Experience Report on Formally Verifying Parts of OpenJDK's API with KeY

Alexander Knüppel, Thomas Thüm, Carsten Pardylla, and Ina Schaefer
TU Braunschweig, Germany
{a.knueppel, t.thuem, carsten.burmeister, i.schaefer}@tu-bs.de

Deductive verification of software has not yet found its way into industry, as complexity and scalability issues require highly specialized experts. The long-term perspective is, however, to develop verification tools aiding industrial software developers to find bugs or bottlenecks in software systems faster and more easily. The KeY project constitutes a framework for specifying and verifying software systems, aiming at making formal verification tools applicable for mainstream software development. To help the developers of KeY, its users, and the deductive verification community, we summarize our experiences with KeY 2.6.1 in specifying and verifying real-world Java code from a users perspective. To this end, we concentrate on parts of the Collections-API of OpenJDK 6, where an informal specification exists. While we describe how we bridged informal and formal specification, we also exhibit accompanied challenges that we encountered. Our experiences are that (a) in principle, deductive verification for API-like code bases is feasible, but requires high expertise, (b) developing formal specifications for existing code bases is still notoriously hard, and (c) the under-specification of certain language constructs in Java is challenging for tool builders. Our initial effort in specifying parts of OpenJDK 6 constitutes a stepping stone towards a case study for future research.

1 Introduction

In particular for safety-critical systems, the main goal of formal methods is to increase confidence in the correctness of a software system by providing means to mechanically reason about it [9, 13, 46, 47]. Driven by research, formal methods have experienced significant advancements over the last decades. Besides lightweight methods like testing and code reviews, formal verification techniques, such as deductive verification [2, 5, 11, 24, 49] or model checking [12, 44], can be used to ensure that a program *behaves correctly* by proving that it adheres to a formal specification. Recent successes even show that verifying large-scale software systems written in real-world programming languages is not only theoretically, but indeed practically feasible [3], and that bugs in real-world software used by millions of devices can be identified [18].

A downside is that formal verification has not yet found its way into industrial software development. One reason is the substantial specification effort, which is particularly uneconomical for legacy software systems [4]. In particular, there exist serious doubts about the cost-effectiveness (i.e., return of investement) of formal methods [29]. Another reason is that developers regard most formal methods as difficult to understand and apply, as full automation is often impossible due to the undecidability of the halting problem [45]. During the verification phase, there is often a demand for user interaction, such as providing loop invariants. User interaction, however, requires highly specialized expert knowledge, which is unreasonable for typical software developers [2].

While most emphasis in the formal methods community is put on the scalability of the verification phase, scalability of the specification phase has only gained momentum recently [3, 4]. Applying *De-*

sign by Contract, a methodology in the context of deductive verification based on Hoare triples [38], developers specify the behavior of methods with contracts comprising first-order preconditions and postconditions. Contracts are typically annotations in source code close to the implementation [2, 15, 38]. The rationale is whenever a caller guarantees the precondition, the method guarantees its postcondition. A caller can then reason about its correctness with respect to its own contract.

However, even when applying Design by Contract in the industry, software developers typically only assure the quality of a small subset of the code base by writing or modifying contracts alongside their implementation. These developers are either not verifying their implementation at all or are at least never in a position to verify that their contracts are sufficient for all callers, which is often done by a different team focusing on the overall quality assurance. Consequently, a considerable amount of insufficient contracts may emerge that must be adapted subsequently. Providing strong enough contracts is error-prone and tedious [3] and, thus, cost-intensive.

We argue that, to become applicable in industrial software development, deductive verification including the formal specification phase must be supported by easy-to-use tools with a high degree of automation, such that typical software developers can already contribute significantly to the implementation's quality. To identify hurdles and challenges, experience reports are indispensible. Our observations are based on software written in objected-oriented languages and specified with contracts. In particular, we draw our experiences from a real-world case study, where we specify and verify parts of OpenJDK's Collections-API with the Java Modeling language (JML) [32]. For the verification phase, we use the state-of-the-art verifier KeY version 2.6.1 [1], an interactive theorem prover with a high degree of automation and a large community. Our long-term vision is to facilitate the development of specification and implementation in concert for everyday software developers. Our work contributes to this goal by investigating usability and applicability of KeY from a user's persepective, as most studies in research about specification and verification with KeY are indeed conducted by KeY developers themselves (e.g., [3, 6, 18]).

