants. A class invariant is a property that all instances of a class should satisfy. In the design by contract setting, each method is proven in isolation (assuming the contracts of methods that it calls), and the class invariant can be assumed in the precondition and must be established in the postcondition, as well as at all call-sites to other methods. To avoid manually adding the class invariant at all these points, JML provides an `invariant` keyword which implicitly conjoins the class invariant to all pre- and postconditions. Method contracts may also contain an `assignable` clause stating the locations that may be changed by the method (if the precondition is satisfied), and an `accessible` clause that expresses the locations that may be read by the method (if the precondition is satisfied). Our approach uses all of the above concepts.

JML also allows annotations of ghost fields and model fields. Ghost fields are virtual fields that become part of the modelled state of an object on the heap, but are never present when actually executing a Java program. Like normal fields, the ghost fields are assigned a default value at object initialization and can be explicitly changed by JML `set` annotations. These annotations occur anywhere in method bodies where otherwise a normal statement can be expected. Model fields are introduced as function symbols, and several axioms are added that allow the definition of model fields to be substituted during proof. An example model field is a *class invariant*, which is implicitly assumed to hold the state of an object between method invocations.

KeY [23] is a semi-interactive theorem prover for Java programs (typically $> 95\%$ of the proof steps are automated). The input for KeY is a Java program together with a formal specification in a KeY-dialect of JML. The user proves the specifications method-by-method. KeY generates appropriate proof obligations and expresses them in a sequent calculus (see Figure 2.2), where the formulas inside the sequent are multi-modal dynamic logic formulas in which Java program fragments are used as the modalities. To reduce such dynamic logic formulas to first-order

formulas, KeY symbolically executes the Java program in the modality (it has rules for nearly all sequential Java constructs). Once the program is fully symbolically executed, only formulas without Java program fragments remain. The main reference work on the KeY system is the KeY book [29].

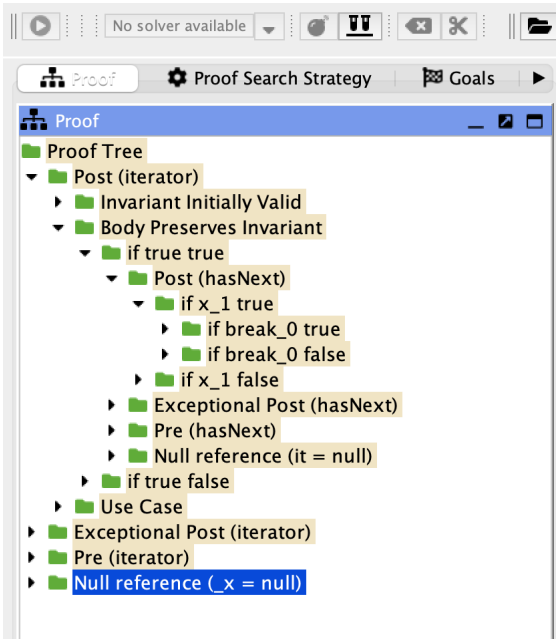

Figure 2.2: Proof tree in KeY(version 2.8.0).

The KeY system consists of three main components: a proof system, a translator of Java programs annotated with JML into proof obligations, and an interactive tool for constructing proofs.

The logic underlying KeY is *JavaDL*, a program logic that directly incorporates Java program fragments. The program logic is a multi-modal logic: $\langle P\rangle\varphi$ expresses that executing the program fragment P definitely terminates and φ holds in the final state; and $[P]\varphi$ expresses *if* the program fragment P terminates, *then* φ holds in the final state. The formula $\varphi \rightarrow \langle P\rangle\psi$ expresses that if φ holds in the initial state, then execution of P terminates in a state for which ψ holds. See [29].

The program logic distinguishes *program variables* from *logical variables*: the value of a program variable can be changed throughout executing a program, whereas a logical variable always has the same value. As logical variables can never be modified by a program fragment, they are used as so-called *freeze variables*.

The proof system that KeY uses to establish the validity of formulas is given as a sequent calculus. A sequent $\varphi_1, \ldots, \varphi_n \Rightarrow \psi_1, \ldots, \psi_m$ consists of n antecendents and m consequents, all formulas of JavaDL, with the usual interpretation: $\varphi_1, \ldots, \varphi_n \Rightarrow \psi_1, \ldots, \psi_m$ means that if all φ_i on the left are true, then at least one ψ_j on the right is true. Derivability of a sequent in the proof system is as usual by means of deduction rules, assembled into a proof tree. Next to the deduction rules for classical first-order logic, the proof system also consists of a large number of other rules.

Deduction rules are given by means of lightweight tactics (called taclets [30]) that perform modifications on the sequent one is proving: e.g. split branches in the proof tree, substitute variables, rewrite terms, or close a branch. There are approximately 1750 rules that implement symbolic execution for Java program fragments, and implement the theories of many sorts: integers, sequences, heaps, location sets, and others.

Of particular interest to us are rules concerning *updates* and *heaps*. Some rules transform modalities with program fragments into so-called update modalities. Updates always terminate and they assign JavaDL terms to program variables. As such, updates cannot assign program variables to side-effectful expressions. Given a formula φ, then $\{\mathtt{x}_1 \coloneqq t_1 || \ldots || \mathtt{x}_n \coloneqq t_n\}\varphi$ is a formula where in parallel $\mathtt{x}_i$ are updated to t_i. Updates are simplified by substitution.

There is a hidden and implicit program variable in JavaDL that is the `heap` of heap sort. The heap is used to model the storage of objects, that is, the value of fields associated with object references. From a practical perspective, updates of program variables other than the heap can be thought of as stack variables. Such program variables can *refer* to objects on the heap. Heap updates are tracked by updating the `heap` program variable. Heap updates and program variable updates easily form complex expressions, where both kinds of updates can be intertwined. See [29, Section 2.4.3] and [29, Section 6.4].

Next to the built-in sorts of integers, sequences, heaps, and location sets, are Java types. Each Java type has its own sort in JavaDL that models (infinitely many) references to objects, including the `null` reference. References can be explicitly coerced between sorts, to model subtypes: e.g. every reference can be coerced to a reference of sort `Object`. A heap assigns values to the fields of a non-null reference. Object references are global, but the creation status of each object is a special Boolean field `<created>` local to each heap. An object becomes created in some heap by taking a fresh reference, that has no value yet assigned to its creation field in that heap, and setting that field to `true`. Fields can only be assigned to non-null object references for which their created field is true. A heap is *well-formed* if a finite number of references have `<created>` set to true.

An important aspect of Java programs is method calls. There are three main issues: calling a method may introduce new program variables that shadow older program variables, calling a method may change the heap, and it is possible to call a method on an object for which its exact type is only known at run-time.

To solve the issue of overshadowing older program variables, KeY uses *method frames*; before a method frame is created, it is ensured that old program variables and new program variables do not collide by renaming the program variables of a method body. Since the `this` keyword cannot be renamed, the method frame provides a context in which program fragments evaluate `this` references; it also tracks where the method return value must be placed.

The implicit heap variable is also stored in the method frame, referred to as the before heap. Any heap update within the method is performed on a separate program variable. Statements following a method call are performed using the heap

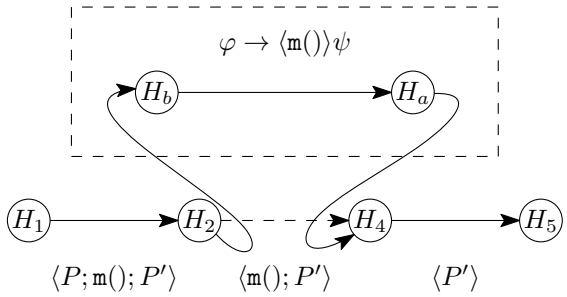

Figure 2.3: Heaps are 'threaded' through method calls. In heap H_1 the program fragment P is executed, resulting in heap H_2. Either a contract for method `m()` is employed, which relates the before heap H_b to some heap after the method call H_a; or, the method body is unfolded by wrapping it in a method frame that resolves overshadowing of program variables. In both cases, to call the method, heap H_b is equal to H_2 and heap H_a is equal to H_4. Then the program fragment P' that occurs after the method call is executed using the heap returned by the method.

as it is after the method call completes: there are rules for 'threading' the heap through a method call, see Figure 2.3. There are roughly two ways of treating method calls: unfolding their method body wrapped in a method frame, or using its method contract.

The issue of run-time types is solved using method contracts. Method contracts and other annotations of Java programs are specified in terms of the Java Modeling Language (see [31]) but with KeY-specific extensions (see [29, Chapter 7]). Each method is annotated by a contract that specifies pre- and post-conditions: all implementations of this method should adhere to the contract. The contract determines corresponding correctness formulas in JavaDL: if they are verified, then the corresponding method is correct with respect to its method contract. Consequently, method contracts can be reused in other proofs involving program fragments that call such a method.

Java programs annotated with JML are entered into the KeY system and the verification begins by generating one or more *proof obligations*. Using the interactive tool, a proof tree is constructed by applying rules until the branches of the tree are closed. Each rule application may result in zero or more branches: in the case of multiple branches, every branch needs to be closed. The tool also provides automation, that automatically applies proof rules using heuristics. This speeds up the verification effort considerably. Finally, after all proof obligations appear as the conclusion of a closed proof tree, the verification effort is done. The proof trees can be stored in a file and reloaded and inspected later.

2.3.2 Isabelle/HOL theorem prover

Isabelle/HOL (Isabelle instantiated with Church's type theory) [32] is an interactive theorem prover that supports the formalization and verification of mathematical theorems and software systems. The main reference for Isabelle/HoL is its reference manual [33, 34].

Isabelle/HOL employs higher-order logic (HOL) to allow for a broader class of mathematical statements to be represented and proven. The underlying logic is conducive to reasoning about functional programming constructs, data types, and even imperative programming paradigms. It comes equipped with a variety of decision procedures and tactical theorem-proving capabilities that automate or semi-automate the verification process [33].

Functional programs in Isabelle/HOL are outlined as shown in Figure 2.4. It defines both datatypes and functions within a theory. Isabelle comes with a large theory library of formally verified mathematics, including elementary number theory, analysis, algebra, and set theory [35].

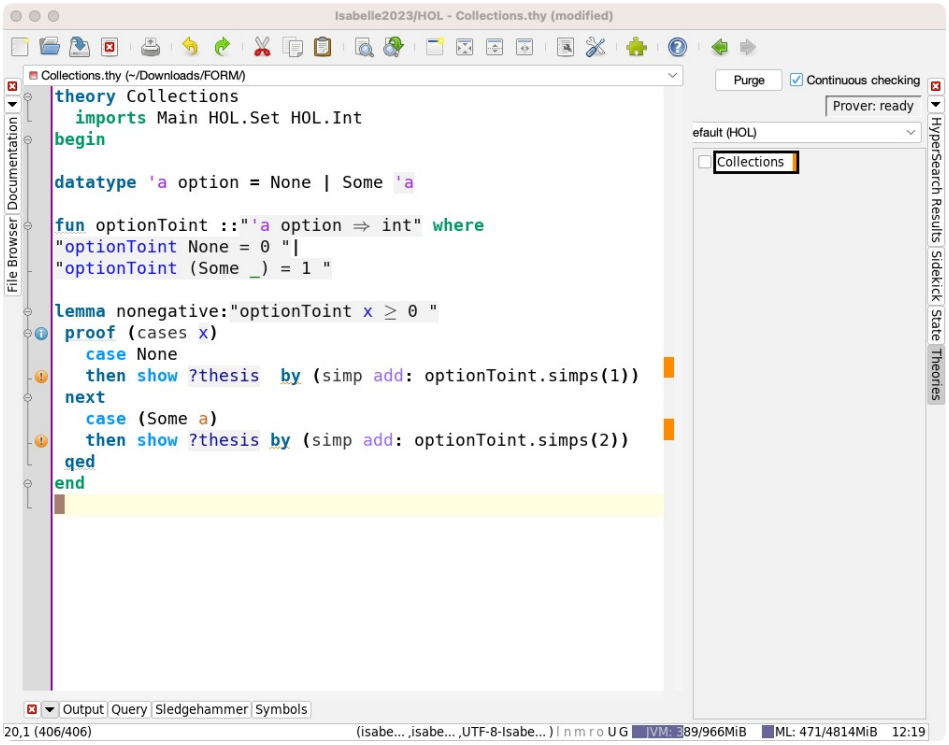

Figure 2.4: Isabelle/HoL proof system(version Isabelle2023/HOL).

Isabelle/HOL includes a definitional package for data types, as described in the work of Biendarra et al. [36]. The definition mechanism provides so-called *freely generated* data types: the user provides some signature consisting of constants and function symbols and their types, and the system automatically derives (rather than postulates) characteristic theorems. Under the hood, each data type definition is associated with a Bounded Natural Functor (BNF) that admits a non-trivial initial algebra [37]. User-defined datatypes are defined using the *datatype* keyword. The definition usually outlines the constructor functions that can be used to create instances of the datatype, along with the types of arguments these constructors can take.

Functions in Isabelle/HOL are declared using the *fun* keyword. These functions are characteristically pure, meaning they do not have side effects. Functions are usually defined by specifying their behavior over user-defined or built-in datatypes, commonly utilizing pattern matching to unambiguously identify the expected output

for specific inputs. Additionally, these functions can also be recursively defined; they may call themselves with altered arguments until a termination condition, often referred to as the base case, is met. This recursion is typically defined with rigorous formal properties to ensure termination and correctness, aligning with the overall goal of theorem proving and formal verification in Isabelle/HOL.

The keywords *theorem* and *lemma* in Isabelle play crucial roles in formalizing and verifying mathematical claims. While a lemma is generally a minor result used as a foundational stepping stone, a theorem is a more significant mathematical proposition that has been rigorously proven based on previously established lemmas, theorems, and axioms. These two keywords are interchangeable and merely indicate the level of importance we assign to a given proposition. Isabelle's rich theory library provides a wide array of pre-proven theorems and lemmas. These resources not only expedite the formal verification of new mathematical claims but also contribute to a robust foundational framework. This framework is invaluable for researchers across multiple disciplines, enabling them to build upon proven results and advance both theoretical and applied research with confidence.

Chapter 3

Class specification and verification: a step-by-step guide

This chapter is designed to serve as a comprehensive tutorial, illustrating the formal methods for specifying and reasoning about program executions in Java. As a specific case study for `java.util.LinkedList`, we provide the source code and video material to facilitate a deeper understanding of the entire proof process.

This chapter is based on the following publications and artifacts:

- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). A Tutorial on Verifying LinkedList Using KeY. Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY, 221-245.
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 1 (2020). https://doi.org/10.6084/m9.figshare.11662824
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 2 (2020). https://doi.org/10.6084/m9.figshare.11673987
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 3a (2020). https://doi.org/10.6084/m9.figshare.11688816
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part3b(2020). https://doi.org/10.6084/m9.figshare.11688858
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 3c (2020). https://doi.org/10.6084/m9.figshare.11688870
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part3d(2020). https://doi.org/10.6084/m9.figshare.11688984
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 3e (2020). https://doi.org/10.6084/m9.figshare.11688891
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Part 4a (2020). https://doi.org/10.6084/m9.figshare.11699178
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: (2020). https://doi.org/10.6084/m9.figshare.11699253
- **Bian, J.**, Hiep, H. A.: A Tutorial on Verifying LinkedList using KeY: Video Material (2020). https://doi.org/10.6084/m9.figshare.c.4826589.v2
- Hiep, H. A., **Bian, J.**, de Boer, F.S., de Gouw, S.: A Tutorial on Verifying LinkedList using KeY: Proof Files (2020). https://doi.org/10.5281/zenodo.3613711

3.1 Introduction

Software libraries, such as the Java standard library, serve as the building blocks of many programs that run on the devices of many users every day. Given their widespread usage, even minor errors in these libraries can result in serious consequences, making it essential that such libraries be thoroughly tested and debugged before being deployed. Therefore, it is crucial to rigorously test and debug these libraries before deploying them. The focus should be on preempting issues, rather than merely addressing them after they occur.

This chapter intends to show how we take an existing Java program that is part of the Java standard library and study it closely to increase our understanding of it. If we are only interested in showing the presence of an issue with the program, e.g. that it lacks certain functionality, it suffices to show an example run that behaves unexpectedly. But to conclude that no unexpected behavior ever results from running the program first requires a precise specification of what behavior one expects, and further requires a convincing argument that all possible executions of the program exhibit that behavior.

We take a formal approach to both specification and reasoning about program executions, allowing us to increase the reliability of our reached conclusions to near certainty. In particular, the specifications we write are expressed in the JML, and our reasoning is tool-supported and partially automated by KeY. To the best of the authors' knowledge, KeY is the only tool that supports enough features of the Java programming language for reasoning about real programs, of which its run-time behavior crucially depends on the presence of features such as dynamic object creation, exception handling, integer arithmetic with overflow, `for` loops with early returns, nested classes (both static and non-static), inheritance, polymorphism, erased generics, etc.

As a demonstration of applying KeY to state-of-the-art, real software, we focus on Java's `LinkedList` class, for two reasons. First, a (doubly-)linked list is a well-known basic data structure for storing and maintaining unbounded data and has many applications: for example, in Java's secure sockets implementation.[1] Second, it has turned out that there is a 20-year-old bug lurking in its program, that might lead to security-in-depth issues on large memory systems, caused by the overflow of a field that caches the length of the list [38]. Our specification and verification effort is aimed at establishing the absence of this bug from a repaired program.

This chapter is based on the results as described in paper [38]. That paper provides a high-level overview of the specification and verification effort of the linked list class as a whole, for a more general audience. In the present chapter, more technical details on how we use the KeY theorem prover are given, and we give more detail concerning the production of the proofs. In particular, this tutorial consists of the online repository of proof files [39], and online video material that shows how to (re)produce the proofs [40]: these are video recordings of the interactive sessions in KeY that demonstrates exactly what steps one could take to complete

[1] e.g. see the JVM internal class `sun.security.ssl.HandshakeStateManager`.

the correctness proofs of the proof obligations generated by KeY from the method contracts.

We see how to set up a project and configure the KeY tool (Section 3.2). We then study the source code of the `LinkedList` to gain an intuitive understanding of its structure: how the instances look like, and how the methods operate (Section 3.3). We formulate, based on previous intuition, a *class invariant* in JML that expresses a property that is true of every instance (Section 3.4). An interesting property that follows from the class invariant, that is used as a separation principle, is described next (Section 3.5). To keep this presentation reasonably short, we further focus on the methods that pose the main challenges to formal verification, `add` and `remove`. We give a *method contract* for the `add` method that describes its expected behavior and we verify that its implementation is correct (Section 3.6). The difficulty level increases after we specify the `remove` method (Section 3.7), as its verification requires more work than for `add`. We study a deeper method that `remove` depends on, and finally use *loop invariants* to prove the correctness of `remove`.

3.2 Preliminaries

First we set up the project files needed to use KeY. The project files are available on-line [39]: these can be downloaded and include the KeY software version that we use. After unpacking the project files, we end up with the directory structure of Table 3.1.

```
linkedlist-tutorial
├── key-2.6.3.zip
├── LinkedList.key
├── src
│   └── java
│       └── util
│           ├── LinkedList.old
│           └── LinkedList.java
├── jre
│   └── java
│       └── ...
└── proof
    └── ...
```

Table 3.1: Directory structure of project files. The `src` directory contains the Java classes we want to specify and verify. The `jre` directory contains stub files, with specifications of unrelated classes. The `LinkedList.java` file is the source file we end up with after following this tutorial. The `proof` directory contains the completed proofs.

The original source file of `LinkedList.old` was obtained from OpenJDK version `jdk8-b132`. The original has been pre-processed: generic class parameters are removed, and all methods and nested classes irrelevant to this tutorial are removed. Both the removal of generics and the stub files in the `jre` folder were generated

Max. rule applications	1000	JavaCard	Off
Stop at	Default	Strings	On
Proof splitting	Delayed	Assertions	Safe
Loop treatment	Invariant	BigInt	On
Block treatment	Contract	Initialization	Disable static ...
Method treatment	Contract	Int Rules	Java semantics
Dependency contract	On	Integer Simpl. Rules	Full
Query treatment	Off	Join Generate	Off
Expand local queries	On	Model Fields	Treat as axiom
Arithmetic treatment	Basic	More Seq. Rules	On
Quantifier treatment	No splits	Permissions	Off
Class axiom rule	Off	Program Rules	Java
Auto induction	Off	Reach	On
User-defined taclets	All off	Runtime Exceptions	Ban
		Sequences	On
		Well-def. Checks	Off
		Well-def. Operator	L

Table 3.2: Proof search strategy and taclet options

automatically, using the Eclipse extensions for KeY. Repeating those steps is not necessary here.

