Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

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.

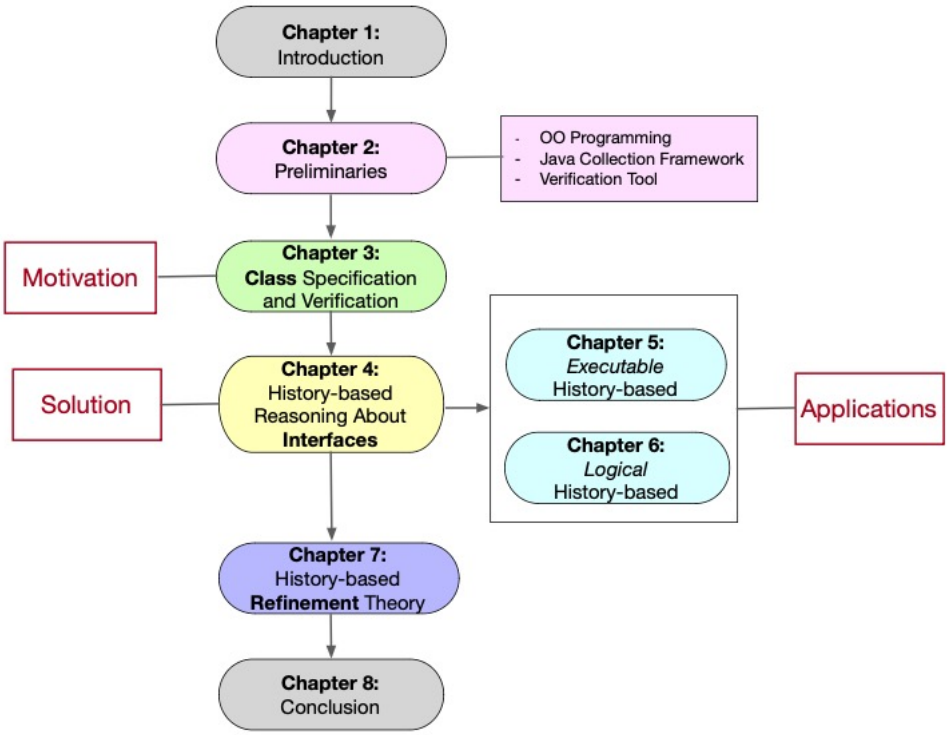

Figure 1.1: Thesis structure.

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.