Our real-world case study exhibits that formal specification and automatic verification of APIs is in principle feasible. It also shows that significant experience is necessary and, accordingly, that current verification technology is not yet suitable for less experienced developers. In this sense, we believe that a community effort is indispensable to specify widely used APIs such as OpenJDK, which would help users tremendously to verify their own implementation against it. In summary, our main contributions are the following.

- We describe our experiences on specifying parts of OpenJDK's Collection-API with JML based on its existing informal specification (i.e., JavaDoc) and verify it using the program verifier KeY in version 2.6.1.
- We distribute OpenJDK's source code together with our formal specification online with the goal to extend it continuously.
- We discuss which characteristics impair automation in the verification phase and give examples.
- We encounter an incidence where KeY proves unsound behavior due to Java's under-specification of the maximal array length.

The paper is structured as follows. In Section 2, we provide the necessary background on contract-based deductive verification of Java programs. In Section 3, we explain how we specified parts of OpenJDK's Collections-API and discuss challenges. In Section 4, we provide a detailed experience report in specifying and verifying Java programs with KeY from a developer's perspective. We discuss related work in Section 5 and conclude this work in Section 6.

```
class Account {
  public final static int DAILY_LIMIT = 1000;
  public int balance;
  public int withdraw;
  /*@ requires withdraw < DAILY_LIMIT, amount !=0;
    @ ensures (\result == (\old(withdraw) − amount < DAILY_LIMIT))
    @ && (\result ==> withdraw == \old(withdraw) − amount)
    @ && (\result ==> balance == \old(balance) + amount);
    @ assignable withdraw, balance;
    @/
  boolean update(int amount) {...}
}
```

Listing 1: An Account Implementation with Contracts in JML

2 Formal Verification of Java Programs

Quality-assurance techniques, such as code reviews, testing, and formal methods, are critical for safety-related software. A prominent approach that is part of many modern programming languages is to use *assertions* [22], which are propositional formulas that should always be satisfied at given locations of method execution. A generalization of assertions is the design-by-contract paradigm [35, 38, 39], which comes with dedicated specification languages supporting flavors of first-order logic. Contracts decorate methods of object-oriented code with *preconditions* ψ and *postconditions* ϕ, and classes with *class invariants*. Preconditions describe what a method can assume and must be provided by callers of that method. Postconditions describe what a method must guarantee if its preconditions are fulfilled. Invariants must always hold (i.e., before and after method execution). While assertions are typically checked at runtime, *deductive reasoning*, as applyed by theorem provers, is used to verify source code statically [48]. A program together with its specification (i.e., contracts) is translated into a flavor of dynamic logic. The resulting logical formula for a method m is then proved to always hold by systematically applying inference rules to it.

For specification, there exist numerous languages with support for contracts, such as Eiffel [39], Spec# [2], and the Java Modeling Language (JML) [32]. For the purpose of this paper, we focus on JML, a contract-supporting extension of Java for specification. The reason is that Java is far more widespread than the other languages and highly applied in industry and research. In Listing 1, we give an example of a concrete contract written in JML of a method update in a class Account. Method update manages the transfer of money from and to respective accounts. The precondition is denoted by keyword requires and states that callers of method update must ensure that (a) input amount is not equal to 0 and that (b) value of field withdraw is less than the value of the field DAILY_LIMIT. The postcondition is denoted by keyword ensures and states that method update guarantees to update withdraw and balance, whenever the daily limit is still not reached. In a postcondition, keyword old refers to the state of the expression before method execution and keyword result represents the return value. Moreover, the framing clause is denoted by keyword assignable and describes which part of the memory is allowed to be changed by the method's implementation. If all classes are known, the framing clause is syntactic sugar and may also be expressed in the postcondition by using old(v) == v for all locations v that should remain unmodified.