Throughout the next sections, we modify the original source file and add annotations to formally specify its behavior, and helper methods for presenting intermediary lemmas. The annotations are usual Java comments and thus ignored when the file is read by a Java compiler. The helper methods introduce slight performance overhead (of calling a method that performs no operations, and immediately returning from it); it is clear that these do not change the original behavior of the program.

To produce proofs in KeY, the first step is to set up KeY's proof strategy and taclet options. This has to be done only once, as these taclet settings are stored per computer user. Sometimes, KeY overwrites or corrupts these settings if different versions are used. To ensure KeY starts in a fresh state, one can remove the `.key` directory from the user's home directory, and clean out preferences from the `.java/.userPrefs` directory by deleting the `de/uka/ilkd/key` hierarchy containing `prefs.xml` files.[1] Now start up KeY, and the example selection screen appears (if not, selecting File ▷ Load Example opens the same screen). Load any example, to enter proof mode.

First, we set up a proof strategy: this ensures the steps as done in the videos can be reproduced. On the left side of the window, change the settings in the Proof Search Strategy tab to match those of the left column in Table 3.2. We ensure to use particular taclet rules that correctly model Java's integer overflow semantics.

[1] On Windows, the preferences are instead stored in the Windows Registry. Use the `regedit` tool and clean out under `HKEY_CURRENT_USER\Software\JavaSoft\Prefs` or `HKEY_CURRENT_USER\Software\Wow6432Node\JavaSoft\Prefs` the same hierarchy. On Mac OS, open a terminal, change the directory to `~/Library/Preferences`, and delete `de.uka.ilkd.plist`

Select Options ▷ Taclet Options, and configure the options as in the right column in Table 3.2. The taclet options become effective after loading the next problem. We do that now: the main proof file `LinkedList.key` can be loaded, and a Proof Management window opens up, showing a class hierarchy and its methods. A method is not shown when no specifications are present. After giving the specifications below, more methods can be selected in this window.

3.3 Structure and behavior of `java.util.LinkedList`

In this section we walk through part of the source code of Java's linked list: see the `LinkedList.old` file in the artifact. Throughout this tutorial, we add annotations at the appropriate places. We finally obtain the `LinkedList.java` file.

Our `LinkedList` class has three attributes and a constructor (Listing 3.1): a `size` field, which stores a cached number of elements in the list, and two fields that store a reference to the first and last `Node`. The public constructor contains no statements: thus it initializes `size` to zero, and `first` and `last` to `null`.

```
package java.util;

public class LinkedList {

    transient int size = 0;
    transient Node first;
    transient Node last;

    public LinkedList() {}
```

Listing 3.1: The `LinkedList` class fields and constructor (begin of file).

The linked list fields are declared transient and package private. The transient flag is not relevant to our verification effort. The reason the fields are declared package private seems to prevent generating accessor methods by the Java compiler. However, in practice, the fields are treated as if they were private.

The `Node` class is defined as a private static nested class to represent the containers of items stored in the list (Listing 3.2). A static nested private class behaves like a top-level class, except that it is not visible outside the enclosing class. The nodes are doubly linked, that is, each node is connected to its preceding node (through field `prev`) and succeeding node (through field `next`). These fields contain `null` in case no preceding or succeeding node exists. The data itself is contained in the `item` field of a node.

```
private static class Node{
    Object item;
    Node next;
    Node prev;

    Node(Node prev, Object element, Node next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
  }
}
```

Listing 3.2: The `Node` nested class fields and constructor (end of file).

The method `add` for adding new elements to the list, takes one argument, the item to add (Listing 3.3). According to Java informal documentation for the `Collection`, this method is designed to always return `true`. The implementation immediately calls `linkLast`.

```
// implements java.util.Collection.add
    public boolean add(Object e) {
        linkLast(e);
        return true;
}
```

Listing 3.3: The method `add`.

The method `remove` for removing elements, also takes one argument, the item to remove (Listing 3.4). If that item was present in the list, then its first occurrence is removed and `true` is returned; otherwise, if the item was not present, then the list is not changed and `false` is returned.

```
// implements java.util.Collection.remove
    public boolean remove(Object o) {
        if (o == null) {
            for (Node x = first; x != null; x = x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node x = first; x != null; x = x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
```

```
            }
        }
        return false;
}
```

Listing 3.4: The method `remove`.

The presence of the item depends on whether the argument of the remove method is `null` or not. If the argument is `null`, then it searches for the first occurrence of a `null` item in the list. Otherwise, it uses the `equals` method, that every `Object` in Java has, to determine the equality of the argument with respect to the contents of the list. The first occurrence of an item that is considered equal by the argument is then returned. In both cases, the execution walks over the linked list, from the first node until it has reached the end. In the case that the node was found that contains the first occurrence of the argument, an internal method is called: `unlink`, and afterward `true` is returned.

Observe that the linked list is not modified if `unlink` is not called. Although not immediately obvious, this requires that the `equals` method of every object cannot modify our current linked list, for the duration of the call to `remove`. When the `remove` method is called with an item that is not contained in the list, either loop eventually exits, and `false` is returned.

The internal method `linkLast` changes the structure of the linked list (Listing 3.5). After performing this method, a new node has been created, and the `last` field of the linked list now points to it. To maintain structural integrity, also other fields change: if the linked list was empty, the `first` field now points to the new, and only, node. If the linked list was not empty, then the new last node is also reachable via the former last node's `next` field. It is always the case that all items of a linked list are reachable through the `first` field, then following the `next` fields, and also for the opposite direction.

```
void linkLast(Object e) {
        final Node l = last;
        final Node newNode = new Node(l, e, null);
        last = newNode;
        if (l == null) first = newNode;
        else l.next = newNode;
        size++;
}
```

Listing 3.5: The internal method `linkLast`.

The internal method `unlink` is among the most complex methods that alter the structure of a linked list (Listing 3.6). The method is used only when the linked list is not empty. Its argument is a node, necessarily one that belongs to the linked list. We first store the fields of the node in local variables: the old item, and the next and previous node references.

```
Object unlink(Node x) {
        final Object element = x.item;
        final Node next = x.next;
        final Node prev = x.prev;
        if (prev == null) {first = next;}
        else {
            prev.next = next;
            x.prev = null;
        }
        if (next == null) {last = prev;}
        else {
            next.prev = prev;
            x.next = null;
        }
        x.item = null;
        size−−;
        return element;
    }
```

Listing 3.6: The internal method `unlink`.

After the method returns, the argument fields are all cleared, presumably to help the garbage collector. However, other fields also change: if the argument is the first node, the `first` field is updated; if the argument is the last node, the `last` field is updated. The predecessor or successor fields `next` and `prev` of other nodes are changed to maintain the integrity of the linked list: the successor of the unlinked node becomes the successor of its predecessor, and the predecessor of the unlinked node becomes the predecessor of its successor.

3.3.1 Expected and unexpected method behavior

We draw pictures of linked list instances to understand better how the structure looks like over time. In Figure 3.1, we see three linked list instances. The left-most linked list is an object without any items: its `size` is zero. When we perform `add` with some item (it is not important what item), a new node is created and the first and last pointers are changed to point to the new node. Now the `prev` and `next` fields of the new node are `null`, indicating that there is no other node before and there is no other node after it. Also, the `size` field is increased by one. Adding another item further creates another node, that is linked up to the previous node properly; the `last` field is then pointed to the newly created node. In the third instance, suppose we would perform `remove` with the first item. We would then have to unlink the node, see the code of `unlink` in Listing 3.6: the new value of `first` becomes the value of `next` which is the last node, and the value of `prev` of the succeeding node becomes the value of `prev` of the node that is unlinked, which is `null`. We thus end up in a similar situation as the second instance (except for the item that may be different). Removing the last item brings us back to the situation depicted by the first instance.

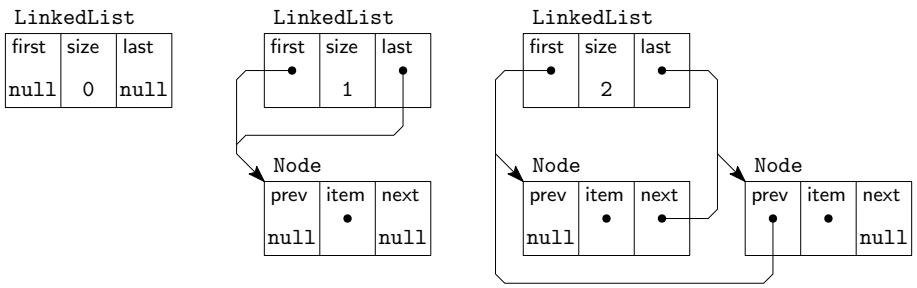

Figure 3.1: Three example linked lists: empty, with a chain of one node, and with a chain of two nodes. Items themselves are not shown.

An important aspect of the implementation of our linked list is the cached `size` field: it represents the number of nodes that form a chain between `first` and `last`. It turns out an overflow may happen under certain conditions [38]. Consider two facts: Java integer primitives are stored in signed 32-bit registers, and it is possible to create a chain that is larger than the maximum positive value that can be stored in such fields, $2^{31}-1$. Now, the cached size and the actual size no longer correspond. In the methods we have seen above, this seems to be no issue. But another method of the linked list may be used to demonstrate the key problem: `toArray`. The intention of `toArray` is to give back an array containing all the items of the list (see Listing 3.7). There are two problems: after the overflow has occurred and the `size` is negative, the `toArray` throws an unexpected `NegativeArraySizeException`. Also, after adding more items that bring the size back to a positive integer, e.g. adding $2^{32}+1$ items in total, the array is of the wrong (positive) size and cannot contain *all* items of the list, and an `IndexOutOfBoundsException` is thrown.

```
public Object[ ] toArray() {
    Object[ ] result = new Object[size];
    int i = 0;
    for (Node x = first; x != null; x = x.next)
        result[i++] = x.item;
    return result;
}
```

Listing 3.7: The method `toArray` has unexpected behavior.

3.3.2 Integer overflow bug

The List interface is part of the collection framework. The method `int size()` provided by class `LinkedList`, as shown in Listing 3.8.

```
 /** Returns the number of elements in this list. **/
private int size() {
     return size;
}
```

Listing 3.8: The `size` method.

Its (informal) specification states that it returns the number of elements in the list. This method is further specified by the interface Collection, where additionally it is stated that if the collection contains more than `Integer.MAX_VALUE` elements, then

`size()` returns `Integer.MAX_VALUE` (Java integers are bounded). Adding an element unconditionally increments the size, and thus can overflow, resulting in a negative size. This error breaks the contract of `size()` and propagates in various ways. For example, sorting a linked list first copies the list to a new array `Object[size]` by the method `toArray`, which thus may crash with a `NegativeArraySizeException`. Further, many methods of the `List` interface use an integer index to refer to an element. For very large lists with more than `Integer.MAX_VALUE` elements, it is unclear what the behavior should be. This is perhaps not surprising: triggering these bugs requires at least 2^{31} elements (i.e., at least 64 GiB of memory), and each element has 2^{32} values; the state space is too large to explore for fully automated analysis methods. Similarly, the smallest test cases that triggered the TimSort crash consist of inputs of 2^{26} ($\approx$ 67 million) elements, and each element has 2^{32} possible values. Such cases are clearly out of range for fully automated methods, such as model checking, which are typically based on finite abstractions of limited size.

As this case study involves real software, we have performed an analysis of existing Java code to estimate the use of `LinkedList` and the potential impact of a bug. A basic analysis of at least 140,000 classes,[1] found by sampling Java packages on Maven Central and in software distributions such as Red Hat Linux, reveals 1,677 cases of direct use of the `LinkedList` class where a constructor of `LinkedList` is invoked. As an overestimation of the potential reach of such instances, we find at least 37,000 usage call sites where some method of the `Collection`, `List`, or `Queue` interface is called. It is infeasible for us to analyze for each constructor call site where its resulting instance will be used. However, some usage of the `LinkedList` class occurs in potentially security-sensitive contexts, such as the library for checking certificates used by the Java secure socket implementation and checking the authenticity of the source of dynamically loaded bytecode.

3.3.3 Verification goal

We thus revise the source code and add a method that implements an overflow check (see Listing 3.9). The intention is that the overflow is signaled before it occurs, by throwing an exception. This ensures that the integrity of the linked list is always maintained. We modify the `add` method and perform a call to this `checkSize()` method before invoking `linkLast`.

```
// new method, not in original LinkedList
private void checkSize() {
    if (size == Integer.MAX_VALUE)
        throw new IllegalStateException(...);
}
```

Listing 3.9: A new internal method `checkSize`.

Our aim in this tutorial is to keep the discussion general enough, without losing interesting particular details. We apply step-wise refinements to our arguments,

[1] We have used the Rascal programming language [41] for loading `.jar`-files and analyzing their class file contents that revealed method call sites. We thank Thomas Degueule for his help in setting up this experiment.

where we start with a higher-level intuition and drill down on technical details as they become relevant. The reader can always see the video material; but, a high-level intuition seems essential for following along.

Our specification and verification goals comprise two points:

1. Specification captures the 'intended' behavior of the methods with respect to their structural properties: in particular, we abstract away from all properties about the contents of the linked list.

2. Verification ensures the overflow bug no longer happens in the revised linked list: the actual number of nodes and the cached size are always the same.

The first point depends on the aim of a verification attempt. Are we using the specification to verify the correctness of clients of the linked list? The properties of the contents of a linked list are essential. But, for our purpose of showing the absence of an overflow bug, we abstract away some properties of the contents of linked lists.

The second point deserves an introduction: how can we be sure that in every linked list the number of nodes and the value stored in the `size` field are the same? Can we keep the number of nodes bounded by what a Java integer can represent? Keeping this number in a ghost field is not sufficient, since the number of nodes depends on the success of a `remove` call: removing an item not present in the linked list should not affect its size, while removing an item that is present decreases its size by one. We refine: we could keep track of the items that are stored by the linked list. The structure to collect these items cannot be a set of items, since we could have duplicate items in the list. A multiset of items, the contents of a linked list, seems right: the size field of the linked list must be the same as the size of its contents, and the remove method is only successful if its argument is contained before the call.

However, working with multisets is quite unnatural, as the `remove` method removes the first item in the list. That can be refined by specifying the contents as a sequence of items instead. Although this could work in principle, a major difficulty when verifying the `remove` method is to give an argument as to why the method terminates. This requires knowledge of the linking structure of the nodes. We could relate the sequence of items to a traversal over nodes, saying that the first item of a linked list is found in the node by traversing `first`, and the item at index $0 < i < n$ is found in the node by traversing `first` and then `next` for $i - 1$ times. Formalizing this seems quite difficult, and as we shall see, not even possible in first-order logic.

Hence, we end up with our last refinement: we keep a sequence of nodes in a ghost field. From this sequence, one obtains the sequence of items. We relate the sequence of nodes to the linked list instance and require certain structural properties to hold the nodes in the sequence. The length of this sequence is the actual number of nodes, that we show to be equal to the cached size.

3.4 Formulating a class invariant

We now formalize a class invariant, thereby characterizing all linked list instances. We focus on unbounded linked lists first, as these are the structures we intend to model: so most properties are expressed using KeY's unbounded integer type \bigint. Only at the latest, we restrict the size of each linked list to a maximum, as a limitation imposed by the implementation. The setting in which to do our characterization is multi-sorted first-order logic. This logic is presented in a simplified form, leaving out irrelevant details (such as the heap): the full logic used by KeY is described in Chapter 2 of [29].

Consider the following sorts or type symbols: *LinkedList* for a linked list, *Node* for a node, *Object* for objects, and *Null* for `null` values. We have a type hierarchy, where *Null* is related to *LinkedList* and *Node*, and *LinkedList* and *Node* are both related to *Object*. This means that any object of sort *Null* is also an object of sort *LinkedList* and *Node*. Moreover, every object that is a linked list is also of type *Object*, and similar for nodes. We have the following signature: $\mathit{first} : \mathit{LinkedList} \to \mathit{Node}$ and $\mathit{last} : \mathit{LinkedList} \to \mathit{Node}$ for the `first` and `last` fields of linked lists, and $\mathit{prev} : \mathit{Node} \to \mathit{Node}$, $\mathit{item} : \mathit{Node} \to \mathit{Object}$, and $\mathit{next} : \mathit{Node} \to \mathit{Node}$ for the `prev`, `item` and `next` fields of nodes. Further, we assume there is exactly one object of *Null* sort, which is the `null` constant, for which the above functions are left undefined: `null` is a valid object of the `LinkedList` and `Node` Java types, but one may not access its fields.

We search for an axiomatization that characterizes linked list instances. One can find these axioms by trial and error. We start listing some (obvious) axioms:

1. $\forall x^{\mathit{LinkedList}}; (x \neq \texttt{null} \to (\mathit{first}(x) \neq \texttt{null} \leftrightarrow \mathit{last}(x) \neq \texttt{null}))$
   Every linked list instance either has both first and last set to `null`, or both point to some (possibly different) node.

2. $\forall x^{\mathit{LinkedList}}; (x \neq \texttt{null} \to (\mathit{first}(x) \neq \texttt{null} \to \mathit{prev}(\mathit{first}(x)) = \texttt{null}))$
   The predecessor of the first node of a linked list is set to `null`.

3. $\forall x^{\mathit{LinkedList}}; (x \neq \texttt{null} \to (\mathit{last}(x) \neq \texttt{null} \to \mathit{next}(\mathit{last}(x)) = \texttt{null}))$
   The successor of the last node of a linked list is set to `null`.

4. $\forall x^{\mathit{Node}}; (x \neq \texttt{null} \to (\mathit{prev}(x) \neq \texttt{null} \to \mathit{next}(\mathit{prev}(x)) = x))$
   Every node that has a predecessor, must be the successor of that predecessor.

5. $\forall x^{\mathit{Node}}; (x \neq \texttt{null} \to (\mathit{next}(x) \neq \texttt{null} \to \mathit{prev}(\mathit{next}(x)) = x))$
   Every node that has a successor, must be the predecessor of that successor.

These axioms are not yet sufficient: consider a linked list, in which its first and last nodes are different and both have neither a predecessor nor a successor. This linked list should not occur: intuitively, we know that the nodes between first and last are all connected and should form a doubly-linked 'chain'. Moreover, for every linked list, this chain is necessarily finite: one can traverse from first to last by following the next reference a finite number of times. This leads to a logical difficulty.

Proposition 1. It is not possible to define the reachability of nodes of a linked list

in first-order logic.

Proof 1. *Let x be a linked list and y a node: there is no formula $\phi(x, y)$ that is true if and only if $next^i(first(x)) = y$ for some integer $i \geq 0$.*[1] *Suppose towards contradiction that there is such a formula $\phi(x, y)$. Now consider the infinite set Δ of first-order formulas $\{\phi(x, y)\} \cup \{\neg(next^i(first(x)) = y) \mid 0 \leq i\}$. Let Γ be an arbitrary finite subset of Δ. Consider that there must exists some j such that $\Gamma \subseteq \{\phi(x, y)\} \cup \{\neg(next^i(first(x)) = y) \mid 0 \leq i < j\}$, so we can construct a linked list with j nodes, and we interpret x as that linked list and y as the last node. Clearly $\phi(x, y)$ is true as the last node is reachable, and all $\neg(next^i(first(x)) = y)$ is true for all $0 \leq i < j$ because j is not reachable within i steps from the first node. Since Γ is arbitrary, we have established that all finite subsets of Δ have a model. By compactness, Δ must have a model too. However, that is contradictory: no such model for Δ can exist, as neither $\phi(x, y)$ and $next^i(first(x)) \neq y$ for all integers $0 \leq i$ can all be true.*

We extend our signature to include other sorts: sequences and integers. These sorts are interpreted in the standard model. A schematic rule to capture integer induction is included in KeY (see [29, Section 2.4.2]). Sequences (see [29, Chapter 5.2]) have a non-negative integer length n and consist of an element at each position $0 \leq i < n$. We write $\sigma[i]$ to mean the ith element of sequence σ, and $\ell(\sigma)$ to mean its length n.