As can be seen in Listing 1, JML provides means to specify the intended behavior of a method close to the implementation. Besides method contracts, JML also allows to specify invariants for fields, loops with *loop invariants*, and even blocks (i.e., scoped statements in curly braces) with *block contracts* inside an implementation. Additional specifications are often necessary to increase automation and decrease interaction in the verification phase.

To investigate the difficulties of specifying and verifying source code from a users perspective, we focus on KeY [1], an interactive theorem prover for JML-specified Java programs with a highly active community. KeY is a frequently used verification system for JML with the goal to bridge the gap between research and industry. In particular, KeY translates specification and implementation to an extension of dynamic logic called *Java Dynamic Logic* [1]. Resulting proof obligations are processed step-wise based on a combination of symbolic execution [1] and weakest precondition calculus [27] and are either closed automatically or remain open to be manually inspected by a user.

The goal of *Design by Contract* is to write implementations together with their contracts in concert [38]. Despite the fact that most software developers are non-experts in formal verification, they typically know all requirements that are important for the code they introduce or modify and, thus, should be supported by the verification system in use to write concise and comprehensible formal specifications. Moreover, to ease the process of specification and increase applicability of formal verification even into the realm of mainstream software development, the verifying tool chain has to provide a high degree of automation, which is in-line with the Spec# experience [2]. Besides the fact that KeY was initially designed to be used interactively [1], it provides numerous means to automate the verification process. For instance, KeY applies sophisticated built-in strategies to find proofs automatically. Developers may even define and add their own strategies. Moreover, KeY offers a considerable amount of parameters that control how the automatic proof search behaves. Setting the right parameters purposefully requires expertise, but also allows a user to decrease the verification effort significantly [30].

3 Contract-Based Specification of the Collection-API

A challenge to address in formal verification is to formally specify a given implementation sufficiently, such that it can be verified automatically – in particular when performed by less experienced developers. To gain experiences in this regard and identify hurdles and challenges for typical software developers, we carried out a real-world case study, for which we decided to specify parts of OpenJDK 6. Reasons to use OpenJDK are manifold. One of our goals is to specify and verify widespread and highly applied real-world software. Whereas building software from scratch was therefore not an option, OpenJDK qualifies for these requirements and is also open source. Moreover, OpenJDK is free to distribute, even when the source code is altered (e.g., adding JML contracts). Oracle's JDK disqualifies for the very same reason. Another point is that reconstructing a developers intention to develop a formal specification is difficult (i.e., in case of legacy systems). However, OpenJDK already provides a comprehensive informal specification in its JavaDoc, which eases the development of a formal specification.

3.1 Specifying the Java Collection-API

For specifying methods in Java programs with contracts, we use the Java Modeling Language (JML). In Listing 2, we exemplify the process of specifying a method based on an informal specification on the method copyOf for class Array. At the top we depict the informal specification provided in the JavaDoc comments. In essence, the informal specification covers the following aspects if method copyOf successfully terminates.

1. The result is a new array with given length newLength.
2. If newLength is less than the length of original, the resulting array is truncated.
3. If newLength is greater than the length of original, the resulting array is padded with null-elements.

```
/**
Copies the specified array, truncating or padding with nulls (if necessary) so the copy
has the specified length. For all indices that are valid in both the original array and
the copy, the two arrays will contain identical values. For any indices that are valid in
the copy but not the original, the copy will contain null. Such indices will exist if and
only if the specified length is greater than that of the original array. The resulting
array is of exactly the same class as the original array.

@param original the array to be copied
@param newLength the length of the copy to be returned
@return a copy of the original array, truncated or padded with nulls
to obtain the specified length
@throws NegativeArraySizeException if <tt>newLength</tt> is negative
@throws NullPointerException if <tt>original</tt> is null
@since 1.6
*/
public static Object[] copyOf(Object[] original, int newLength)
```

```
/*@
  @ public exceptional_behavior
  @ requires newLength < 0;
  @ signals (NegativeArraySizeException e) true;
  @
  @ also public exceptional_behavior
  @ requires original == null;
  @ signals (NullPointerException e) true;
  @
  @ also public normal_behavior
  @ requires original != null && newLength >= 0;
  @ ensures \result != null && \fresh(\result) && \result != original
  @  && \typeof(\result) == \typeof(original) && \result.length == newLength;
  @ ensures (\forall int i; 0 <= i && i < \result.length &&
  @  i < original.length; original[i] == \result[i]);
  @ ensures (\forall int i; original.length <= i &&
  @  i < newLength; \result[i] == null);
  @*/
public static /*@ nullable pure@*/ Object[] copyOf(/*@ nullable @*/
     Object[] original, int newLength)
```

Listing 2: JavaDoc and Signature of Arrays.copyOf()

4. Values of all indices that are valid in original and the resulting array are identical.
5. Array original and the resulting array have the same type.

Below the JavaDoc comment, we present the JML contract that we derived for that informal specification. In concert with the informal specification, the contract comprises three specification cases confined by keyword also; in two cases, exceptions are thrown when newLength is negative or array original equals null. The third specification case depicts the intended behavior as explained before.

The *Java Collection-API* provides an architecture to temporarily store and manipulate a group of objects. The interface java.util.Collection is the foundation for numerous data structures and is, for instance, implemented by java.util.List and java.util.Set. In the specification process, we concentrated on a small number of methods of the collection interface that we wanted to specify and verify. Some prominent methods of the collection interface we focused on are the following.

- int size(): returns the number of objects.
- boolean isEmpty(): informs whether the number of objects is zero.
- boolean contrains(Object): informs whether the collection holds a specific object.
- boolean add(Object): adds an object and returns true if the collection changed.
- boolean remove(Object): removes an object and returns true if the collection changed.
- void clear(): removes all objects.
- ...

Here, we follow a bottom-up approach in specifying the implementation; we first specify and verify less complex methods associated with a small call stack when executed. The rationale is that sufficient contracts to automatically verify such a method should be easier to derive, as only a few dependencies to called methods exist. Subsequently, we can specify and verify more complex methods associated with larger call stacks when executed by relying on specifications of called methods we derived before. Moreover, strong enough postconditions are in some cases easier to identfy compared to a top-down approach, as the postcondition of a caller depends to a great extend on the postconditions of called methods. However, as discussed in more detail elsewhere [3, 4], neither pure bottom-up nor pure top-down approaches are always superior in general.

3.2 Lessons Learned in Formalizing Parts of OpenJDK

In this section, we elaborate on our process and gained experiences of specifying parts of the Collections-API with JML. Our specifications can be found online and we invite other researchers to contribute to that repository and extend it in the future.[1]

Behavioral Subtyping.

In the presence of subtyping, contracts of a type and its subtypes should follow behavoral subtyping [36]. If T and S are both types where S is subtype of T, behavioral subtyping states that T can be replaced by S in any scenario where T is used without distorting a program's behavior. For deductive verification, this means that we can always use S.m() instead of T.m() and that their contracts must therefore be in a compatible relation (e.g., preconditions of S.m() can only be strengthened, whereas postconditions can only be weakened). The reason to follow this principle is that behavioral subtyping enables modular reasoning [34], as, even in case of dynamic dispatching, the supertype can be used. In particular, JML features a particular instance of behavioral subtyping called *specification inheritance*, which is enforced by KeY. Specification cases of an overriding method are conjoined with the supertype's specification cases by employing keyword also.

The collection interface is highly generic and refers in its informal specification (i.e., JavaDoc) to numerous properties that subclasses may either establish or not. An example for a property is whether a subclass allows storing of duplicates. To follow behavioral subtyping, a considerable amount of methods are directly specified by us in the collection interface and inherited by subclasses. Thus, we need to take these properties into account when generally specifying a method. That is why we started to introduce *model fields* in the collection interface, which allows us to parameterize contracts (i.e., parameters are

[1] `http://github.com/AlexanderKnueppel/OpenJDKwithJML/tree/F-IDE18`

```
/*@
@ public model boolean supportsDuplicates;
@ public model boolean supportsNull;
@ public model boolean isOrdered;
@
@ public model instance \bigint collectionSize;
@ public model instance nullable Object[] elements;
@
@ public model instance \locset changeable;
@*/
```

Listing 3: Model Fields of the Java Collections Framework

used in contracts and instantiated in concrete subclasses individually). Model fields can only be used in JML-annotations and represent states that are evaluated in the verification process.