Intuitively, each linked list consists of a sequence of nodes between its first and last node. Let $\text{instanceof}_{Node} : \text{Object}$ be a built-in predicate that states that the object is not `null` and of sort *Node*. A *chain* is a sequence σ such that:

(a) $\forall i^{\text{int}}; (0 \leq i < \ell(\sigma) \rightarrow \text{instanceof}_{Node}(\sigma[i]))$
All its elements are nodes and not `null`

(b) $\forall i^{\text{int}}; (0 < i < \ell(\sigma) \rightarrow prev(\sigma[i]) = \sigma[i-1])$
The predecessor of the node at position i is the node at position $i - 1$

(c) $\forall i^{\text{int}} : (0 \leq i < \ell(\sigma) - 1 \rightarrow next(\sigma[i]) = \sigma[i+1])$
The successor of the node at position i is the node at position $i + 1$

Let $\phi(\sigma)$ denote the above property that σ is a chain. If $\ell(\sigma) = 0$ then $\phi(\sigma)$ is vacuously true: the empty sequence is thus a chain. We now describe properties $\psi_1(\sigma, x)$ and $\psi_2(\sigma, x)$ that relate a chain σ to a linked list x. These denote the following intuitive properties: there is no first and last node and the chain is empty, or the chain is not empty and the first and last node are the first and last elements of the chain.

$$\psi_1(\sigma, x) \equiv (\ell(\sigma) = 0 \wedge first(x) = last(x) = \texttt{null})$$
$$\psi_2(\sigma, x) \equiv (\ell(\sigma) > 0 \wedge first(x) = \sigma[0] \wedge last(x) = \sigma[\ell(x) - 1])$$

6. $\forall x^{LinkedList}; (x \neq \texttt{null} \rightarrow \exists \sigma^{\text{sig}}; (\phi(\sigma) \wedge (\psi_1(\sigma, x) \vee \psi_2(\sigma, x))))$
Every linked list necessitates the existence of a chain of either property

[1] $next^i$ is not a function symbol in first-order logic but an abbreviation of a finite term built by iteration of i times *next*, where $next^0(x) = x$ and $next^i(x) = next(next^{i-1}(x))$ for all $i > 0$.

Further, we require that the size field of the linked list and the length of the chain are the same: this property is essential to our verification goal. The size field is modeled by the function $size : LinkedList \to int$, and we require its value (1) to equal the length of the chain, and (2) to be bounded by the maximum value stored in a 32-bit integer. In formulating the above properties in JML, we skolemize the existential quantifier using a ghost field: see Listing 3.10. This has the additional benefit that we can easily refer to the chain ghost field in specifications.

```
/*@ nullable @*/ transient Node first;
/*@ nullable @*/ transient Node last;
//@ private ghost \seq nodeList;
/*@ invariant
  @ nodeList.length == size &&
  @ nodeList.length <= Integer.MAX_VALUE &&
  @ (\forall \bigint i; 0 <= i < nodeList.length;
  @ nodeList[i] instanceof Node) &&
  @ ((nodeList == \seq_empty && first == null && last == null)
  @ || (nodeList != \seq_empty && first != null &&
  @ first.prev == null && last != null &&
  @ last.next == null && first == (Node)nodeList[0] &&
  @ last == (Node)nodeList[nodeList.length-1])) &&
  @ (\forall \bigint i; 0 < i < nodeList.length;
  @ ((Node)nodeList[i]).prev == (Node)nodeList[i-1]) &&
  @ (\forall \bigint i; 0 <= i < nodeList.length-1;
  @ ((Node)nodeList[i]).next == (Node)nodeList[i+1]);
  @*/
```

Listing 3.10: The class invariant of `LinkedList` expressed in JML.

The class invariant is implicitly required to hold for the `this` object when invoking methods on a linked list instance. In particular, for the constructor of the linked list, the class invariant needs to be established after it returns. In Listing 3.11, we state that the constructor always constructs a linked list instance for which its chain is empty. The proof of correctness follows easily: at construct time, the fields (including the ghost field) of the linked list instance are initialized with their default values. This means the size is zero, and the first and last references are `null`, and the ghost field is the empty sequence.

```
/*@
  @ public normal behavior
  @ ensures nodeList == \seq_empty;
  @*/
public LinkedList() {}
```

Listing 3.11: The method contract of the constructor of `LinkedList` in JML.

For verifying the constructor above, see the video [42, 0:23–0:53], where the relevant video material is between timestamps 0:23 and 0:53.

3.5 The acyclicity property

An interesting consequence of the class invariant is the property that traversal of only `next` fields is acyclic. In other words, following only `next` references of any node that is present in a chain never reaches itself. The acyclicity property implies there is a number of times to follow the `next` reference until the last node is reached. For the last node, this number is zero (the last node is already reached). A symmetric property holds for `prev` too.

We logically specify the acyclicity property as follows. Let σ be the chain of a non-empty linked list x for which the class invariant holds. The following holds:

$$\forall i^{\text{int}}; (0 \leq i < \ell(\sigma) - 1 \rightarrow \forall j^{\text{int}}; (i < j < \ell(\sigma) \rightarrow \sigma[i] \neq \sigma[j]))$$

Let n abbreviate $\ell(\sigma)$. By contradiction: assume there are two indices, $0 \leq i < j < n$, such that the nodes $\sigma[i]$ and $\sigma[j]$ are equal. Then it must hold that for all k such that $j \leq k < n$, the node $\sigma[k]$ is equal to the node $\sigma[k - (j - i)]$: by induction on k. Base case: if $k = j$, then node $\sigma[j]$ and node $\sigma[j - (j - i)]$ are equal by assumption, since $\sigma[j - (j - i)] = \sigma[i]$. Induction step: suppose node at $\sigma[k]$ is equal to the node at $\sigma[k - (j - i)]$. We must show if $k + 1 < n$ then node $\sigma[k + 1]$ equals node $\sigma[k + 1 - (j - i)]$. This follows from the fact that $\sigma[k + 1] = \mathit{next}(\sigma[k])$ and $\sigma[k + 1 - (j - i)] = \mathit{next}(\sigma[k - (j - i)])$ for $k < n - 1$, since σ is a chain and the chain property (c) of last section. Now we have established, for all $j \leq k < n$, node $\sigma[k]$ equals node $\sigma[k - (j - i)]$. In particular, this holds when k is $n - 1$, the index of the last node: so we have $\sigma[n - 1] = \sigma[n - 1 - (j - i)]$. Since the difference $(j - i)$ is positive, we know $\sigma[n - 1 - (j - i)]$ is not the last node. By the linked list property 3 we have $\mathit{next}(\mathit{last}(x)) = \texttt{null}$ and by $\psi_2(\sigma, x)$ we have $\mathit{last}(x) = \sigma[n-1]$: so we have $\mathit{next}(\sigma[n - 1]) = \texttt{null}$. By the chain properties (c) and (a) we have $\mathit{next}(\sigma[n-1-(j-i)]) = \sigma[n-(j-i)]$ and instanceof$_{\mathit{Node}}(\sigma[n-(j-i)])$, respectively. From the latter we know $\sigma[n-(j-i)] \neq \texttt{null}$. So we have $\mathit{next}(\sigma[n-1-(j-1)]) \neq \texttt{null}$. But this is a contradiction: if nodes $\sigma[n - 1]$ and $\sigma[n - 1 - (j - i)]$ are equal then their `next` fields must also have equal values, but $\mathit{next}(\sigma[n - 1]) = \texttt{null}$ and $\mathit{next}(\sigma[n - 1 - (j - i)]) \neq \texttt{null}$!

For verifying the lemma as formalized in Listing 3.12, see the video [43].

```
/*@ public normal_behavior
  @ requires true;
  @ ensures (\forall \bigint i;
  @ 0 <= i < (\bigint)nodeList.length - (\bigint)1;
  @ (\forall \bigint j; i < j < nodeList.length; nodeList[i] != nodeList[j]));
  @*/
private /*@ strictly_pure @*/ void lemma_acyclic() {}
```

Listing 3.12: The method of a lemma added to `LinkedList` expressed in JML.

3.6 The `add` method

Due to the revision of the source code, the add method now calls `checkSize` first (see Listing 3.9) to ensure that the size field does not overflow when we add another

item. This means that the add method has two expected behaviors: the normal behavior when the length of the linked list is not yet at its maximum, and the exceptional behavior when the length of the linked list is at its maximum.

In the normal case, we expect the `add` method to add the given argument as an item to the linked list. Thus the sequence of nodes must become larger. We further specify the position where the item is added: at the end of the list. If add returns normally, it returns true. In the exceptional case, we expect that an exception is thrown. We formalize the contract for `add` in Listing 3.13.

```
/*@ private normal_behavior
  @ requires nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
  @ ensures
  @ nodeList == \seq_concat(\old(nodeList),
  @ \seq_singleton(nodeList[nodeList.length−1])) &&
  @ ((Node)nodeList[nodeList.length−1]).item == e && \result;
  @private exceptional_behavior
  @ requires nodeList.length == Integer.MAX_VALUE;
  @ signals_only IllegalStateException;
  @ signals (IllegalStateException e) true;
  @*/
public boolean add(/*@ nullable @*/ Object e) {
    checkSize(); // new
    linkLast(e);
    return true;
}
```

Listing 3.13: The `add` method with its method contract expressed in JML.

Since the `add` method calls the deeper methods `checkSize` and `linkLast`, we may employ their method contracts when verifying this method. So, before we verify `add`, we specify and verify these methods first.

We expect `checkSize` to throw an exception if the length of the linked list is too large to add another element, and it returns normally otherwise: see Listing 3.14 for its specification. Verification of `checkSize` in both normal and exceptional cases is done automatically by KeY, as can be seen in [42, 0:54–1:24].

```
/*@  private exceptional_behavior
  @  requires nodeList.length == Integer.MAX_VALUE;
  @  signals_only IllegalStateException;
  @  signals (IllegalStateException e) true;
  @  private normal_behavior
  @  requires nodeList.length != Integer.MAX_VALUE;
  @  ensures true;
  @*/
```

Listing 3.14: The method contract in JML of the `checkSize` method.

For the `linkLast` method, we assume that the length of the linked list is smaller than its maximum length, so we can safely add another node without causing an overflow

of the size field. When adding a new node, the resulting chain now is an extension of the previous chain, and additionally the class invariant holds afterwards—this is an implicit post-condition. Since we modify the chain, we need a `set` annotation that changes the ghost field.

```
/*@
  @ normal_behavior
  @ requires nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
  @ ensures nodeList == \seq_concat(\old(nodeList),
  @ \seq_singleton(nodeList[nodeList.length-1])) &&
  @ ((Node)nodeList[nodeList.length-1]).item == e;
  @*/
void linkLast(/*@ nullable @*/ Object e) {
    final Node l = last;
    final Node newNode = new Node(l, e, null);
    last = newNode;
    if (l == null) first = newNode;
    else l.next = newNode;
    size++;
    //@ set nodeList = \seq_concat(nodeList,\seq_singleton(last));
}
```

Listing 3.15: The `linkLast` method with its method contract expressed in JML.

The verification of this method is no longer fully automatic, see [42, 1:25–6:52].

Observe that there are two different situations we have to deal with: either the linked list was empty, or it was not. If the linked list was empty, then `last` is `null`, and we not only set the `last` field but also the `first`. Otherwise, if the linked list was not empty, we update the former last node to set its `next` field. The challenge is to prove that the class invariant holds after these heap updates, knowing that the class invariant holds in the before heap. The main insight is that the creation of a new node does not alias with any of the existing nodes, and that the modification of the `next` field only affects the old last node. Intuitively, we have a proof situation with two heaps as depicted in Figure 3.2.

The properties (b) on page 33, that fixes `prev` fields to point to the previous node in the sequence, and (c), that fixes `next` fields to point to the next node in the sequence, of the chain are the remaining goals in [42, 3:58]. Proving (b) is straightforward if one makes a distinction between the old nodes and the new node. Proving (c) in the 'heap after' involves two cases: either the index is between 0 and less than $\ell(\sigma) - 2$, or it used to be the last node and now has index $\ell(\sigma) - 2$. In the former case, the heap update has no effect, as we can show that these nodes are separate from the old last node because they differ in the old value of the next field. In the latter case, the heap update can be used to prove the property directly.

Finally, we can verify the `add` method: see [42, 6:58] for the normal behavior case, and [42, 8:09] for the exceptional behavior case.

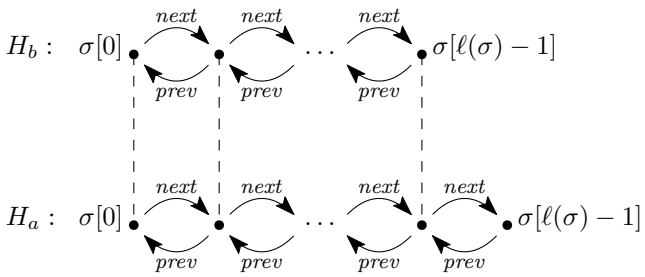

Figure 3.2: The heap before (H_b) consists of an arbitrary chain of nodes. In the heap after (H_a) the dashed lines show which objects are identical to the heap before. The old last node at $\sigma[\ell(\sigma)-1]$ has a different value for its *next* field in the heap after: this must be the result of a heap update. The new last node is not created in the heap before: indeed, it is the result of creating a new node.

3.7 The `remove` method

The `remove` method takes as an argument an object; it searches the linked list for the first node which contains the argument as an item. If found, it unlinks the node from the linked list. Using this intuition, we specify the `remove` method contract. Like with the `add` method, there is a deeper method that is called, `unlink`, which we have to specify and verify first.

An immediate difficulty in specifying the `remove` method contract is that its intended behavior depends on the behavior of the `Object.equals` method. Namely, the informal Java documentation states that the first element occurrence in the list that is 'equal to' the argument must be removed. Equality can be user-defined by overriding the `equals` method! We solve this difficulty by assuming a method contract for the equality method, see Listing 3.16.

```
/*@ public normal_behavior
  @ requires true;
  @ ensures \result == self.equals(param0);
  @*/
public /*@ helper strictly_pure @*/
boolean equals(/*@ nullable @*/ Object param0);
```

Listing 3.16: The `equals` stub method with its method contract expressed in JML.

We declare the equality method to be strictly pure, which implies that it must be a side-effect-free and terminating method (see [29, Section 7.3.5]). Each strictly pure method is also directly accessible as an observer symbol (a function symbol) that can be used in specifications (see [29, Section 8.1.2]). However, no obvious relation between the possibly overridden equality method and its observer symbol is present. The intention of the contract given in Listing 3.16 is to relate the outcome of the method call of `equals` to the observer symbol *equals*, and this furthermore requires that the implementation is deterministic.

The ramifications of adding this assumed contract are not clear. We note that there are Java classes for which equality is not terminating under certain circumstances. Even `LinkedList` itself does not have terminating equality, where two linked lists that contain each other may lead to a `StackOverflowError` when testing their

equality. This example is described in the Javadoc [44] of the linked list: "Some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself." Another approach is to specify the outcome of equality as referential equality only, e.g. see [10, Section 4.4].

Now we can specify the behavior of `remove`. It can be seen as consisting of two cases: either its result is `true` and it has removed the first item equal to its argument from the list, or its result is `false` and the argument was not found and thus not removed. In the first case, the number of elements decreases by one. In the second case, the number of elements in the linked list remains unchanged. See Listing 3.17.

```
/*@
  @ public normal_ behavior
  @ requires true;
  @ ensures \result == false ==>
  @ (\forall \bigint i; 0 <= i < \old(nodeList.length);
  @ (o==null ==> \old(((Node)nodeList[i]).item) != null) &&
  @ (o!=null ==> !\old(o.equals(((Node)nodeList[i]).item)))) &&
  @ nodeList == \old(nodeList);
  @ ensures \result == true ==>
  @ (\exists \bigint j; 0 <= j < \old(nodeList.length);
  @ (\forall \bigint i; 0 <= i < j;
  @ (o==null ==> \old(((Node)nodeList[i]).item) != null) &&
  @ (o!=null ==> !\old(o.equals(((Node)nodeList[i]).item)))) &&
  @ nodeList == \seq_concat(\old(nodeList)[0..j],
  @ \old(nodeList)[j+1..\old(nodeList.length)]) &&
  @ (o==null ==> \old(((Node)nodeList[j]).item) == null) &&
  @ (o!=null ==> \old(o.equals(((Node)nodeList[j]).item))));
  @*/
```

Listing 3.17: The `remove` method contract expressed in JML.

It is important to note that we make use of JML's `\old` operator to refer to the equality observer symbol in the old heap. Using equality in the new heap is a different observation, and it should not be possible to verify the remove method in this case. To see why, consider two linked list instances x and y: we add x to itself, and to y we add x and then y. Now we perform the `remove` operation on y with y as argument. Clearly, x and y are not equal, because they have a different length. But the second item is y itself, and y equals y, so it is removed: see Figure 3.3. In the resulting heap, both x and y contain x as the only item: thus, x and y are equal. If we would observe equality in the new heap, then the implementation is incorrect: the item to remove should not be the second but the first!

Before we can verify the `remove` method, we must specify and verify its deeper method: `unlink`. Within the method of unlink, we have to update the chain ghost field as well, to remove a node from the sequence, so we add a set annotation to the method body. Additionally, we make use of the lemma `lemma_acyclic` by calling it as the first statement of the method. See Listing 3.18. This method call is also

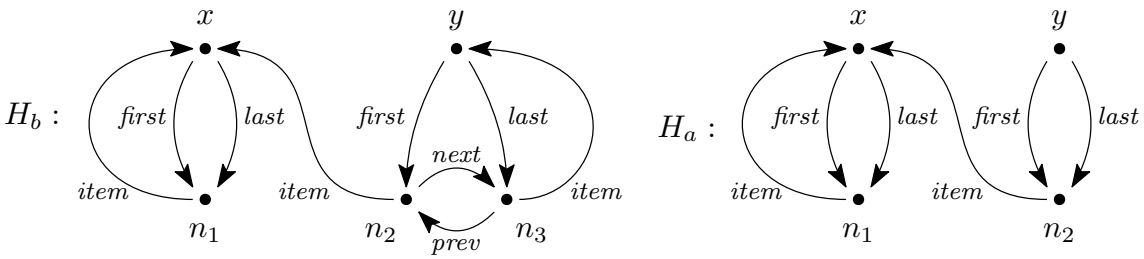

Figure 3.3: The situation (H_b) before `remove` is invoked on y with argument y, and after (H_a). The result of this operation is that y's second node n_3 is unlinked, hence the first node n_2 becomes the last node and its next pointer is cleared: now x and y are equal because they have the same length, and they have the same item, namely x.

not present in the original definition for `unlink`, but we already argued that it does not affect behavior.

```
/*@ normal_behavior
  @ requires nodeList != \seq_empty &&
  @ 0 <= nodeIndex < nodeList.length &&
  @ (Node)nodeList[nodeIndex] == x;
  @ ensures \result == \old(x.item) &&
  @ nodeList == \seq_concat(\old(nodeList)[0..nodeIndex],
  @ \old(nodeList)[nodeIndex+1..\old(nodeList).length]) &&
  @ nodeIndex == \old(nodeIndex);
  @*/
/*@ nullable @*/ Object unlink(Node x) {
    lemma_acyclic(); // new
    //@ set nodeList = \seq_concat(\dl_seqSub(nodeList,0,nodeIndex),
\dl_seqSub(nodeList,nodeIndex+1,\dl_seqLen(nodeList)));
    // rest of method body
    ...
```

Listing 3.18: The first part of method `unlink` and its method contract. Note that the @set annotation must not contain a new line, but kept on a single line: otherwise KeY 2.6.3 cannot load the source file. Here, /*@ @*/ does not work.

An interesting aspect of the specification of `unlink` is its use of a *ghost parameter*. Although KeY does not directly support ghost parameters, we can work around that by adding the parameter as a ghost field to our class:

```
//@ private ghost \bigint nodeIndex;
```

Its value is left undefined for most of the lifetime of the linked list until we are about to invoke `unlink`. In particular, the ghost parameter contains the index of the node argument, thereby requiring that the node object passed in is part of the chain. In the following discussion, let I be the node index ghost parameter and σ the chain of the linked list: then $\sigma[I]$ is assumed to be the node argument of the method `unlink`.

The verification of the unlink method is not fully automatic, see the five videos

[45, 46, 47, 48, 49].

Verifying unlink consists of four main cases: these correspond to the possible branches of the two if-statements (see Listing 3.6). The challenge again is to reestablish the class invariant in the heap after the method completes. The main insight is that, by the acyclicity property, all the nodes are separate: this allows us to distinguish the heap updates to apply only to the node that is actually affected while leaving the other nodes equal to the situation in the heap before. The three important cases are depicted in Figure 3.4, Figure 3.5, Figure 3.6 (compare with Figure 3.2).

1. Suppose the test of both if-statements evaluate to true: for node x, it holds that $\text{next}(x) = \texttt{null}$ and $\text{prev}(x) = \texttt{null}$. Then we know the list consists of exactly one node, as the node we are unlinking is the first and the last node. So I cannot be larger than 0. In the case the node index is zero, the class invariant is proven automatically [45, 7:25].

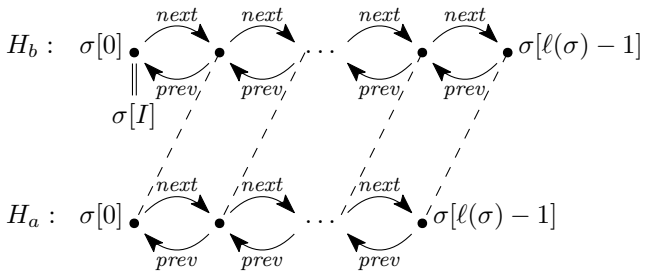

Figure 3.4: The heap before (H_b) consists of a chain σ with $\ell(\sigma) \geq 2$. The dashed lines show which objects are identical in the heaps. Interpreting $\sigma[I]$ in the new heap gives $\sigma[I+1]$ in the old heap, as I does not change. The `first` field of the linked list has changed (not shown), and the second node in the old heap now has `null` as `prev`. Moreover, the `next` and `prev` fields of the unlinked node have been set to `null` (not shown).