In Listing 3, we illustrate all model fields that we use for the Collection-API. For example, some implementations do not allow to hold duplicates (e.g., Set), which is why we specified a boolean variable named supportsDuplicates for this purpose. Classes that do not allow to hold duplicates instantiate this field with false.

Informal Specification is Often Imprecise.

OpenJDK already provides a comprehensive informal specification for most methods in form of JavaDoc comments (cf. Listing 2). An inherited problem of an informal specification is, however, its imprecise nature. A consequence is that imprecision impedes the direct translation to a formal specification. Moreover, methods may depend on numerous other methods and in the process of software evolution specifications can become outdated – especially if they are not verifiable.

We encountered that, in some cases, implicit behavior had to be made explicit, particularly for private methods. For example, the insufficient informal specification of ArrayList.fastRemove is the following: *Private remove method that skips bounds checking and does not return the value removed.* We also found cases where an informal specification was not reasonable and had to be ignored. An example is the size() method, where the informal specification states that *whenever the current size of the collection is greater than* Integer.MAX_VALUE*, value* Integer.MAX_VALUE *has to be returned.* This, however, is not compliant with the implementation. Based on our experience, informal specifications are often to a great extent incomplete and erroneous. Defects even remain incognito for years or decades, which consequently means that a direct translation from an informal specification is often impractical and has to be taken with caution.

Missing Tool Support for Contract-Based Specification.

Ideally, the verification system provides means to support the specification phase as well as the verification phase. For instance, early feedback is crucial to prevent unnecessary iterations in specification and re-verification, which is time consuming and error prone. However, following a modular approach, most currently active verification systems only focus on the method under verification and do not consider callers of that method. For instance, when a user specifies a method, some feedback on whether the contract is sufficient for callers would be helpful. Moreover, completely specifying a large code base is unrealistic. Hence, when given a set of prime methods that need to be verified, a small set of additional

```
/*@ elementData != null;
@ invariant \typeof(elementData) == \typeof(Object[]);
@*/
private transient Object[] /*@ nullable @*/ elementData;
...
/*@ public normal_behavior
  @ requires initialCapacity >= 0;
  @ ensures elementData.length == initialCapacity;
  @ assignable elementData;
  @ also
  @ public exceptional_behavior
  @ requires initialCapacity < 0;
  @ signals_only IllegalArgumentException;
  @ signals (IllegalArgumentException e) true;
  @*/
public ArrayList(int initialCapacity) {
  super();
  //this.elementData = new Object[0]; //resolves the problem
  if (initialCapacity < 0)
    throw new IllegalArgumentException(" Illegal Capacity : "+
    initialCapacity);
  this.elementData = new Object[initialCapacity];
}
```

Listing 4: Specification of ArrayList(int)

specified methods may help to decrease the verification effort significantly. In the future, identifying those methods becomes crucial to bridge the gap between research and industry.

To ease the process for industrial software developers, the hurdles of contract-based specification must decrease through better tool support as well. Although a developer must know how to specify an implementation with contracts, there is still room for improvements. For instance, there exist an enormous amount of research on automatic inference of loop invariants or generating specification cases in case of *trivial* implementations, but its practically remains to be investigated. Especially under change, when specification or implementation are easily violated, automated reasoning and feedback for resolving such violations (e.g., suggesting fixes) are needed to prevent the deployment of bugs or costly re-verification.

4 Contract-Based Verification with KeY

In the following, we exhibit our experiences and results of verifying parts of OpenJDK 6 with KeY 2.6.1 (cf. Section 4.1) and discuss challenges that software developers face when applying deductive verification automatically (cf. Section 4.2).