2. Suppose the test of the first if-statement evaluates to true, but the test of the second if-statement evaluates to false: for node x, it holds that $\text{next}(x) \neq \texttt{null}$ and $\text{prev}(x) = \texttt{null}$. We thus know that the list consists of at least two nodes, and it is the first node we are unlinking. Thus, I cannot be larger than 0. If the node index is zero, the class invariant is not proven automatically [46, 2:40], but we have two open goals corresponding to the chain properties (b) and (c), cf. proof of `linkLast`. Our situation is different now, see Figure 3.4. Here our insight applies: because of acyclicity, we know all nodes are different. Thus, an update of $\sigma[I]$'s fields does not affect the other nodes. When proving (c) this is sufficient as no next field of nodes in the new chain is changed compared to the old heap [46, 10:14-15:28]. When proving (b), we furthermore make a case distinction between the new first node and the other nodes: the former follows from the heap update, the latter from the old invariant [46, 3:27-10:13].

3. Suppose the test of the first if-statement evaluates to false, and the test of the second to true: for node x, it holds that $\text{next}(x) = \texttt{null}$ and $\text{prev}(x) \neq \texttt{null}$. This means that the list consists of at least two nodes, and it is the last node we are unlinking. The proof is similar to the previous case: see Figure 3.5 and [47].

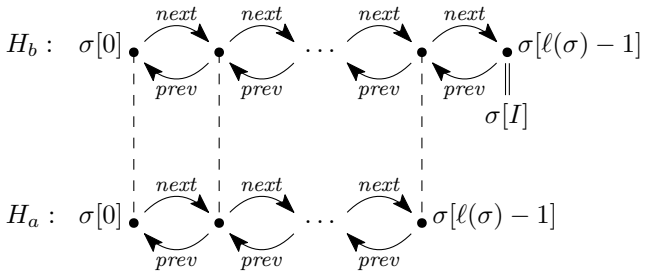

Figure 3.5: The heap before (H_b) consists of a chain σ with $\ell(\sigma) \geq 2$. The dashed lines show which objects are identical in the heaps. Interpreting $\sigma[I]$ in the new heap is invalid, as $I = \ell(\sigma)$ in the new heap. The `last` field of the linked list has changed (not shown), and the before last node in the old heap now has `null` as `next` field.

4. Suppose both tests of if-statements evaluate to false: for node x, it holds that $\text{next}(x) \neq \texttt{null}$ and $\text{prev}(x) \neq \texttt{null}$. This implies that the list consists of at least three nodes: where x is some 'interior' node. This part of the proof is the largest, as it involves many case distinctions. Up to the point where the class invariant is established in the heap after goes as before, except for (b) and (c). Keep in mind the situation as depicted in Figure 3.6, and see [48, 49].

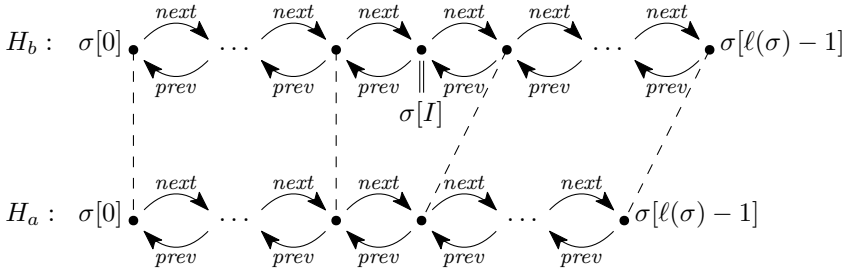

Figure 3.6: The situation (H_b) consists of a chain σ with $\ell(\sigma) \geq 3$. The index I remains unchanged in the heaps, thus $\sigma[I]$ in the heap after is equal to $\sigma[I+1]$ in the heap before. The following fields are updated in the new heap: the `next` field of the node at $\sigma[I-1]$ in the old heap becomes the node at $\sigma[I+1]$ in the old heap, and the `prev` field of the node at $\sigma[I+1]$ in the old heap becomes the node at $\sigma[I-1]$ in the old heap. In the new heap, these two nodes are present in succession in the chain, thus satisfying the chain properties (b) and (c).

We distinguish the two cases:

(b) Establishing the `prev` field property of the chain involves the following observation: there are three cases. First case, for all nodes at an index $0 \leq i < I$ in the old heap, we know they are identical to the nodes at the same index in the new heap. We know the heap is updated to assign the `prev` field of $\sigma[I+1]$ in the old heap, and by acyclicity we know this node is separate from the nodes all before $\sigma[I]$ in the old heap. Second case, $\sigma[I]$ interpreted in the new heap is identical to $\sigma[I+1]$ in the old heap, and precisely for this node the `prev` field was updated to become $\sigma[I-1]$ in the old heap (which is $\sigma[I]$ in the new heap). Third and last case, for all nodes at an index $I+1 < i < \ell(\sigma)$ in the old heap, we know they are identical to the nodes at $\sigma[i-1]$ in the new heap. Again, by acyclicity, we know that the node $\sigma[I+1]$ in the old heap is separate from the nodes

with a higher index, so we know their `prev` field cannot be affected by the update.

(c) Establishing the `next` fields property of the chain is very similar but with the index offset by one. Observe that the `next` field of $\sigma[I-1]$ in the old heap is updated to $\sigma[I+1]$ in the old heap. Thus the three cases are: first, for the nodes with index $0 \leq i < I-1$ in the old heap, second, for the node $\sigma[I-1]$ in the old heap, and third, for the nodes with index $I < i < \ell(\sigma)$ in the old heap.

Now that we have established that `unlink` removes a node from the chain while maintaining the class invariant, we can return to the verification of the `remove` method. The remove method iterates over the linked list until it has obtained a node to remove. However, the termination of this iteration is not obvious. Moreover, before invoking the `unlink` method, we need to specify the value of its ghost parameter: the index corresponding to the node. So, before we can verify the `remove` method, we add three kinds of annotations to its source: a ghost variable for maintaining the current index, a loop invariant that establishes termination and maintains the class invariant, and a set annotation before invoking the `unlink` method (see Listing 3.19).

```
public boolean remove(/*@ nullable @*/ Object o) {
    //@ ghost \bigint index = -1;
    if (o == null) {
        /*@ maintaining 0 <= (index + 1) &&
          @ (index + 1) <= nodeList.length;
          @ maintaining (\forall \bigint i; 0 <= i < (index + 1);
          @ ((Node)nodeList[i]).item != null);
          @ maintaining (index + 1) < nodeList.length ==>
          @ x == nodeList[index + 1];
          @ maintaining
          @ (index + 1) == nodeList.length <==> x == null;
          @ decreasing nodeList.length - (index + 1);
          @ assignable \strictly_nothing; */
        for (Node x = first; x != null; x = x.next) {
            //@ set index = index + 1;
            if (x.item == null) {
                //@ set nodeIndex = index;
                unlink(x);
                return true;
        } }
    } else {
        /*@ maintaining 0 <= (index + 1) &&
          @ (index + 1) <= nodeList.length;
          @ maintaining (\forall \bigint i; 0 <= i < (index + 1);
          @ !o.equals(((Node)nodeList[i]).item));
          @ maintaining (index + 1) < nodeList.length ==>
          @ x == nodeList[index + 1];
```

```
        @ maintaining
        @ (index + 1) == nodeList.length <==> x == null;
        @ decreasing nodeList.length − (index + 1);
        @ assignable \strictly_nothing; */
      for (Node x = first; x != null; x = x.next) {
        //@ set index = index + 1;
        if (o.equals(x.item)) {
          //@ set nodeIndex = index;
          unlink(x);
          return true;
  } } }
  return false;
}
```

Listing 3.19: The JML annotations of method `remove`. We use a slightly unnatural initial value for the index ghost variable since the KeY 2.6.3 parser does not recognize the @set annotation if it appears after the if-statement.

The verification of the above method is not fully automatic, see [50, 51]. The proof consists of two parts, corresponding to the branches of the if-statement. In the proof, one shows (among other properties) that the loop invariant holds initially, after each iteration, and at the end of the loop. It is important to note that $(index + 1)$ is equal to the length of the chain precisely when the end of the loop has been reached. This holds since we use the `next` field to traverse the chain, and only the last node has a `null` successor. Moreover, the distance to the last node decreases each iteration, and this distance is bounded from below by zero: thus the loop must terminate. Moreover, the loop is *strictly pure*, as it never modifies the heap in any of its completed iterations. The exceptional case is the last iteration in which the `remove` method returns early. Due to the early return, the loop invariant no longer needs to be shown (and so also not its heap purity). For reasons of limited space, further examination of its proof is left as a challenge to the reader.

3.8 Summary

In the chapter, we have shown the verification of two essential methods of Java's `LinkedList` class: `add` and `remove`. The original implementation contains an overflow bug (see Section 3.3.2), and we have looked at a revised version that imposes a maximum length of the list. Furthermore, we have set out to verify that the overflow bug indeed no longer occurs. Towards this end, we have formally specified a class invariant and method contracts, with two goals: establishing the absence of the overflow bug and capturing the 'essential' behavior of the methods with respect to the structural properties of the linked list. All methods have been formally verified [39] using the KeY theorem prover, and video material shows on [40].

This chapter aims to provide a comprehensive guide on using KeY and JML for the specification and verification of selected portions of Java libraries. We did a lot of work on recording the video materials, aiming to cater to both the beginning user,

the expert user, and the developer of KeY as a 'benchmark' for specification and (automatic) verification techniques.

Chapter 4

History-based reasoning about interfaces

Programming to interfaces is one of the core principles in object-oriented programming. However, current practical static analysis tools, including model checkers and theorem provers such as KeY, are primarily state-based. Since interfaces do not expose a state or concrete representation, a major question is how to support interfaces.

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

This chapter is based on the following publications:

- **Bian, J.**, Hiep, H. A., de Boer, F. S., de Gouw, S. (2023). Integrating ADTs in KeY and their application to history-based reasoning about collection. Formal Methods in System Design, 1-27.
- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). History-based specification and verification of Java collections in KeY. In Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings 16 (pp. 199-217). Springer International Publishing.

4.1 Introduction

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

However, some of the methods of the linked list implementation contain an interface type as a parameter and were out of the scope of the work in [5]. For example, we could take the `retainAll` method. Verification of `LinkedList`'s implementation of `retainAll` requires the verification of the inherited `retainAll` method from `AbstractCollection`. The implementation in `AbstractCollection` (see Listing 4.1) shows a difficult method to verify: the method body implements an interface method, acts as a client of the supplied `Collection` instance by calling `contains`, but it also acts as a client of the `this` instance by calling `iterator`. Moreover, as `AbstractCollection` is an abstract class and does not provide a concrete implementation of the interface, implementing `iterator` is left to a subclass such as `LinkedList`. Thus arises the need for an approach to specify interfaces that allows us to verify its (abstract) implementations and its clients.

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

Listing 4.1: A difficult method to verify: `retainAll` in `AbstractCollection`.

More generally, libraries form the basis of the "programming to interfaces" discipline, which is one of the most important principles in software engineering. Interfaces abstract from state and other internal implementation details, and aid modular program development. However, tool-supported programming logic and specification languages are predominantly state-based which as such cannot be directly used for interfaces. For example, JML is inherently state-based. JML mainly provides support for the specification of classes in terms of their fields, including so-called model fields that represent certain aspects of the data structure underlying the implementation.

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

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

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

The background of our approach is given in Sect. 4.2. An important use case, which leads us to formal requirements on interface specifications, is to reason about the correctness of clients, viz. programs that use instances of an interface by calling methods on it. In Sect. 4.3 we analyze concrete examples that motivate the design choices that lead us to the core of our approach: we associate to each instance of an interface a history that represents the sequence of method calls performed on the object since its creation. For each method call, the parameters and return value are recorded symbolically in the history. This crucially allows us to define abstractions over histories, called *attributes*, used to describe all possible behaviors of objects regardless of their implementation. Our methodology for embedding histories and attributes into the KeY theorem prover is described in Sect. 7.3.1. We explore two approaches: either encoding histories and attributes as Java objects or encoding them as abstract data types. We discuss these two approaches and compare their respective strengths and weaknesses across several aspects in the last section.

4.2 Background

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

global history. A global history is therefore not suitable for specifying and verifying Java libraries, since libraries are reused in many different client programs. Hence, in our setting, we tend towards using a local history for each object separately.[1]

Following the concept of information hiding, we assume that an object encapsulates its own state, i.e. other objects cannot directly access its fields, but only indirectly by calling methods. This is not a severe limitation: one can introduce getter and setter methods rather than reading and writing a field directly. But this assumption is crucial to enable any kind of (sound) reasoning about objects: if objects do not encapsulate their own state, any other object that has a reference to it can simply modify the values of the fields directly in a malicious fashion where the new internal state breaks the class invariant of the object without the object being able to prevent (or even being aware of) this. Roughly speaking, a class invariant is a property that all objects of the class must satisfy before and after every method call. Class invariants typically express the consistency properties of the object. For example, an instance of `ArrayList` has a `size` field that is supposed to represent the number of items in the list. A simple class invariant is $\texttt{size} \geq 0$. If the `ArrayList` does *not* encapsulate the `size` field, the following can happen:

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

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

Assuming encapsulation, each object has full control over its own internal state, it can enforce invariants over its own fields and its state can be completely determined by the sequence of method calls invoked on the object. How an object realizes the intended behavior of each method may differ per implementation: to a client of the object, the internal method body is of no concern, including any calls to other objects that may be done in the method body. We name the calls that an object invokes on other objects inside a method outgoing calls (their direction is out of the object, into another object), and we name the calls made to the object on methods it exposes incoming calls. The above discussion makes clear that the semantics of an object-oriented program can be described purely in terms of its behavior on incoming method calls. Indeed, formally, this is confirmed by Jeffrey and Rathke's work [22] which presents a fully abstract semantics for Java based on traces.

4.3 Specification and verification challenges for the `Collection` interface

In this section, we highlight several specification and verification challenges with histories that occur in real-world programs. We guide our discussion with examples based on `Collection`, the central interface of the Java Collection Framework.

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

However, note that our approach, and methodology in general, can be applied to all interfaces, as our discussion can be generalized from `Collection`.

A collection contains elements of type `Object` and can be manipulated independently of its implementation details. Typical manipulations are adding and removing elements, and checking whether they contain an element. Sub-interfaces of `Collection` may have refined behavior. In the case of interface `List`, each element is also associated with a unique position. In the case of interface `Set`, every element is contained at most once. Further, collections are extensible: interfaces can also be implemented by programs outside of the Java Collection Framework.

How do we specify and verify interface methods using histories?

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

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

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

For example, for two separate elements o and o',
$\texttt{add}(o) = \textbf{true},\ \texttt{add}(o') = \textbf{true},\ \texttt{add}(o') = \textbf{false},\ \texttt{remove}(o') = \textbf{true}$
is a history of some collection (where the left-most event happens first). The multiplicity of o in the multiset attribute of this history is 1 (there is one successful add event), and the multiplicity of o' is 0 (there is one successful add event and one successful remove event).

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

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

Consider the client program in Listing 4.2, where `x` is a `Collection` and `y` is an `Object`. To specify the behavior of this program fragment, we could now use the

multiset attribute to express that the content of the `Collection` instance `x` is not affected.

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

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

Another example of this challenge is shown in Listing 4.3: can we prove the termination of a client? For an arbitrary collection, it is possible to obtain an object that can traverse the collection: this is an instance of the `Iterator` interface containing the core methods `hasNext` and `next`. To check whether the traversal is still ongoing, we use `hasNext`. Subsequently, a call to `next` returns an object that is an element of the backing collection and continues the traversal. Finally, if all objects of the collection are traversed, `hasNext` returns `false`.

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

Listing 4.3: Iterating over the collection.

How do we deal with intertwined object behaviors?

Since an iterator by its very nature directly accesses the internal representation of the collection it was obtained from,[1] the behavior of the collection and its iterator(s) are intertwined: to specify and reason about collections with iterators a notion of *ownership* is needed. The behavior of the iterator itself depends on the collection from which it was created.

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

Consider the example in Listing 4.4, where the collection `x` is assumed non-empty. We obtain an iterator and its call to `next` succeeds (because `x` is non-empty). Consequently, we perform the calls as in Listing 4.2: this leaves the collection with the same elements as before the calls to `add` and `remove`. However, the iterator may become invalidated by a call that modifies the collection; then the iterator `it` is no longer valid, and we should not call any methods on it—doing so throws an exception.

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

Listing 4.4: Invalidating an iterator by modifying the owning collection.

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

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

call to `add` or `remove` (or any mutating method) on the collection invalidates all its iterators.

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

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

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

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

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

Listing 4.5: Invalidating an iterator by another iterator.

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

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

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

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

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

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

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

4.4 History-based reasoning approach

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

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

4.4.1 The executable history-based approach

The EHB approach is to embed histories and attributes in the KeY theorem prover [23] by encoding them as Java objects, thereby avoiding the need to change the KeY system itself. Interface specifications can then be written in the state-based

specification language JML [24] by referring to histories and their attributes to describe the intended behavior of implementations.

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

1. We introduce histories by Java classes that represent the inductive data type of sequences of events, and we introduce attributes of histories encoded by static Java methods. These attributes are defined inductively over the structure of a history. The attributes are used within the interface method contracts to specify the intended behavior of every implementation in terms of history attributes.

2. Attributes are deterministic and thus represent a function. Certain logical properties of and between attributes hold, comparable to an equational specification of attributes. These are represented by the method contracts associated with the static Java methods that encode the attributes.

3. Finally, we append an event to a history by creating a new history object in a static factory method. The new object consists of the new event as the head and the old history object as the tail. The contract for these static methods also expresses certain logical properties of and between attributes, of the new history related to the old history.

The practical specification and verification effort of a part of the `Collection` interface employing the EHB approach is detailed in Chapter 5.

4.4.2 The logical history-based approach

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

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

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

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

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

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

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

4.4.3 Comparative analysis

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

Logical representation

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

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

Expressiveness of attributes

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

Logical consistency

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

Verification complexity

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

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

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

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

4.5 Summary

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

Chapter 5

Executable history-based reasoning: a case study

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

This chapter is based on the following publications:

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

5.1 Introduction

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

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

5.2 History-based specification in KeY

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

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

5.2.1 The `History` class for `Collection`

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

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

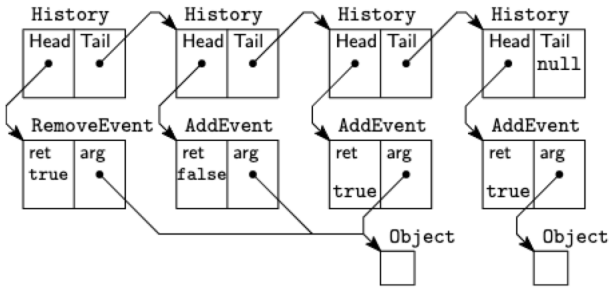

Figure 5.1: A number of history objects. The left-most represents the history of a collection in which `add` is called three times followed by a remove. Intuitively, this history captures the behavior of a set (the addition of an object already contained returns `false`).

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

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

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

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

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

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

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

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

5.2.2 Attributes of `History`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.3 The `Collection` interface

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

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

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

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

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

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

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

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

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

5.2.4 History-based refinement

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

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

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

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

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

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

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

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

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

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

5.3 History-based verification of `Collection`

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

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

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

intervention.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 5.11: Iterating over the collection.

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

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

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

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

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

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

5.4 Summary

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

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

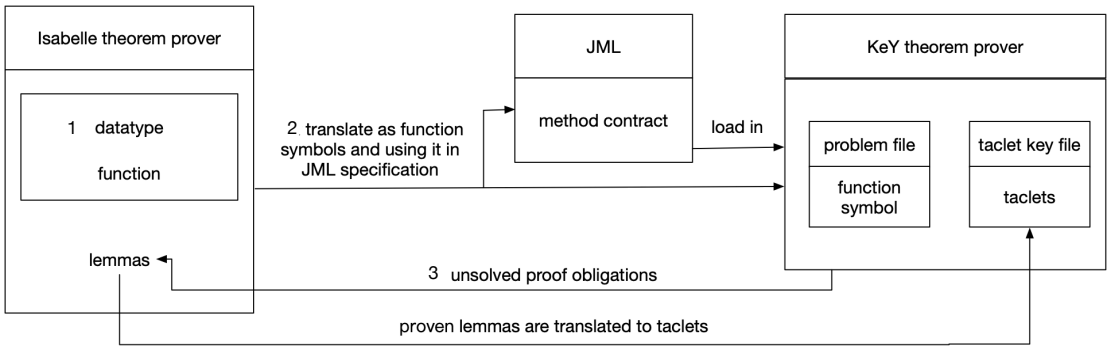

Figure 6.1: The workflow of integrating ADTs in KeY.