4.1 Experiences Drawn from Verifying OpenJDK

Invariant Checking in Constructors

In Listing 4, we illustrate the normal and exceptional specification of ArrayList's constructor. Whereas the normal behavior is verifiable, the exceptional behavior is indeed not. The reason is that if an exception is thrown, this.elementData will not be instantiated (i.e., remains null). However, this contradicts the invariant stating that this.elementData must not be equal to null. The question here is whether the invariant should be checked even if object construction fails. We postpone the question to Section 4.2.

Resolution: To resolve this issue and prove the constructor's correctness in the exceptional case, we can modify the implementation and instantiate this.elementData with zero elements before an exception can be thrown. However, checking the invariant in this specific case when object construction fails seems to be unnecessary. Providing means – preferably though an extended specification – to hinder invariant checking in specific cases would be desirable.

Indeed, there exists the possibility to explicitly declare the constructor as a helper method using the keyword helper in front of the method's name [33]. Helper methods exclude checking invariants in their pre- and postconditions. In this specific example, however, using helper is not an appropriate solution. First, we may only want to hinder checking invariants in particular specification cases (i.e., the exceptional behavior). Second, the reference manual for JML states that helper methods and constructors need to be declared as private.

Pure Methods without Specification

A different problem we encountered by specifying List.indexOf(Object) was that the method could not be verified automatically due to the usage of Object.equals(Object). List.indexOf(Object) returns the lowest index in a list where its element equals the input object or -1 if there is no such element. In its informal specification, equality is based on the equals method of the respective object. Object.equals() is used in the specification as well as the implementation, but does not have a specification itself. Thus, each call of it is replaced by its implementation in the verification process. If we replace each call to Object.equals() in the implementation with == manually (which is generally against the intention of List.indexOf(Object)) the method becomes verifiable. Replacing Object.equals(Object) in the specification with ==, however, does not help.

Resolution: Our assumption is that pure methods called in the specification are treated differently than in the implementation. In both cases, method calls where no contract exists should be replaced by its implementation, which apparently only happened in the specification.

Underspecification of the Java Semantics (Maximal Array Length)

We experienced an issue with method ArrayList.toArray(), when we tried to verify it. KeY does not check whether the length of an array is *too big*. In particular, there is no maximal array length specified in the Java language specification and, thus, KeY, as well as other tools, have no obligation to provide a check for it. Problematic is that developers of virtual machines can set their own maximal array length, which is often around Integer.MAX_VALUE - 4 (i.e., specific bytes are reserved for header information), but also may change from version to version. A typical user would expect the maximal array length to be equal to the maximal integer value (i.e., according to the informal specification).

One goal of design by contract is to render *defensive programming* needless (i.e., checking in the implementation that the input is in the right range), as it can be considered redundant to a formal specification. Moreover, formal verification is a means to not only ensure safety properties but also to prevent security-related problems. Both objectives are invalidated by this issue. In practice, an attacker could exploit this problem by providing an array as input of approximately 2GB in size, which would consequently crash the program.

Resolution: KeY and other tools could offer the possibility to dynamically set the maximum length of an array in its front-end (i.e., this option would behave like a global invariant for all arrays). Otherwise, users of KeY and possibly other verification systems must be aware of this problem and provide maximal array lengths themselves.

Parameterization

A different challenge we faced is parameterization. In the front-end of KeY, a user can select numerous options to control how KeY should try to automatically solve the current verification task. Some options are trivial, such as *method call treatment* (i.e., whether the contract of a called method is used or the implementation is inlined). Other options, however, are unclear to typical users and require deep knowledge about KeY's underlying theory and solving procedure. This is particularly a problem, as various options and specific combinations thereof strongly influence provability and verification effort. Additionally, the number of options rather increases with each new version [30]. Finding the right settings for the current verification task is therefore indispensable. For instance, we faced a problem during the verification of method `ArrayList.add(Object)`, where it could only be verified if we ignore integer overflows (i.e., there exists a specific parameter to either ignore or check for overflows, or directly rely on the Java semantics for integers). This problem is connected with the unspecified length (i.e., field size) of a collection.

Resolution: One ad-hoc solution is to perform trial and error, which is exceptionally time consuming, but consequently leads to a set of prime configurations for different verification tasks. A more sophisticated approach was proposed in a complementing study [30], where we conducted an empirical study on the influence of parameters with respect to provability and verification effort. Such experiments help to derive a guideline that a user can follow to identify when to use which option. One can even consider setting options automatically, when the implementation is easy enough to process by an algorithm, or at least extended tool support by integrating a recommendation system.