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):

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

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 $None : \alpha\ option$ 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 backend 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*.

```
\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 implementa-

tion 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:

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

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 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]):

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

Intuitively, the clauses of the *isValid* predicate 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 $\mathit{isIteratorValid}(h)$, 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`).

```
1  /** Ensures that this collection contains the specified element
2    * (optional operation).
3    * Returns true if this collection changed as a result of the call.
4    * Returns false if this collection does not permit duplicates and
5    * already contains the specified element. **/
6  /* @ public normal_ behavior
7      @ ensures \dl_ multiset(history(),o) ==
8       \dl_ multiset(\old(history()), o) + ((\result == true) ? 1 : 0);
9      @ ensures (\forall Object o1; o1 != o; \dl_ multiset(history(),o1) ==
10      \dl_ multiset(\old(history()), o1));
11     @ ensures \dl_ multiset(history(),o) > 0;
12     @ ensures \result == false ==>
13      (\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 $multiset(h', y) = multiset(h, y) - 1$ 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 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:

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

According to the definition of *iteratorSize*, it only adds 1 when executing the `next()` method, but the definition of *isIteratorValid* in 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].

Chapter 7

History-based reasoning about behavioral subtyping

Behavioral subtyping [28], a concept predicated on the behavior of objects, is a key principle in object-oriented programming. This principle ensures that a subtype should seamlessly substitute its supertype without affecting the desirable properties or expected behavior of a program. The importance of this concept is particularly evident in the development of type hierarchies and type systems of programming languages that enable polymorphism and inheritance.

In this chapter, we introduce a new history-based proof-theory for reasoning about behavioral subtyping in class and interface hierarchies. Our approach is based on a semantic definition of types in terms of sets of sequences of method calls and returns, so-called histories. Behavioral subtyping is then naturally defined semantically as a set-theoretic subset relation between sets of histories, modulo a *projection relation* that captures the syntactic subtype relation. The main contribution is a Hoare-style proof theory for the specification and verification of the behavioral subtyping relation in terms of histories, abstracting from the underlying implementation. Through the use of a banking example we show the practical applicability of our approach.

This chapter is based on the following publication and artifact:

- **Bian, J.**, Hiep, H.A., de Boer, F.S. History-based Reasoning about Behavioral Subtyping. (Submitted for publication.)
- **Bian, J.**, Hiep, H. A., de Boer, F. S. (2024). History-Based Reasoning about Behavioral Subtyping: Proof files.
  https://doi.org/10.5281/zenodo.10998227

7.1 Introduction

The *programming to interfaces* discipline is one of the most important principles in software engineering. This methodology allows the developer of client code to abstract away from internal implementation details, such as object state, thereby aiding modular program development. Type hierarchies support this principle in object-oriented design by allowing the declaration of new subtypes that inherit properties and behaviors from their supertypes, while also providing the flexibility to add or override specific features as needed. The concept of *behavioral subtyping* (which refers to subtyping based on behavior, in contrast to nominal subtyping and structural subtyping [77]) ensures that in clients one should be able to replace the use of a supertype by a subtype without causing unexpected behavior [27, 78]. This concept is employed in object-oriented programming to ensure software maintainability and robustness.

Histories, as defined in our previous work [53], are sequences of method calls performed on the object. We define the semantics of a type as a set of histories, thus abstracting from the underlying state/implementation. This allows to define the behavioral subtype relation semantically as subset relation between sets of histories, modulo a projection relation between histories that corresponds with the syntactic definition of the subtype relation. For the specification and verification of the behavioral subtype relation we introduce method contracts using Hoare triples that involve suitable user-defined abstractions over histories, called *attributes*. We discuss behavioral subtyping in three settings: class-class inheritance, class-interface inheritance, as well as interface-interface inheritance.

There has been numerous research on behavioral subtyping [79, 80, 81, 82], starting from the seminal work by Liskov and Wing [28], who point out that a subtype must adhere to the behavioral contracts of its supertype. To define the subtype relation, they introduced an *abstraction function* that maps the state of each subtype to a state of its corresponding supertype. The soundness of the substitution principle follows from two conditions: the precondition of the supertype implies the precondition of the subtype, and the postcondition of the subtype implies the postcondition of the supertype. The pre/postconditions of the subtype speak of the state of the subtype, whereas the pre/postconditions of the supertype speak of a different state: so the abstraction function takes a state of the subtype and maps it to a state of the supertype in such a way that these conditions hold. Is worth mentioning that in Liskov and Wing's work, they introduce a notion of *history constraint*, which is different from our notion of a history. Their history refers to temporal properties of objects, which are used to declare a relationship between pre-states and post-states preserved by any method of a type [83]. Leavens and Weih [84, 85, 86] present a technique for the modular reasoning about object-oriented programs, called *supertype abstraction*, which allows adding behavioral subtypes without reverification. However, their method is based on the assumption that each specified subtype relation constitutes a behavioral subtype. Demonstrating such behavioral subtyping requires again the use of an abstraction function. Although there have been several logics for the reasoning about object-oriented programs including a notion of behavioral subtyping, such as [87, 88, 89], they are all based on the abstraction function.

In the field of refinement calculus [90, 91, 92], which focuses on the stepwise transformation of an abstract specification into an executable implementation, one also uses the abstraction functions. These functions help in mapping implementation to specification, ensuring that each refinement step is correct.

In contrast to the above related work the history-based reasoning approach in this paper avoids formulating ad hoc abstraction functions between different state-based implementations. Instead it is based on a general semantic definition of the behavioral sub-type relation as a subset relation between sets of histories modulo a projection relation. Further, our proof method is based on the use of suitable user-defined history abstractions which allows for a modular verification of the proof obligations. Finally, our approach is applicable to both interfaces and classes, and allows reasoning about behavioral subtyping in settings that are typically absent in most related studies [84, 93, 94, 95].

The paper is intentionally written to introduce and motivate a new idea rather than to work out all the formal details. We discuss the methodology of history-based behavioral subtyping in Sect. 7.2. Our specification methods are presented in the context of history and attributes. In Sect. 7.3, we use a banking example to illustrate our approach. We provide only informal proofs for three particular subtype relations: interface-interface, interface-class, and class-class. The part of this example is proven using the KeY theorem prover [23] and Isabelle/HOL [33]. The verification workflow is based on our previous work [96].

7.2 Methodology

In object-oriented programming, a method signature consists of a list of parameter types and a return type. An interface contains a set of method signatures. A class consists of a set of field declarations and a set of method declarations.

The type hierarchy for classes and interfaces in languages with a nominal type system can be declared as below:

$$\textbf{interface } I \text{ [} \textbf{extends } I_1, I_2, \ldots, I_n \text{]}$$
$$\textbf{class } C' \text{ [} \textbf{extends } C \text{] [} \textbf{implements } I_1, I_2, \ldots, I_n \text{]}$$

An interface can extend zero or more interfaces, which is known as interface inheritance. When one interface extends another, it inherits all of the methods defined in its super interfaces, but it can also add new methods of its own. A class can inherit from multiple interfaces, by providing implementations for all methods defined in the interfaces. However, a class can only inherit from a *single* class. This is due to the fact that class inheritance is typically used for defining the (memory) structure of a class. Allowing multiple inheritance of classes can potentially lead to conflicts among class invariants [97] and ambiguity, as exemplified by the so-called diamond problem [98].

The basic behavioral notion of subtyping discussed in [28] is shown as in Fig. 7.1. A Hoare triple specification, denoted as $\{p\}\, m\, \{q\}$, consists of a method m, a pre-

condition p that describes the object state before the method is executed, and a postcondition q that describes the expected object state after the method is executed. In Fig. 7.1, we have $\{p\}\, m\, \{q\}$ on top and $\{p'\}\, m\, \{q'\}$ below, which represent the supertype and subtype specification of m, respectively, where m is a method inherited by the subtype from the supertype.

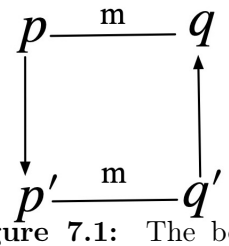

Figure 7.1: The behavioral notion of subtyping.

From the perspective of a client, we ensure before a method call that the precondition of the supertype holds, so that after dynamic dispatch where we jump to the implementation of the subtype, we also need that the precondition of the corresponding method in the subtype holds. After the execution of the subtype method finishes, it reaches the postcondition of the subtype's method. This postcondition should also imply the postcondition of the method in the supertype, since the client assumes that the postcondition of the supertype holds after the method returns. Moreover, if both types are classes then the invariant of the supertype must be preserved in the subtype.

However, typically the precondition and postcondition, given in some specification language, are intrinsically state-based and as such are not directly suitable for the specification of a state-hiding interface. In history-based reasoning, we introduce the concept of a history that can be seen as the most general abstraction of the state space of an interface. There are two approaches: the executable history-based (EHB) approach [53], and the logical history-based (LHB) approach [96]. In the former approach, histories become part of the run-time environment and are encoded as objects. In the latter approach, histories do not exist at run-time and are only introduced as bookkeeping devices for reasoning, similar to ghost variables. We proceed with the latter approach. In the LHB approach, histories are modeled as elements of an abstract data type (ADT). This means histories are immutable and inaccessible: no program can modify or even inspect a history value.

A history is a sequence of events. Every method is represented by a corresponding event type, that records the types of the parameters and the type of the return value. For technical convenience, we only regard normal returns from method calls as events. For each class and interface, we introduce a history type by defining it as an inductive data type of sequences of events.

Following the information hiding principle, we assume an object encapsulates its own state. Consequently, each object can enforce invariants over its own fields and its state can be completely determined by the sequence of method calls invoked on the object. Attributes are user-defined abstractions of histories that are in general defined inductively over the history. These attributes are used in method specifications to specify the intended behavior of implementations, and by using attributes the method specifications do not depend on the (hidden) state of an object.

The overall approach in history-based reasoning can be summarized by the following diagram, see Fig. 7.2. We will now provide more details on each of the components in Fig. 7.2.

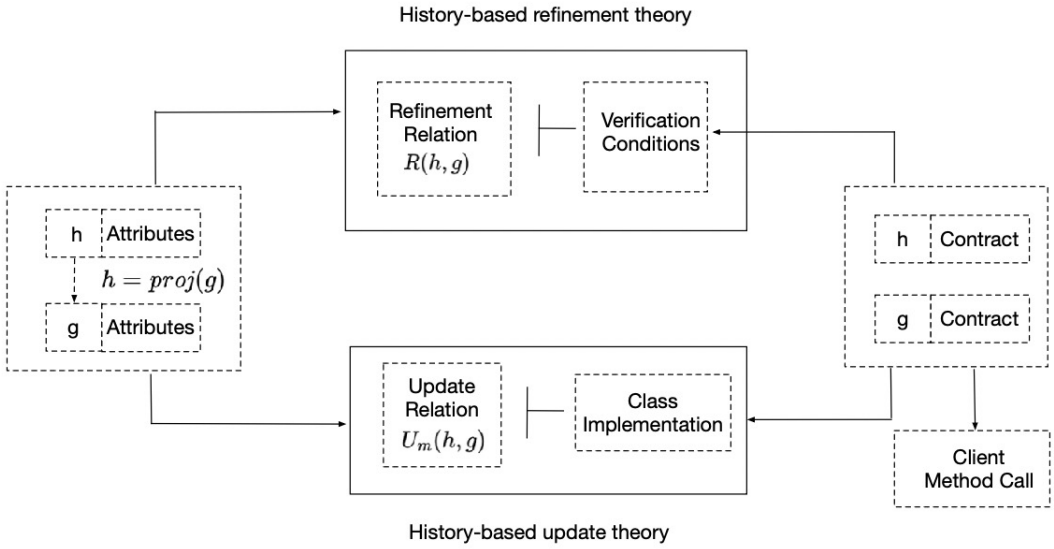

Figure 7.2: History-based reasoning about behavioral subtyping. In $R(h, g)$, h represents the history of the supertype and g represents the history of the subtype. In $U_m(h, g)$, h represents the history in the post-state, g represents the history in the pre-state and m is the corresponding method.

7.2.1 History-based refinement theory

To establish that the behavioral subtype relation holds between two types, we define a set of proof obligations between the preconditions and postconditions of methods inherited by a subtype. For a modular verification of these proof obligations we introduce a methodology that consists of two parts: verification of the refinement relation and, separately, verification of the proof obligations generated from method specifications, assuming the refinement relation. The method specifications refer to the attributes of the associated history, and abstract from the inductive definition of the history and its attributes. The *refinement relation* on the other hand captures logically the relationship between attributes of *different* histories, namely the histories of the supertype and the subtype. The axioms of the refinement relation itself, as a logical theory, should be established as logical consequences of the inductive definitions of the attributes and the projection relation.

This assumption can be justified as follows. When a method from a subtype that inherits from a supertype in the hierarchy is called, updates are made to both the histories of supertype and subtype. However, for methods only present in the subtype, updates are made only to the history of the subtype, while that of the supertype remains unchanged. This design choice is intentional to avoid the potential issues that may occur if the subtype is cast to the supertype. More general approaches, where the history of a subtype can be simulated by a history of the supertype, are out of scope in this paper. For any given histories h and g, where h is a projection of g, the user-defined refinement relation $R(h, g)$ describes a logical relation between the attributes of h and g, *abstracting* from their inductive definitions. Note that attributes in general may have different meanings when interpreted by the history of a supertype or by the history of a subtype.

The proof obligations, also called verification conditions, for interface-interface refinement, interface-class refinement and class-class refinement are shown below. For a supertype, h represents the history of the supertype, p represents the precondition and q represents the postcondition. For a subtype, g represents the history of the subtype, p' represents the precondition and q' represents the postcondition. For classes, I and I' denote the superclass and the subclass invariant, respectively. Class invariants in general describe a (logical) relation between the fields of a class and the attributes of the assoctated class. In proving the verification conditions for the pre- and postconditions of the inherited methods, the refinement relation is assumed. It should be noted that, by using the logical consequence relation $\vdash$, the history variables h and g are implicitly universally quantified (on both sides of $\vdash$). The refinement relation itself involves a separate proof obligation which is formulated by

$$h = proj(g) \rightarrow R(h, g)$$

That is, the logical relation between the attributes of the histories of the supertype and the subtype should follow from the projection relation and their inductive definitions.

Verification Condition IIR (Interface-Interface Refinement)**.**

$$R(h, g) \vdash (p \rightarrow p')$$
$$R(h, g) \vdash (q' \rightarrow q)$$

Verification Condition ICR (Interface-Class Refinement)**.**

$$R(h, g) \vdash (p \wedge Inv' \rightarrow p')$$
$$R(h, g) \vdash (q' \wedge Inv' \rightarrow q)$$

Verification Condition CCR (Class-Class Refinement)**.**

$$R(h, g) \vdash (Inv' \rightarrow Inv)$$
$$R(h, g) \vdash (p \wedge Inv' \rightarrow p')$$
$$R(h, g) \vdash (q' \wedge Inv' \rightarrow q)$$

7.2.2 Verifying method call and method implementation

In the usual manner, method calls are verified in terms of the corresponding method specification (as determined by the *static* type of the callee expression). This involves the usual substitution of the formal parameter by the actual parameter. More specifically, a method specification $\{p\}\, m(\bar{u})\, \{q\}$ can be instantiated to a method call $x = y.m(\bar{e})$ by substituting **this** with the calling object y and the method parameters $\bar{u}$ with the actual arguments $\bar{e}$ in the preconditions and postconditions.

To validate the postcondition of a method body, which specifies the corresponding update of the associated history, we assume in the following method implementation rule a logical *update relation* $U_m(h, h')$ between the attributes of the updated history h and the 'old' history h'.

Rule 1 (Method implementation Rule)**.** Given the method definition $\{p\}\, m\, \{q\}$, the

body S of m, and the class invariant I, we have the rule

$$\frac{\{p \wedge I\}\, S\, \{r\} \quad U_m(h, h') \vdash r \rightarrow (q \wedge I)}{\{p\}\, m\, \{q\}}$$

This rule thus allows to abstract from the inductive definitions of the history attributes in the validation of the method body. The logical *update relation* $U_m(h, h')$ between the attributes of the updated history h and the 'old' history h', then can be established separately as a logical consequence of $h = Cons(m(\bar{x}, \textbf{result}), h')$ which directly describes the relation between the updated history and the old one in terms of their sequence structure. Here $\bar{x}$ are the actual parameters and **result** is the return value. The **result** variable here may either be *null* (indicating no return value) or contain a return value.

7.3 Case study

In this section, we introduce a banking example to illustrate the methodology discussed in Section 2. This example features a type hierarchy, which allows us to demonstrate our ideas effectively. It also presents some interesting challenges that occur in real-world programs, such as how to enforce protocol at the interface level where we have no access to the underlying state. We also consider some real-world scenarios, like how to extend functionality in existing programs. We implement the case study by defining the ADTs in Isabelle/HOL, and used ADTs in specification and KeY proof for Java programs. The artifact accompanying this paper [99] includes the full Isabelle theory of banking example and the proof files for the example discussed later.

In the banking example, we have two interfaces: the `Saving` interface and the `Payment` interface.

```
interface Saving {
   void deposit(int i);
   int getbalance();
}
```

Listing 7.1: The `Saving` interface.

```
interface Payment extends Saving {
   boolean query(int i);
   void withdraw();
}
```

Listing 7.2: The `Payment` interface.

The `Saving` interface, as shown in Listing 7.1, specifies methods for depositing an integer amount into the account and for retrieving the current balance. The `Payment` interface (Listing 7.2) defines two methods: the `query` method and the `withdraw` method. The `query` method is used to check whether there are sufficient funds in the account before each withdrawal.

The method signatures of the interface are designed to allow for the expression of the intended protocol, similar to how interfaces in Java Collection Framework are explained in informal Javadoc documentation [71]. The protocol for the `Payment` interface stipulates that the `withdraw` method can only be invoked when the return value of the `query` is true. This protocol is designed to protect the interface from

executing invalid withdrawal operations that could potentially lead to errors in the system (e.g. increasing the balance by calling `withdraw` multiple times).

We have three classes for our example, the `Account` class, the `Credit` class and the `Debit` class. The `Account` class (Listing 7.3) implements the methods defined in the `Saving` interface.

```
class Account implements Saving {
   int balance; // field
   void deposit(int i) { balance=balance + i; }
   int getbalance() { return balance; } }
```

Listing 7.3: The `Account` class.

The `Credit` class (Listing 7.4) permits withdrawals even if the balance is insufficient, similar to a real-world credit card. The `query` method, which is designed to check whether there are sufficient funds in the account before each `withdraw` method: in the case of the credit card, the caller always receives an affirmative response.

```
class Credit implements Payment {
   int request = −1; int balance; // fields
   void deposit(int i) { balance=balance + i; }
   int getbalance() { return balance; } }
   boolean query(int i) { request = i; return true; }
   void withdraw() { balance = balance − request; request = −1; }}
```

Listing 7.4: The `Credit` class.

In contrast, the `Debit` class (Listing 7.5) allows withdrawals only if the account has sufficient funds. This condition is determined by the return value of the `query` method. Specifically, it is true if and only if the balance is greater than or equal the argument.

```
class Debit extends Account implements Payment {
   int request = −1; // field
   boolean query(int i) {
        if (balance ≥ i) { request = i; return true; } else return false; }
   void withdraw() { balance = balance − request; request = −1; }}
```

Listing 7.5: The `Debit` class.

The hierarchical structure for our running example is depicted in Fig. 7.3.

7.3.1 History-based reasoning

In this subsection, we illustrate how we formalize ADTs for banking examples. We define data types and functions to logically model domain-specific knowledge of the Java program that we want to verify. Although these definitions cannot directly reference Java types, they can instead be defined using polymorphic type parameters. One defines data types and recursive functions using the **datatype** and **fun** commands.

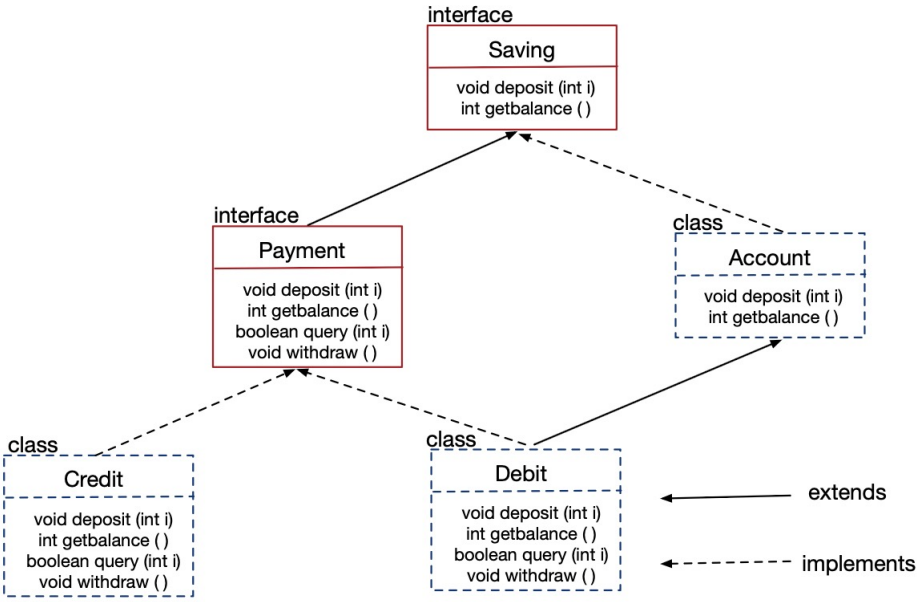

Figure 7.3: The type hierarchy of our running example.

The data types *events* contain the method name, the actual parameter values, and the output value (which is the final argument of the event) of a method call. The events are designed to be generic and do not contain information about the caller. This design choice makes events useful for a general history-based theory that is related to the caller. With regard to methods, there are methods like `deposit` that take an integer as a parameter but have no return value, whereas the `getbalance` method takes no input parameters but returns an integer. To distinguish between the input type and output type, we use a unit type **void** to represent the absence of a meaningful value. As discussed above, each subtype includes the events that are inherited from its supertype. For example, the definitions of the events for the interfaces in our running example are as follows:

$$\begin{aligned}\textbf{datatype}\ \mathit{saveEvent} &= \mathit{deposit}(\textbf{int}, \textbf{void}) \mid\ \mathit{balance}(\textbf{void}, \textbf{int})\\ \textbf{datatype}\ \mathit{payEvent} &= \mathit{deposit}(\textbf{int}, \textbf{void}) \mid\ \mathit{balance}(\textbf{void}, \textbf{int}) \mid\\ &\quad\ \mathit{query}(\textbf{int}, \textbf{bool}) \mid \mathit{withdraw}(\textbf{void}, \textbf{void})\end{aligned}$$

The concept of *history* is formally defined as an inductive data type of a sequence of events. Rather than employing temporal logic or formalizing history as an indexed set of events, we find that inductive data types offer a more convenient approach for defining attributes by induction and are easier to integrate with theorem provers in general. Thus, we introduce the history as a parameterized inductive datatype:

$$\textbf{datatype}\ \mathit{history}(\alpha) = \mathit{Empty} \mid\ \mathit{Cons}(\alpha, \mathit{history}(\alpha))$$

The type parameter α corresponds to the type of event occurring in the history, such as *saveEvent* and *payEvent*. For example, we can instantiate the parameter by the datatype *saveEvent* to obtain the histories for the `Saving` interface, which is represented as $\mathit{history}(\mathit{saveEvent})$. The history data type uses the constructors *Empty* and *Cons*, indicating that the history is either empty or composed of an event as its head and another history as its tail. When a new event is added, the new

event, along with its argument and return type, becomes the head of the history, while the old history turns to become the tail. It is worth mentioning that a history is generated in reverse order, which means that the last generated event appears at the start of the sequence.

The *amount* attribute, defined below, denotes the total amount of money in a given account. Intuitively, it serves as a snapshot representation of the interface's 'contents' at a particular instant. In the `Saving` interface, *amount* is defined inductively over the structure of the saving history, as shown below: a successful deposit increases the amount of money according to the value of the provided argument. We use *null* to represent a constant of type **void**.

$$
\begin{aligned}
&\textbf{fun}\ amount : history(saveEvent) \Rightarrow \textbf{int} \\
&\quad amount(Empty) = 0 \\
&\quad amount(Cons(deposit(i, null), h)) = amount(h) + i \\
&\quad amount(Cons(balance(null, i), h)) = amount(h)
\end{aligned}
$$

Attributes of history are treated similarly as "fields" in a class. To be specific, attributes defined by a supertype can be freely used and reinterpreted by its subtype. In our case, we redefine the *amount* attribute based on the payment history, taking the new methods `query` and `withdraw` into account.

$$
\begin{aligned}
&\textbf{fun}\ amount : history(payEvent) \Rightarrow \textbf{int} \\
&\quad amount(Empty) = 0 \\
&\quad amount(Cons(deposit(i, null), h)) = amount(h) + i \\
&\quad amount(Cons(balance(null, i), h)) = amount(h) \\
&\quad amount(Cons(query(i, b), h)) = amount(h) \\
&\quad amount(Cons(withdraw(null, null), h)) = \\
&\qquad\quad amount(h) - (ready(h)\ ?\ take(h) : 0)
\end{aligned}
$$

We define attributes *ready* and *take* as follows: given a history, *ready* checks whether the previous query event has returned **true**, and if so, *take* returns the parameter of `query` method.

$$
\begin{aligned}
&\textbf{fun}\ ready : history(payEvent) \Rightarrow \textbf{boolean} \\
&\quad ready(Empty) = \textbf{false} \\
&\quad ready(Cons(query(i, b), h)) = b \\
&\quad ready(Cons(withdraw(null, null), h)) = \textbf{false} \\
&\quad ready(Cons(e, h)) = ready(h) \\
&\textbf{fun}\ take : history(payEvent) \Rightarrow \textbf{int} \\
&\quad take(Empty) = -1 \\
&\quad take(Cons(query(i, b), h)) = (b\ ?\ i : -1) \\
&\quad take(Cons(withdraw(null, null), h)) = -1 \\
&\quad take(Cons(e, h)) = take(h)
\end{aligned}
$$

In the last clause, e is any event of *payEvent* not specified in the clauses above.

7.3.2 History-based specification

We now formulate method contracts of the methods of interface and class, making use of histories and its attributes. By overloading field access notation, we can treat the attributes of a history associated with **this** like how we would treat an unqualified field. For example, when considering the *amount* defined in the `Saving` interface, we can use syntactic sugar to simplify $amount(h)$ which h is of type $history(saveEvent)$ to just *amount* within the `Saving` interface. For external objects, we can explicitly indicate the object of which the corresponding history is taken in an attribute. Listing 7.6 shows a concrete example: suppose we would add a default method `transfer` to the `Payment` interface, that performs a withdrawal and immediately transfers the amount to the given `Saving` instance, then the postcondition illustrates that the amount of (the history of) **this** decreases, while the amount of (the history of) the receiver increases. Moreover, this example illustrates why hiding the concrete structure of a history from specifications is useful: while the default implementation does not record *transfer* as an event in the history of `Payment` and instead records the events which are used by the default implementation (in our case *withdraw*), non-default implementations do record a *transfer* event, thus have a different structure of the history.

```
// Transfer money from this Payment account to the given Saving account
{ready = true ∧ take = i}
default void transfer(int i, Saving s) { withdraw(); s.deposit(i); }
{(s ≠ this → amount = old(amount)−i ∧ s.amount = old(s.amount)+i) ∧
 (s = this → amount = old(amount))}
```

Listing 7.6: An example to specify the object of a history explicitly.

To avoid introducing a logical *freeze* variable, which would capture the history as it were in the pre-state, we use the notation **old** as a logical operation on terms to denote the attribute value evaluated in the pre-state of the method call, where **old** distributes over pure operations such as arithmetical functions. In the postcondition, *amount* in our example refers to the amount after the method call, while **old**(*amount*) represents the amount before the method call.

An interface specification includes the name of the interface being specified and the method signatures that the interface provides along with their respective precondition and postcondition. Listing 7.7 illustrates the use of the history attribute in the specification of the `Saving` interface.

```
Specification(Saving) =
({i ≥ 0} void deposit(int i) {amount = old(amount)+i},
 {true} int getbalance() {amount = old(amount) ∧ result = amount})
```

Listing 7.7: The specification of the `Saving` interface in terms of attribute *amount*.

The special variable **result** in the postcondition captures the return value of a method.

The specification of the `Payment` interface in terms of the attribute *amount* is shown in Listing 7.8.

Specification(`Payment`) =
($\{\texttt{i} \geq 0\}$ **void** `deposit`(**int** `i`) $\{amount = \textbf{old}(amount)+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{amount = \textbf{old}(amount) \wedge \textbf{result} = amount\}$,
$\{\texttt{i} \geq 0\}$ **boolean** `query`(**int** `i`) $\{amount = \textbf{old}(amount) \wedge$
$\textbf{result} = ready \wedge (ready \rightarrow take = \texttt{i})\}$,
$\{ready\}$ **void** `withdraw`() $\{amount = \textbf{old}(amount-take) \wedge \neg ready\}$)

Listing 7.8: The specification of the `Payment` interface in terms of attributes *amount*.

One should observe that the return value of the `query` method remains unspecified, thereby leaving design decisions open for subtypes and implementors. The intended meaning of the `query` method is to check whether money can be withdrawn from the account. Two implementations can be considered: a credit account (see Listing 7.10) and a debit account (see Listing 7.11). It is not possible to further specify the result in the `Payment` interface in a way that is compatible with both subtypes.

In addition to the type name and method specifications, a class specification may also contain the *class invariant*. The class invariant is an essential component of the class specification that should hold in the pre- and post-state of each method execution [97]. The method specifications of a class are described in terms of both fields and history attributes. The specification of the `Account` class is shown in Listing 7.9.

Specification(`Account`) =
(`balance` $= amount \wedge$ `balance` ≥ 0, // *class invariant*
$\{\texttt{i} \geq 0\}$ **void** `deposit`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance})+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge \textbf{result} = \texttt{balance}\}$)

Listing 7.9: The specification of the `Account` class.

The specification of the `Credit` class and the `Debit` class are present in Listing 7.10 and Listing 7.11, respectively.

Specification(`Credit`) =
(`balance` $= amount \wedge$ `request` $= take$ // *class invariant*
$\{\texttt{i}\geq 0\}$ **void** `deposit`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance})+\texttt{i}\}$,
$\{\textbf{true}\}$ **int** `getbalance`() $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge \textbf{result} = \texttt{balance}\}$,
$\{\texttt{i}\geq 0\}$ **boolean** `query`(**int** `i`) $\{\texttt{balance} = \textbf{old}(\texttt{balance}) \wedge$
$\textbf{result} = \textbf{true} \wedge \texttt{request} = \texttt{i}\}$,
$\{\texttt{request} \neq -1\}$ **void** `withdraw`() $\{\texttt{request} = -1 \wedge$
$\texttt{balance} = \textbf{old}(\texttt{balance}-\texttt{request})\}$)

Listing 7.10: The specification of the `Credit` class.

The value of **result** for the `query` method is explicitly specified in the classes `Debit` and `Credit` that implement the interface. Specifically, for the `Credit` class, **result** is unconditionally true. Conversely, for the `Debit` class, **result** is true when and

only when the *balance* $\geq i$. Note that these two conditions are not compatible: no debit object can be considered as a credit object.

```
Specification(Debit) =
(balance = amount ∧ request = take ∧ balance ≥ 0 ∧ balance ≥ request,
 {i ≥ 0} void deposit(i) {balance = old(balance)+i},
 {true} int getbalance() {balance = old(balance) ∧ result = balance},
 {i≥0} boolean query(i) {balance = old(balance) ∧ request = i ∧
                         result=(balance≥i)},
 {balance ≥ request ∧ request ≠ −1} void withdraw()
                    {request = −1 ∧ balance = old(balance−request)})
```

Listing 7.11: The specification of the `Debit` class.

The preconditions and postconditions effectively specify protocols for methods. For instance, suppose that the `withdraw` method should only be invoked following a valid query. In both the `withdraw` method in the `Debit` and `Credit` class, we impose a precondition constraint `request` $\neq -1$. When dealing with an interface method where we have no access to the underlying state, such as in the `Payment` interface (Listing 7.8), the protocol can describe using the attribute of history, specifically the *ready* attribute in this case. This allows us to capture the return value of a previous query, abstracting from the underlying implementation.

7.3.3 Behavioral subtyping

In this subsection, we discuss the refinement of interface-interface, interface-class, and class-class in the context of the banking example. Let us start with interface-interface refinement. The example we provide is the `deposit` method in the `Saving` interface (Listing 7.7) and its subtype, the `Payment` interface (Listing 7.8). First, we consider formulating the refinement relation. For user-defined refinement relation $R(h, g)$, h is of type $history(saveEvent)$, and g is of type $history(payEvent)$, we can formulate $R(h, g)$ according to the definition of *amount* for both the `Saving` and `Payment` interfaces, as provided below: every time the `withdraw` is called within the `Payment` interface and returns a *true* value, the *amount* attribute defined on the `Payment` history will decrease. To reflect this behavior, we introduce a new attribute, *withdrawamount*, to accumulate the total amount successfully withdrawn:

$$
\begin{aligned}
&\textbf{fun } withdrawamount : history(payEvent) \Rightarrow \textbf{int} \\
&\quad withdrawamount(Empty) = 0 \\
&\quad withdrawamount(Cons(withdraw(null, null), h)) = \\
&\qquad\qquad withdrawamount(h) + (ready(h)\ ?\ take(h) : 0) \\
&\quad withdrawamount(Cons(e, h)) = withdrawamount(h)
\end{aligned}
$$

again e is any *payEvent* not mentioned above.

Due to the introduction of the new attributes, we need to modify the method specification to capture the behavior of the interface. In this example, the method within the `Payment` interface requires modification for the introduction of attribute *withdrawamount*, as illustrated in Listing 7.12.

Spec(`Payment`) =
({`i`≥0} **void** `deposit`(`i`) {*amount*=**old**(*amount*)+`i` ∧
withdrawamount=**old**(*withdrawamount*)}, ...)

Listing 7.12: The `deposit` method specification for the `Payment` interface. The specifications for other methods within the `Payment` interface have also been revised.

We can then define the refinement relation as follows:

$$R(h, g) \stackrel{\text{def}}{\equiv} amount(h) = amount(g) + withdrawamount(g)$$

For $h : history(saveEvent)$ and $g : history(payEvent)$, we can formulate $R(h, g)$ by unfolding the attribute definition of h and g.

Now we apply Liskov and Wing's method rules to supertype and subtype in order to generate the verification conditions for behavioral subtyping. We first consider the implication between the preconditions of both types, where `i` serves as the actual parameter of the `deposit` method. The proof seems straightforward: $\texttt{i} \geq 0 \rightarrow \texttt{i} \geq 0$ is trivial. But what about postcondition $amount = \mathbf{old}(amount) + \texttt{i} \rightarrow amount = \mathbf{old}(amount) + \texttt{i}$? We cannot directly prove this due to the attribute *amount* of the `Saving` history is different from the attribute *amount* of the `Payment` history, as *amount* definition given in below. Even though attribute names may be identical, their definitions are specific to and can differ between different histories.

Instead, by de-sugaring and renaming the attribute, we explicitly get the *amount* of h, which is of type $history(saveEvent)$, and g, which is of type $history(payEvent)$. Since **old** distributes over pure operations, the designation of an attribute with the keyword **old** means to take the attribute of the old history, that is, the history prior to the method call. Thus, the expression $\mathbf{old}(amount(g))$ is equivalent to $amount(\mathbf{old}(g))$. We now have to show the following verification condition:

$$\begin{array}{c} amount(g) = amount(\mathbf{old}(g)) + \texttt{i} \\ \downarrow \\ amount(h) = amount(\mathbf{old}(h)) + \texttt{i} \end{array} \qquad \text{(VC1)}$$

However, the condition (VC1) remains unproven because there is a lack knowledge about the internal structure of the history and the definition of the attributes. To solve this issue, we use the *refinement relation* which allows us to relate the histories of supertype and subtype, and then we can further relate predicates about the supertype to those about the subtype and vice versa. By assuming $R(h, g)$, h represents the history of the supertype and g represents the history of the subtype, we can get the refinement relation for the history in the state where the method call started for free, that is $R(old(h), old(g))$. We then can simply prove the VC1.

$$\begin{array}{c} amount(g) = amount(\mathbf{old}(g)) + \texttt{i} \wedge \\ withdrawamount(g) = withdrawamount(\mathbf{old}(g)) \\ amount(h) = amount(g) + withdrawamount(g) \wedge \\ amount(\mathbf{old}(h)) = amount(\mathbf{old}(g)) + withdrawamount(\mathbf{old}(g)) \\ \downarrow \\ amount(h) = amount(\mathbf{old}(h)) + \texttt{i} \end{array}$$

The refinement for interface and class needs to take into account the class invariants. The specific example of interface-class refinement involves the precondition of the `withdraw` method of the `Payment` interface and its subclass, the `Debit` class. We can derive the following from their specifications (Listing 7.8 and Listing 7.11) based on the precondition rule:

$$\mathit{ready}(h) \rightarrow (\texttt{balance} \geq \texttt{request} \land \texttt{request} \neq -1) \tag{VC2}$$

In the attribute declaration $\mathit{ready}(h)$, the type of variable h is $\mathit{history}(\mathit{payEvent})$.

In this case, the `Debit` class fully inherits from the `Payment` interface. Thus, the refinement relation between them is as follows:

$$R(h, g) \stackrel{\text{def}}{\equiv} \mathit{ready}(h) = \mathit{ready}(g) \land \mathit{amount}(h) = \mathit{amount}(g)$$

The parameter h is an instance of the datatype $\mathit{history}(\mathit{payEvent})$, while g is an instance of the datatype $\mathit{history}(\mathit{debitEvent})$.

One can see that only the refinement relation is not sufficient to solve the VC2. The class invariant of the `Debit` class contains `balance` = *amount* and `request` = *take* which relates the attributes *amount* and *take* to the fields `balance` and `request`. By assuming the refinement relation, alongside the class invariants shown in Listing 7.11, we can prove the VC2: according to the definition of the attribute, if *ready* returns true, then *take* $\neq -1$.

$$\begin{gathered}
\mathit{ready}(h) \land \\
\texttt{balance} = \mathit{amount}(g) \ \land \ \texttt{request} = \mathit{take}(g) \\
\texttt{balance} \geq 0 \land \texttt{balance} \geq \texttt{request} \\
\mathit{ready}(h) = \mathit{ready}(g) \land \mathit{amount}(h) = \mathit{amount}(g) \\
\downarrow \\
\texttt{balance} \geq \texttt{request} \land \texttt{request} \neq -1
\end{gathered}$$

Now we turn to focus on class-class refinement. The relation between two classes needs to consider the use of class invariants in both the superclass and the subclass. If invariants can be employed in supertypes for reasoning, subtypes must also obey these invariants. To be specific, the complete invariant of a subclass specification is formed as a conjunction of both the invariant in the supertype and the unique invariant specific to the subtype itself. The class invariant, which typically connects the fields and attributes of history, can leverage refinement relation to prove the verification conditions. To be specific, given a refinement relation between a subclass and a superclass, if the class invariants for the subclass hold, it would logically follow that the class invariants for the superclass also hold. In the banking example, an invariant property of the `Account` class is that its balance is always greater or equal to zero. The `Credit` class allows one to withdraw money even if the balance is negative, so the `Credit` class cannot be a subclass of the `Account` class. Conversely, the `Debit` class inherits invariants from the specification of the `Account` class and also has its own invariants, as outlined in Listing 7.11.

We now delve deeper into the design problem of method placement within the type hierarchy, in order to emphasize the importance of adherence to the behavioral

subtyping rule. In most cases, developers have the flexibility to decide at what level a method should be placed. For instance, the `Payment` interface introduces two new methods: `query` and `withdraw`. One potential approach is to define the `withdraw` method within the `Payment` interface and introduce the `query` method exclusively as an addition to the `Debit` implementation. However, this approach clashes with behavioral subtyping, which requires the implication *ready* $\rightarrow$ `balance` $\geq$ `request` $\wedge$ `request` $\neq -1$ to hold, but *ready* cannot be given a sensible meaning at the level of the `Payment` interface without the `query` method. Thus, in our running example, we define the `query` method in the supertype. This allows the subclasses to implement the method, thereby ensuring compliance with behavioral subtyping.

From a developers' perspective, another important consideration is how to extend the functionality of a program. For example, with the increasing need for security, the system may require an additional step: entering the pin code to verify whether the person withdrawing money is legitimate. Instead of modifying the existing `Payment` interface, a new sub-interface, named `Security`, can be defined. This sub-interface would include a new attribute, *pin*, designed to capture the entered pin code. Thus, in the `query` method, the subtype need to add a new precondition to verify the correctness of the pin code. Benefiting from our approach, we do not need to reconstruct abstract functions for each state, but only formulate refinement relations between the new type and its supertypes and subtypes to ensure behavioral subtyping.

7.3.4 Example of method call and implementation

We exemplify the method call rule through a client-side example: the verification of the `mybalance` method, as shown in Listing 7.13.

```
class ClientExample{
  {true}
  int mybalance(Saving s){
  int i = s.getbalance(); return i;}
  {result = s.amount ∧ s.amount = old(s.amount)}

  {j≥0}
  int myAccount(Debit d, int j){
     d.deposit(j); int r = mybalance(a); return r;}
  {result = d.amount ∧ d.amount = old(d.amount)+j}
}
```

Listing 7.13: Client code that illustrate the method call rule.

At the beginning of the method, its precondition is assumed. To verify the call to `getbalance` method, we rely on the specification of the callee, in this case, the `Saving` interface (with specifications provided in Listing 7.7). By substituting the callee for the implicit receiver **this** in the specification, we can assume the postcondition.

$$\{\textbf{true}\}\,\textbf{result} = \mathit{this}.\texttt{mybalance}(\texttt{Saving s})\,\{\textbf{result} = \texttt{s}.\mathit{amount}\}$$

The technique for method call verification relies on the method specification, which uses the attributes and fields to describe the expected behavior of the method. The verification of clients based on those method specifications leaves histories uninterpreted, thereby eliminating the need to prove the correctness of the method's implementation each time the method is called. For `myAccount` method, the verification of the method call is independent of the implementation of the argument. This means that only the specification given in the `Account` is accessible. We can prove the postcondition of the `myAccount` method by referring to the history of the `Account` class and its corresponding redefinition of the attribute *amount*.

A specific example of method implementation we can consider the `deposit` method in the `Account` class, as shown in Listing 7.14.

```
class Account implements Saving {
  balance ≥ 0 ∧ balance = amount, //invariant
  {i≥0}
  void deposit(i){balance=balance+i;}
  {balance=old(balance)+i}
}
```

Listing 7.14: The `deposit` implementation in the `Account` class.

One verification condition involves reasoning about the class invariant `balance` = *amount* should hold before and after the method `deposit` call. We verify the `deposit` method is correct with respect to the contract. How can we show the attribute *amount* also changed without knowing the internal structure of the history and the attribute definition? Within the implementing class, the history is defined by the field **this**, which is updated during the method call with a newly created history that involves the new event: the `deposit` event. Attributes are used to map the history to a particular value, with the update of the history, the value of the attribute also changed.

For the example in Listing 7.14, we can establish the update relation in terms of the attributes as below. This can be verified by unfolding the *amount* definition.

$$amount(Cons(deposit(i, null), h')) = amount(h') + \texttt{i}$$

We provide a manual translation as $amount = \mathbf{old}(amount) + \texttt{i}$, so it can be used in the verification condition. One can prove the class invariant by showing that both the class field and attribute increase accordingly.

7.4 Summary

Programming to interfaces, a key principle in object-oriented programming, is fundamental to numerous popular frameworks that offer hierarchies of interfaces and classes. These interfaces abstract from state and implementation, enhancing modularity and maintainability in software systems. Behavioral subtyping complements the practice of programming to interfaces by ensuring that subtypes not only match

the method signatures defined by their supertypes but also adhere to the intended behaviors.

The main contribution of this chapter is to develop a general history-based refinement approach for verifying behavioral subtyping, allowing consistent program specification at different abstraction levels. Our methodology enables us to reason about interfaces, generating interfaces-based behavioral subtyping rules that are notably absent in the majority of studies. We showed how our refinement approach can be effectively employed to rationalize various kinds of refinements in terms of projection relation: from interface to interface, interface to class and class to class.

As logical properties, attributes serve several purposes. They can map a single history to a value, represent the relationship for different histories like the refinement relation, and reflect different states of the same history like the update relation. Moreover, we applied our approach to verifying method calls as well as classes that implement interfaces. Our running example served as a practical guide, showcasing the value of our approach in realistic scenarios.

Our history-based refinement theory is suitable for all three scenarios: method overriding, method inheritance, and methods explicitly defined within the subtype itself [27]. When a method is inherited, the subclass simply inherits the same precondition and postcondition as the method in its supertype. For method overriding, a subtype that overrides its supertype's method must adhere to the behavioral subtyping rule. In the case of method overloading, the method in the subtype may have a different signature compared to the methods of the same name in its supertype. We can interpret this distinctive scenario as the subtype defining a new method, which is independent of the supertype's method.

This work simplifies the workflow by clearly distinguishing between the role of the designer, who deals with attributes, and the role of the verifier, who handles verification conditions. Designers can not only define attributes but also provide the grounding to confirm the realizability of the theory. The verifier's assumptions are based on refinement relation provided by the designer, which can used to prove the verification conditions generated for behavioral subtyping.

Chapter 8

Conclusion

Throughout the main body of the thesis, we implemented a series of studies on exploring ways to apply formal methods systematically for the verification of complex object-oriented libraries such as the Java Collection Framework. We start with specifying and verifying methods in the `java.util.LinkedList` class, but we encounter challenges with methods that take an interface type as a parameter. To address this, we proposed to use histories as method calls and returns to completely determine the concrete state of any implementation and thus can be seen as a way to reason about the interface. The executable history-based (EHB) approach, designed to facilitate history-based reasoning and creating reusable specifications for Java programs, embeds histories and attributes directly as Java objects. This approach could be seamlessly integrated in the KeY theorem prover and avoids the need to change the KeY system itself. However, the EHB approach still has its limitations, particularly when it comes to reasoning about the heap and properties of user-defined attributes, which can require a lot of work due to alias analysis and dynamic footprints. To mitigate this, we introduce the logical history-based (LHB) approach, which models histories as an external abstract data type with functions. This opened up new possibilities for modeling complex behavior in object-oriented programs. Building on the LHB approach, we have developed a history-based refinement theory for reasoning about hierarchy in object-oriented programs. To systematically conclude the thesis, in this final chapter, we first summarize the contributions we have made to addressing the key challenges of formal verification in object-oriented libraries, as formulated in the introductory chapter. Finally, we provide a list of possible directions for future work.

8.1 Summary of contributions

Ensuring that software libraries operate without errors and function as intended has always been a central concern in the field of computer science. This is especially critical given that these libraries serve as the foundation for countless applications and are used by billions of devices worldwide. Formal verification offers a rigorous, mathematically sound way to confirm the accuracy of the software, grounded on clearly defined behavior criteria expressed in formal logic. While formal verification

provides robust assurance of software's correctness, unlike testing, it often demands considerable time and resources to define specifications and develop proofs. This thesis has extended the application of the KeY theorem prover to achieve systematic verification of object-oriented libraries of the popular programming language Java. This work may interest non-specialists, as it shows what features of a specification and verification system we need in order to reason about real-world programs. It is also beneficial for beginning users of KeY and Isabelle/HOL, as we introduce and informally explain several key concepts in Chapter 2. We also provide the artifacts and video materials for each chapter to help in reproducing the proofs underlying the results. These materials also help the expert user and the developer of KeY as a 'benchmark' for specification and (automatic) verification techniques. Below, we give a short summary of the contribution for each chapter.

In Chapter 3, we outline the methodology for analyzing an existing Java program to gain a deeper understanding of its behavior. It emphasizes the importance of precise specifications, using the JML for clarity. To validate the program's behavior against these specifications, the chapter advocates for a formal approach supported by the KeY tool, which uniquely allows for comprehensive reasoning on Java programs. This tutorial emphasizes the critical importance of ensuring program correctness in software libraries, particularly in Java's standard library, due to their widespread use and potential for systemic impact.

In Chapter 4, we explore the reasoning about the correctness of Java interfaces, with a particular application to Java's `Collection` interface. We introduce the concept of a *history* as a sequence of method calls and returns as a general methodology for specifying interfaces and verifying clients and implementations of interfaces. This helps us to develop a novel "proving to interface" methodology.

As a proof-of-concept, using the KeY theorem prover, in Chapter 5, the so-called EHB approach has been applied to the core methods of Java's `Collection` interface. The EHB approach is to embed histories and attributes in the KeY theorem prover by encoding them as Java objects on the heap, thereby avoiding the need to change the KeY system itself. We show our approach is sufficient for reasoning about interfaces from the client's perspective, as well as about classes that implement interfaces. However, the EHB approach uses pure methods that rely on the heap, giving rise to additional proof obligations every time these pure methods are used in JML specifications. Moreover, reasoning about the properties of user-defined functions is complex. For instance, the proofs about *multiset* attribute modeled as a pure method take 72 minutes of work.

We then proposed the LHB approach. The LHB approach encodes histories as built-in ADTs with special proof rules, to avoid modeling histories as Java objects. We discuss integrating ADTs in the KeY theorem prover by a new approach to model data types using Isabelle/HOL as an interactive back-end and representing Isabelle theorems as user-defined taclets in KeY. In Chapter 6, we detail on how we designed our specification of the `Collection` interface, and describe in more detail the steps needed to verify several complex example clients. In this chapter, we have seen an application of our technique to the case of history-based reasoning. The main contribution of this chapter 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. Our LHB approach is not only useful for reasoning about the Java Collection Framework, but 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.

The work using the LHB approach 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 uses collections. The binary method takes more than 100 minutes to verify: it is hard to imagine that it can be done with the EHB approach. Moreover, we significantly simplified reasoning about the properties of user-defined functions themselves. We can fully automate verification in Isabelle/HOL with user-defined attributes modeled as a function. 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, was problematic or even impossible.

In Chapter 7, we introduce a new history-based proof-theory that allows us to formally verify that inherited methods are correct with respect to refinements of overridden methods. Benefiting from the LHB approach, we formulate behavioral subtyping rules that can be employed to axiomatize various kinds of refinements in terms of a projection relation: from interface to interface, interface to class, and class to class. To bring these concepts to real code, we describe a simple running example that captures the key hierarchy structure and some interesting challenges in object-oriented programs, e.g. specifying interface protocols. Through this example, we demonstrate the practical applicability of our history-based refinement approach.

8.2 Future work

The research presented in this thesis achieved some interesting results and opened up several potential research directions that we leave for future work. In this section, we briefly discuss future directions related to our main topic.

In Chapter 3 and Chapter 4, we discuss the specification and verification of part of the classes and interfaces provided by the Java Collection framework. To achieve the ultimate goal of complete formal verification of Java's Collection Framework still requires a lot of effort. For example, with our novel approach, one can continue our specification and verification work on `LinkedList`, which we introduced in Chapter 3, to include methods like `retainAll` and `removeAll` that have not yet been verified. Furthermore, the verification of other classes in the Java collection framework, such as `ArryList`, remains open. While Chapter 4 focuses on the `Collection` interface, there are several other interfaces, such as, `Map`, `List`, `Set` and `ListIterator`, that warrant attention in future work.

In Chapter 6, we introduced a technique for integrating ADTs into the KeY theorem prover. We outline how data types, functions, and lemmas can be imported into domain-specific theorem prover KeY from the general-purpose theorem prove Isabelle/HOL. It is noteworthy that the translation from Isabelle/HOL to KeY is implemented manually. Our approach leverages that Isabelle/HOL guarantees the consistency of introducing user-defined ADTs and functions. We manually translate these ADTs and functions as axioms into KeY using taclet rules, and ensure that these rules can be accepted and used by KeY. This process requires the verifier to be very familiar with KeY, Isablle/HOL, JML, taclet rules, etc. From the practical perspective, an automatic tool that imports Isabelle/HOL theories into KeY based on our work could be implemented. This would further reduce manual intervention and enable full automation of the verification process.

In Chapter 7, we proposed a history-based refinement theory to verify the hierarchy structure in widely used object-oriented programs. For instance, within the Java Collection Framework, the `Collection` interface serves as a foundational component within the framework, representing a group of objects and providing a blueprint for various concrete implementations, including `List`, `Set`, and `Queue`. The more complex hierarchy structure in the `Collection` interface can be found in the class `Linkedlist` that inherits from `AbstractSequqntialList` which inherits from `AbstractList` and then inherits from `AbstractCollection` and implements the `List` interface. Benefiting from our history-based refinement theory, we can follow the hierarchy structure to systematically analyze and validate the behavioral subtyping relations between each class and interface. Besides, the refinement theory between `Iterator` and `ListIterator` is also an interesting direction, as an iterator requires a notion of ownership since its behavior depends on the history of other objects. It remains future work to apply this theory to verifying real software. Such an effort could be used to demonstrate how formal methods improve the reliability and accuracy of popular object-oriented libraries.

From a long-term perspective, it is worthwhile to consider future work related to verified code revisions and proof reuse. In Chapter 3, we discussed fixing the `LinkedList` class by explicitly bounding its maximum size to `Integer.MAX_VALUE` elements, but other solutions are possible. Rather than using integers indices for elements, one could change to an index of type `long` or `BigInteger`. Such a code revision is however incompatible with the general `Collection` and `List` interfaces (whose method signatures mandate the use of integer indices), thereby breaking all existing client code that uses `LinkedList`. Clearly, this is not an option in a widely used language like Java or any language that aims to be backward compatible. It raises the challenge: can we find code revisions that are compatible with existing interfaces and their clients? We can take this challenge even further: can we use our workflow to find such compatible code revisions, and are they also amenable to formal verification? For code reuse, many case studies in mechanic verification [1, 23, 73] indicate that the main bottleneck today is not verification, but specification. For example, the `LinkedList` case study comprised approximately 18-21 person months in total. But once the specifications were in place, after many iterations of the workflow, producing the actual final proofs took only 1-2 weeks!

Specifications typically need to be developed incrementally during the proof effort, but there is little support for such an incremental approach in KeY: minor specification changes, like adding a conjunct to a class invariant, often require to redo nearly the whole proof, causing an explosion in the amount of effort needed. This vulnerability to change arises partly from proof rules that have a very fine granularity: proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied, resulting in fragile specifications. In the current version of KeY, proofs consist of actual rule applications (rather than higher-level macro/strategy applications), and proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied to. This results in a fragile proof format, where small changes to the specifications or source code (such as code refactoring) break the proof. Moreover, the rule set used may change in different versions of KeY, limiting backward compatibility for proofs made in a different KeY version. To improve the reusability of proofs, one can develop a versioning system for proof rules that use very fine-grained proof representations. The automatic generation of high-level proof scripts by monitoring the interactions between the proof engineer and the prover is also future work. Dealing with modifications of the underlying proof system of the theorem prover while supporting resuming existing, possibly partial proofs (through the versioning system and proof gaps) is also an interesting future direction.

Bibliography

[1] de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: 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 (2015) 273–289

[2] de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reasoning **62** (2019) 93–126

[3] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7** (2005) 212–232

[4] Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical study of usage and performance of Java collections. In: 8th Conference on Performance Engineering, ACM (2017) 389–400

[5] Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M.C.J.D., de Gouw, S.: 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 (2020) 217–234

[6] Krasner, H.: The cost of poor software quality in the US: A 2022 report. Proc. Consortium Inf. Softw. QualityTM (CISQTM) (2022)

[7] Rooney, J.J., Heuvel, L.N.V.: Root cause analysis for beginners. Quality progress **37** (2004) 45–56

[8] Mili, A., Tchier, F.: Software testing: Concepts and operations. John Wiley & Sons (2015)

[9] Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: 9th Conference on Verified Software, Theories, Tools, and Experiments (VSTTE). Volume 10712 of LNCS., Springer (2017) 35–48

[10] Huisman, M.: Verification of Java's AbstractCollection class: A case study. In: 6th Conference on Mathematics of Program Construction. Volume 2386 of LNCS., Springer (2002) 175–194

[11] Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: 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 (2018) 53–70

[12] de Boer, F.S., de Gouw, S., Vinju, J.J.: Prototyping a tool environment for run-time assertion checking in JML with communication histories. In: Formal Techniques for Java-Like Programs (FTfJP), ACM (2010) 6:1–6:7

[13] Kandziora, J., Huisman, M., Bockisch, C., Zaharieva-Stojanovski, M.: Run-time assertion checking of JML annotations in multithreaded applications with e-OpenJML. In: Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs. (2015) 1–6

[14] Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM (2007) 569–588

[15] Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal **15** (2007) 7–25

[16] Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java pro-

grams (tool paper). In: Software Engineering and Formal Methods (SEFM), IEEE Computer Society (2009) 33–37

[17] Azzopardi, S., Colombo, C., Pace, G.J.: CLARVA: Model-based residual verification of Java programs. In: Model-Driven Engineering and Software Development (MODELSWARD), SciTePress (2020) 352–359

[18] Welsch, Y., Poetzsch-Heffter, A.: A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. Science of Computer Programming **92** (2014) 129–161

[19] Visser Willem, Păsăreanu Corina S., K.S.: Test input generation with java pathfinder. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. (2004) 97–107

[20] Havelund K, P.T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer **2** (2000) 366–381

[21] Huisman, M., Jacobs, B., van den Berg, J.: A case study in class library verification: Java's Vector class. Int. J. Softw. Tools Technol. Transf. **3** (2001) 332–352

[22] Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java language. In: Programming Languages and Systems (PLS). Volume 3444 of LNCS., Springer (2005) 423–438

[23] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification – The KeY Book. Volume 10001 of LNCS. Springer (2016)

[24] Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java Modeling Language. In: [23]. Springer (2016) 193–241

[25] Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: 17th European Symposium on Programming. Volume 4960 of LNCS., Springer (2008) 307–321

[26] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)

[27] Meyer, B.: Object-oriented software construction. Volume 2. Prentice hall Englewood Cliffs (1997)

[28] Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994) 1811–1841

[29] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification - The KeY Book - From Theory to Practice. Volume 10001 of Lecture Notes in Computer Science. Springer (2016)

[30] Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Schlager, S.: Taclets: A new paradigm for constructing interactive theorem provers. RACSAM **98** (2004) 17–53

[31] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual. Unpublished manuscript, revision 2344 (2013)

[32] Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)

[33] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283 of LNCS. Springer (2002)

[34] Wenzel, M., Berghofer, S.: The Isabelle system manual (2014)

[35] Bauer, G., Nipkow, T., Oheimb, D.v., Paulson, L.C., Rasmussen, T.M., Tabacznyj, C., Wenzel, M.: The supplemental Isabelle/HOL library (2002)

[36] Biendarra, J., Blanchette, J.C., Desharnais, M., Panny, L., Popescu, A., Traytel, D.: Defining (co)datatypes and primitively (co)recursive functions in Isabelle/HOL (2016) Available at: https://isabelle.in.tum.de/doc/datatypes.pdf.

[37] Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: 27th Symposium on Logic in Computer Science (LICS), IEEE (2012) 596–605

[38] Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer (2020) 217–234

[39] Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: A Tutorial on Verifying LinkedList using KeY: Proof Files (2020) Available at: https://doi.org/10.5281/zenodo.3613711.

[40] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Video Material (2020) Available at: https://doi.org/10.6084/m9.figshare.c.4826589.v2.

[41] Klint, P., Van Der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE (2009) 168–177

[42] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 1 (2020) Available at: https://doi.org/10.6084/m9.figshare.11662824.

[43] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 2 (2020) Available at: https://doi.org/10.6084/m9.figshare.11673987.

[44] Bloch, J., Gafter, N.: Collection (Java Platform SE 8) (2020) Available at: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

[45] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3a (2020) Available at: https://doi.org/10.6084/m9.figshare.11688816.

[46] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3b (2020) Available at: https://doi.org/10.6084/m9.figshare.11688858.

[47] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3c (2020) Available at: https://doi.org/10.6084/m9.figshare.11688870.

[48] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3d (2020) Available at: https://doi.org/10.6084/m9.figshare.11688984.

[49] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3e (2020) Available at: https://doi.org/10.6084/m9.figshare.11688891.

[50] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4a (2020) Available at: https://doi.org/10.6084/m9.figshare.11699178.

[51] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4b (2020) Available at: https://doi.org/10.6084/m9.figshare.11699253.

[52] Bruce, K.B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S.F., Trifonov, V., Leavens, G.T., Pierce, B.C.: On binary methods. Theory Pract. Object Syst. **1** (1995) 221–242

[53] Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: History-based specification and verification of Java collections in KeY. In: 16th International Conference on Integrated Formal Methods, Springer (2020) 199–217

[54] Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. PhD thesis, Karlsruhe Institute of Technology (2011)

[55] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th Symposium on Logic in Computer Science (LICS), IEEE (2002) 55–74

[56] Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM (2008) 213–226

[57] Banerjee, A., Naumann, D.A., Nikouei, M.: A logical analysis of framing for specifications with pure method calls. ACM Trans. Program. Lang. Syst. **40** (2018)

[58] Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Fundamental Approaches to Software Engineering (FASE). Volume 4422 of LNCS., Springer (2007) 336–351

[59] Nipkow, T.: Embedding programming languages in theorem provers. In: International Conference on Automated Deduction, Springer (1999) 398–398

[60] von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation:

Practice and Experience **13** (2001) 1173–1214

[61] Jacobs, B., Van Den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about Java classes: preliminary report. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. (1998) 329–340

[62] Huisman, M.: Reasoning about Java programs in higher order logic using PVS and Isabelle. PhD thesis, University of Nijmegen (2001)

[63] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 348–370

[64] Filliâtre, J.C., Paskevich, A.: Why3: where programs meet provers. In: 22nd European Symposium on Programming. Volume 7792 of LNCS., Springer (2013) 125–128

[65] Liskov, B., Zilles, S.: Programming with abstract data types. ACM SIGPLAN Notices **9** (1974) 50–59

[66] Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. Springer (2012)

[67] Leavens, G.T., Cheon, Y.: Design by contract with JML (2006) Available at: http://www.cs.utep.edu/cheon/cs3331/data/jmldbc.pdf.

[68] Darvas, A., Müller, P.: Faithful mapping of model classes to mathematical structures. In: 2007 Conference on Specification and Verification of Component-Based Systems (SAVCBS), ACM (2007) 31–38

[69] Giese, M.: Taclets and the KeY prover. Electronic Notes in Theoretical Computer Science **103** (2004) 67–79

[70] Habermalz, E.: Ein dynamisches automatisierbares interaktives Kalkül für schematische theorie spezifische Regeln. PhD thesis, University of Karlsruhe (2000)

[71] Bloch, J., Gafter, N.: Collection (Java Platform SE 7) (2010) Available at: https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html.

[72] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their Application to History-based Reasoning about Collection: Proof Files. Zenodo (2022) Available at: https://doi.org/10.5281/zenodo.7079126.

[73] de Gouw, S., de Boer, F.S., Rot, J.: Proof pearl: The key to correct and stable sorting. J. Autom. Reason. **53** (2014) 129–139

[74] de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of jdk's identity hash map implementation. In ter Beek, M.H., 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 (2022) 45–62

[75] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their application to history-based reasoning. In Huisman, M., Păsăreanu, C., Zhan, N., eds.: Formal Methods, Cham, Springer International Publishing (2021) 255–272

[76] Bian, J., Hiep, H.A.: Integrating ADTs in KeY and their Application to History-based Reasoning: Video Material. FigShare (2021) Available at: https://doi.org/10.6084/m9.figshare.c.5413263.

[77] AbdelGawad, M.A.: Why nominal-typing matters in oop. arXiv preprint arXiv:1606.03809 (2016)

[78] Leavens, G.T.: Introduction to the literature on object-oriented design, programming, and languages. ACM SIGPLAN OOPS Messenger **2** (1991) 40–53

[79] America, P.: Inheritance and subtyping in a parallel object-oriented language. In: ECOOP'87 European Conference on Object-Oriented Programming: Paris, France, June 15–17, 1987 Proceedings 1, Springer (1987) 234–242

[80] America, P.: Designing an object-oriented programming language with behavioural subtyping. In: Foundations of Object-Oriented Languages: REX School/Workshop Noordwijkerhout, The Netherlands, May 28–June 1, 1990 Proceedings, Springer (1991) 60–90

[81] Bruce, K.B., Wegner, P.: An algebraic model of subtypes in object-oriented languages (draft). ACM Sigplan Notices **21** (1986) 163–172

[82] Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proceedings of IEEE 18th International Conference on Software Engineering, IEEE (1996) 258–267

[83] Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006. Proceedings 8, Springer (2006) 2–34

[84] Leavens, G.T., Naumann, D.A.: Behavioral subtyping, specification inheritance, and modular reasoning. ACM Transactions on Programming Languages and Systems (TOPLAS) **37** (2015) 1–88

[85] Leavens, G.T., Weihl, W.E.: Reasoning about object-oriented programs that use subtypes. In: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications. (1990) 212–223

[86] Leavens, G.T., Weihl, W.E.: Specification and verification of object-oriented programs using supertype abstraction. Acta Informatica **32** (1995) 705–778

[87] Müller, P.: Modular specification and verification of object-oriented programs. Springer (2002)

[88] Parkinson, M.J.: Local reasoning for Java. Technical report, University of Cambridge, Computer Laboratory (2005)

[89] Pierik, C.: Validation techniques for object-oriented proof outlines. PhD thesis, Utrecht University (2006)

[90] Back, R.J., Wright, J.: Refinement calculus: a systematic introduction. Springer Science & Business Media (2012)

[91] Back, R.: On correct refinement of programs. Journal of Computer and System Sciences **23** (1981) 49–68

[92] Morgan, C.: Programming from specifications. Prentice-Hall, Inc. (1990)

[93] Goldsack, S., Kent, S.: A type-theoretic basis for an object-oriented refinement calculus. In: Formal methods and object technology, Springer (1996) 317–335

[94] Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62** (2006) 253–286

[95] Reus, B.: Modular semantics and logics of classes. In: CSL. Volume 3., Springer (2003) 456–469

[96] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: 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, Springer (2021) 255–272

[97] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., et al.: JML reference manual (2008)

[98] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not. **21** (1986) 38–45

[99] Bian, J., Hiep, H.A., de Boer, F.S.: History-Based Reasoning about Behavioral Subtyping:Proof fiels. Zenodo (2024) Available at: https://doi.org/10.5281/zenodo.10998227.

English summary

Software plays a crucial role in our interaction with the real world and is also embedded within some of the most critical systems. Ensuring that software is free of bugs and works as intended presents a significant challenge in software development. Object-oriented programming is a programming paradigm widely used in the development of many software systems. Applying formal specifications to verify the correctness of object-oriented programs can be very beneficial, as even a minor error within widely used programs can lead to significant issues, such as system outages and failures. This thesis demonstrates the use of formal methods for systematically verifying state-of-art, real object-oriented programs.

In Chapter 3, we focus on the formal verification of object-oriented classes. We discuss the specification and verification of a corrected version of the linked list implementation from the Java Collection framework, which originally contained an overflow bug. Our formal specification aimed at two goals: to establish the absence of the overflow bug, and to capture the essential behavior of the methods with respect to the structural properties of the linked list. We successfully demonstrated, using the KeY theorem prover, that the fixed version of the core methods of the linked list implementation in Java is formally correct.

The work on verifying linked list successfully verified most methods but excluded some method implementations that contain an interface type as a parameter. Interfaces abstract away from state and other internal implementation details, facilitating modular program development. However, tool-supported programming logics and specification languages are predominantly state-based, which as such cannot be directly used for interfaces. In Chapter 4, we introduce a novel specification method using histories, recording method calls and returns, on an interface. The abstractions over histories, called attributes, are used to describe all possible behaviors of objects regardless of its implementation. Interface specifications can then be written in the state-based specification language JML by referring to histories and its attributes to describe the intended behavior of implementations.

To demonstrate the feasibility of the history-based reasoning approach, we have specified part of the core methods of Java's Collection interfaces in Chapter 5, using the *executable* history-based approach (the EHB approach). This approach uses an encoding of histories as Java objects on the heap. That encoding, however, made use of pure methods in its specification. 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.

To enhance the history-based approach, Chapter 6 discusses 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. We model histories as elements of an ADT, separate from the sorts used by Java in the EHB approach (Chapter 5). Histories then can not be touched by Java programs under verification themselves. We refer to this as the *logical* history-based approach (the LHB approach). We showed how ADTs externally defined in Isabelle/HOL can be used in JML specifications and KeY proofs for Java programs. As a more advanced case study, we provide a specification of the addAll method and verify the correctness properties of its clients. Furthermore, we reasoned about advanced, realistic use cases involving multiple instances of the same interface.

Chapter 7 focuses on the use of hierarchy in object-oriented programs. Hierarchy naturally follows from behavioral subtyping, in which the subtypes must not only match the method signatures defined by their supertypes but also adhere to the intended behaviors. We develop a general history-based refinement theory that used to verify the subtype relation. We associate each interface and class with a history that represents the sequence of method calls performed on the object since its creation. The subtype relation is described in terms of projection relation, which means the relationship between subtypes and supertypes hinges on the projection between histories, with each type having its own history. Through the use of a running example, we demonstrate the practical applicability of our approach.

Programming to interfaces is one of the core principles in object-oriented programming and is central to most of the standard libraries, which provide a hierarchy of interfaces and classes that represent object containers. We proposed techniques that are capable of specifying and verifying class, interface, and hierarchy structure. Taking the Java collection framework as a case study, we show the usefulness of our techniques. Thus, this thesis provides important novel contributions, insights, and findings for the research community and future applications in the field of software verification.

Nederlandse samenvatting

Software speelt een essentiële rol in hoe we met de echte wereld interacteren en is ingebed in enkele van de meest cruciale systemen die we dagelijks gebruiken. Het verzekeren dat deze software foutloos is en naar behoren functioneert, vormt een aanzienlijke uitdaging in de wereld van softwareontwikkeling. Het programmeringsparadigma van objectgeoriënteerd programmeren, dat breed wordt toegepast in de ontwikkeling van talrijke software systemen, staat centraal in deze discussie. Het inzetten van formele specificaties om de correctheid van objectgeoriënteerde programma's te verifiëren, biedt aanzienlijke voordelen. Zelfs geringe fouten in veelgebruikte programma's kunnen immers leiden tot grote problemen, zoals uitval en storingen van systemen. Dit proefschrift illustreert hoe formele methoden kunnen worden aangewend voor de systematische verificatie van geavanceerde, daadwerkelijk gebruikte objectgeoriënteerde programma's.

In het derde hoofdstuk richten we ons op de formele verificatie van objectgeoriënteerde klassen. We behandelen de specificatie en verificatie van een verbeterde versie van de implementatie van de gelinkte lijst uit het Java Collection Framework, die oorspronkelijk een bug gerelateerd aan geheugenoverloop bevatte. Onze formele specificatie streefde naar twee doelstellingen: ten eerste het aantonen van de afwezigheid van deze overflow bug en ten tweede het nauwkeurig beschrijven van het essentiële gedrag van de methoden, in relatie tot de structurele eigenschappen van de gelinkte lijst. We hebben met succes aangetoond, door het gebruik van de KeY theorem prover, dat de gecorrigeerde versie van de kernmethoden van de gelinkte lijst implementatie in Java formeel correct is.

In ons onderzoek naar gelinkte lijsten hebben we de meeste methoden geverifieerd, maar sommigen, die interfaces gebruiken, overgeslagen. Interfaces abstraheren van toestand en implementatiedetails, wat modulaire softwareontwikkeling bevordert. Echter, veel programmeerlogica en specificatietalen zijn toestandgebaseerd en niet direct geschikt voor interfaces. In Hoofdstuk 4 introduceren we een nieuwe methode die 'geschiedenissen' gebruikt voor het vastleggen van interface-interacties. Deze methode, via 'attributen', beschrijft objectgedrag onafhankelijk van implementatie. Hiermee kunnen in JML geschreven interfacespecificaties verwijzen naar deze 'geschiedenissen' en attributen om implementatiegedrag te definiëren.

Om de toepasbaarheid van de benadering gebaseerd op geschiedenisredenering te demonstreren, hebben we in Hoofdstuk 5 enkele kernmethoden van de Java Collection interfaces beschreven met behulp van de *uitvoerbare* geschiedenisgebaseerde methode (de EHB-methode). Deze aanpak hanteert een weergave van geschiedenis-

sen als Java-objecten in het heapgeheugen. Deze representatie maakte echter gebruik van zogenaamde pure methoden in de specificatie. Hoewel deze methodologie in theorie functioneert, bleken de pure methoden in de praktijk, vooral bij geavanceerd gebruik, te leiden tot aanzienlijke overhead en complexiteit in het bewijsproces.

Om de geschiedenisgebaseerde methode verder te verfijnen, behandelt Hoofdstuk 6 de integratie van abstracte datatypes (ADT's) in de KeY theorem prover. Dit gebeurt via een innovatieve aanpak die datatypes modelleert met behulp van Isabelle/HOL als een interactieve back-end, en Isabelle-theorema's vertaalt naar door gebruikers gedefinieerde taclets in KeY. We conceptualiseren geschiedenissen als elementen van een ADT, onderscheiden van de door Java gebruikte types in de EHB-benadering (Hoofdstuk 5). Hierdoor kunnen de Java-programma's die geverifieerd worden de geschiedenissen zelf niet wijzigen. We duiden dit aan als de *logische* geschiedenisgebaseerde benadering (LHB-benadering). We hebben aangetoond hoe extern in Isabelle/HOL gedefinieerde ADT's ingezet kunnen worden binnen JML-specificaties en KeY-bewijzen voor Java-programma's. Als een diepgaandere casestudie hebben we de methode addAll gespecificeerd en de correctheid van de eigenschappen van zijn cliënten geverifieerd. Bovendien hebben we geavanceerde, realistische scenario's onderzocht waarbij meerdere instanties van dezelfde interface betrokken zijn.

Hoofdstuk 7 focust op het gebruik van hiërarchie binnen objectgeoriënteerde programma's. Deze hiërarchie volgt natuurlijk uit gedragsafhankelijke subtyperingen, waarin de subtypen niet alleen moeten overeenkomen met de methodesignaturen zoals gedefinieerd door hun supertypen, maar ook het beoogde gedrag moeten volgen. Voor dit doeleinde ontwikkelen we een algemene theorie voor op geschiedenis gebaseerde verfijning, die ingezet wordt om de relatie tussen subtypen te verifiëren. Elke interface en klasse koppelen we aan een geschiedenis die de reeks methodenaanroepen weergeeft die op het object zijn uitgevoerd sinds zijn creatie. De relatie tussen subtypen wordt uitgedrukt in termen van een projectierelatie, die de connectie tussen subtypen en supertypen baseert op de projectie tussen hun respectievelijke geschiedenissen, waarbij elk type zijn eigen unieke geschiedenis heeft. Door middel van een doorlopend voorbeeld tonen we de praktische bruikbaarheid van onze aanpak aan.

Het programmeren op basis van interfaces vormt een fundamenteel principe binnen objectgeoriënteerd programmeren en speelt een centrale rol in de meeste standaardbibliotheken, die een hiërarchie van interfaces en klassen bieden ter vertegenwoordiging van objectcontainers. We stellen technieken voor die in staat zijn om de structuur van klassen, interfaces en hun hiërarchie te specificeren en verifiëren. Aan de hand van het Java Collection Framework als casestudie, demonstreren we de effectiviteit van onze technieken. Hiermee biedt dit proefschrift significante bijdragen en nieuwe inzichten die van waarde zijn voor de onderzoeksgemeenschap en toekomstige softwareverificatieprojecten.

Acknowledgements

Getting through the journey of a Ph.D. is no small feat: it's tough and demands a lot, but also filled with rewards and growth. I could not made this challenging without a lot of people who have been there for me, offering guidance, supporting, and inspiring me. I wish to express my gratitude here to those who made this journey possible.

First and foremost, I want to extend my deepest appreciation to my supervisors, Prof.dr. F.S. de Boer and Prof.dr. M.M. Bonsangue. Thank you for offering me the chance to pursue my Ph.D. in the field of formal methods. Frank, your guidance during the pivotal moments of my academic journey were invaluable. Your unwavering support and insightful advice have been instrumental in my Ph.D. research. It's truly a privilege to have had the opportunity to work under your guidance throughout my doctoral journey. Marcello, I feel incredibly lucky to have met you five years ago in Beijing. I'm deeply thankful for your mentorship and generous support during my Ph.D. studies.

Next, I want to thank Hans-Dieter Hiep. I appreciate all the discussions and collaborations with you as a daily supervisor throughout this long PhD journey. I have experienced many moments of frustration and feeling lost in research. No matter when I was confused or fell in trouble, you always tried your best to help me. I am thankful for what you have done as a co-author and as a friend. I also want to express my gratitude to my other co-author, Stijn de Gouw. It was a pleasure and an honor to collaborate with you.

I would like to express my appreciation to my doctorate committee members: Prof. Dr. R.V. van Nieuwpoort, Prof. Dr. H.C.M. Kleijn, Prof. Dr. M.-C. Jakob, Dr. E. Poll, Prof. Dr. M. Sirjani, Dr. A.W. Laarman. Thank you for taking the time to read my Ph.D. thesis. Your comments have greatly improved the state of the thesis.

I want to express my gratitude to all the members of the formal method group at CWI: Farhad, Benjamin, Kasper, Luc. I enjoyed the many interesting lunchtime discussions with you. Thank you for sharing your knowledge and providing insightful suggestions about my research. I feel lucky to have made so many friends in Leiden and across the Netherlands, which made my studies much more enjoyable and less stressful. To my best friend in Leiden, Hui Feng, who was the first person I got to know upon my arrival: your support in both my research and personal life has been incredibly helpful.

I also want to thank my friend in China who has provided me with care and assistance from afar. Despite the vast distance between us, I can still sense your concern and support. I particularly want to express my gratitude to my lifelong best friend, Zhiyu Gu, who also designed the cover of this thesis.

Finally, I would like to give my greatest thanks to my family: my parents and my younger brother, Jinde Bian (David). To my father and my mother: though you may not understand my research, your unconditional love, encouragement, and support have never faltered. Your belief in me, even in moments of doubt, has been a guiding light. To my brother David: your presence is a source of joy and fortune in my life, and I am immensely thankful for it. My special appreciation goes to my husband, Mingrui Lao. In your thesis acknowledgment, you recognized me as your girlfriend, and now, it fills me with great joy to express my gratitude to you as your wife. Your constant encouragement and company have been pillars of strength for me. The journey has been remarkable, and I look forward to the many more beautiful moments we will create together.

Jinting Bian
May, 2024
Leiden, the Netherlands

Curriculum Vitae

Jinting Bian was born in Taiyuan, China, on December 18, 1994. In 2013, she started her Bsc. study at Changzhi University in Changzhi, ShanXi, China, and received her Bsc. degree in 2017. After that, she started her Msc. studies in Computer Security and Resilience at the University of Newcastle upon Tyne in Newcastle, the United Kingdom, and obtained her Msc. degree in 2018.

In October 2019, she started her PhD research supported by the China Scholarship Council (CSC No. 202007720094). She worked at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, the Netherlands, and the Centrum voor Wiskunde & Informatica (CWI), Amsterdam, the Netherlands, under the supervision of prof.dr. F.S. de Boer and prof.dr. M.M. Bonsangue. Jinting's research interests include formal methods, type theory, and program correctness. She has published papers in international journals and conferences, including Formal Methods and Formal Methods in System Design.

Publication List

- **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, volume 61, pages 63-89.
- **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 Proceedings of the 24th International Symposium on Formal Methods (FM 2021), LNCS 13047, pages 255-272, Springer.
- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). History-based specification and verification of Java collections in KeY. In Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020), LNCS 12546, pages 199-217.
- Hiep, H. A., **Bian, J.**, de Boer, F. S., de Gouw, S. (2020). A Tutorial on Verifying LinkedList Using KeY. Deductive Software Verification: Future Perspectives, LNCS 12345, pages 221-245. Springer.
- de Boer, F. S., de Gouw, S., Hiep, H. A., **Bian, J.** (2022). Footprint Logic for Object-Oriented Components. In Proceedings of the 18th International Conference on Formal Aspects of Component Software, LNCS 13712, pages 141-160. Springer.
- Hiep, H. A., Maathuis, O., **Bian, J.**, de Boer, F. S., de Gouw, S. (2022). Verifying OpenJDK's LinkedList using KeY (extended paper). International Journal on Software Tools for Technology Transfer, volume 24(5), pages 783-802.
- Hiep, H. A., Maathuis, O., **Bian, J.**, de Boer, F. S., van Eekelen, M., de Gouw, S. (2020). Verifying openjdk's linkedlist using KeY. In Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020), LNCS 12079, pages 217-234.

Propositions

pertaining to the thesis

Reasoning about object-oriented programs: from classes to interfaces

by Jinting Bian

1. The main bottleneck in the verification process is not the verification itself, but the formulation of specifications. [Chapter 3]

2. The history-based reasoning approach provides a way to show the satisfiability of specifications by a witness implementation of the interface, making it possible to reason about the state-hidden interface. [Chapter 4]

3. The selection of abstractions of history in a concrete program requires careful consideration. [Chapter 5 & 6]

4. It is necessary to verify the correctness of the subtype relation in the design stage, especially in the case of complex systems. [Chapter 7]

5. Although the cost upfront for ensuring program correctness is expensive and the benefits come late (even after time to market), it is still worthy.

6. The correctness of the theorem prover used in a formal verification requires careful attention.

7. Testing can reveal the presence of faults in software, while formal verification aims to prove the absence of failures; both of them are indispensable and irreplaceable.

8. The rapid pace of technological change and the demand for new features pose a significant challenge to the adaptability and effectiveness of formal verification in system development.

9. Intelligent dialogue systems can help in the development of formal methods, yet the techniques and expertise required for formal methods cannot be replaced by artificial intelligence.

10. Open-source projects benefit from community contributions for faster bug identification and resolution, but this doesn't mean they are bug-free.