Summary

In Table 1, we summarize all initially specified methods and respective proof results. On average, 71% of all methods could be verified automatically. We wrote a total of 175 lines of JML specification in the presented classes (excluding some inherited specification on the interface level) over the course of four person months. In this process, we encountered numerous obstacles, which we either resolved (cf. Section 4.1) or which may require additional effort by the community. Namely, these obstacles were the problem of integer overflow, the problem of checking invariants when object construction fails, the problem with `equals()`, the problem with too big arrays, and a problem with `getClass()`, which internally depends on the `.class` field. `.class`, however, was not parsable by KeY. Based on these results, we conclude that, in principle, verification of real library code is practically in reach, but the specification process is extremely time-consuming.

A cumulated overview of lines of Java code, lines of JavaDoc, and written lines of JML specification is depicted in Table 2. Field *Other* refers to any line that is neither associated with an invariant, a precondition, nor a postcondition (e.g., keyword `nullable` or exception handling through `signals`). The specification contains 50% more lines than the actual implementation.

Class	Method	# spec. cases	Proof result
ArrayList	add(Object)	6	Overflow problem
ArrayList	*constructor*()	1	Proven
ArrayList	*constructor*(int)	2	Invariant problem
ArrayList	clear()	2	Proven
ArrayList	contains(Object)	3	Proven
ArrayList	ensureCapacity(Object)	3	Proven
ArrayList	fastRemove(int)	1	Proven
ArrayList	indexOf(Object)	1	Works with ==, not with equals
ArrayList	isEmpty()	1	Proven
ArrayList	outOfBoundsMsg(int)	1	Proven
ArrayList	rangeCheck(int)	1	Proven
ArrayList	rangeCheckForAdd(int)	1	Proven
ArrayList	remove(Object)	3	Works with ==, not with equals
ArrayList	size()	1	Proven
ArrayList	toArray()	1	Proven - but Array instantiation problem.
ArrayList	trimToSize()	1	Proven
Arrays	copyOf(Object[], int)	3	Uses getClass()
Arrays	copyOfRange(Object[], int, int)	4	Uses getClass()
Math	abs(int)	1	Proven
Math	max(int, int)	1	Proven
Math	min(int, int)	1	Proven

Table 1: Specified Methods and Proof Results of our Real-World Case Study

4.2 Misunderstandings from a User's perspective

Occasionally, KeY acted differently than we expected. In the following, we want to highlight some of these occurrences in more detail to give tool builders feedback with respect to practical application and to help users who encounter the same situations.

Array Instantiation - Nullable

In Listing 5, we depict two identically implemented methods. The specification of the second method is extended by keyword nullable for its return value. The second method is verifiable, whereas the first method is not. The reason is that keyword nullable does not only affect the reference object itself but also all of its elements. Indeed, the default behavior in JML enforces that all reference types must be non-null and, thus, elements of the instantiated array must also be non-null. In case of arrSize>0 and non-primitive data, however, both methods fill the array with null elements according to Java's default behavior for initializing reference types.

This *trivial* mistake was hard to spot for us in the first few attempts. The open proof goal in KeY and he symbolic execution debugger did also not provide enough information to resolve this issue. In our experience, developing techniques to infer trivial specifications from the code to at least give suggestions and feedback to users about what may be missing is crucial in the future to speed up the specification process.

				Lines of JML				
Classes	Methods	Java lines	JavaDoc lines	Invariant	Requires	Ensures	Other	**Total**
ArrayList	16	96	57	11	30	29	46	**116**
Arrays	2	12	35	0	10	14	24	**48**
Math	3	6	23	0	0	7	4	**11**
Total	**21**	**116**	**115**	**11**	**40**	**50**	**74**	**175**

Table 2: Statistics on the Specified Part of OpenJDK

```
/*@ public normal_behavior
  @ requires arrSize >= 0;
  @ ensures \result != null;
  @ ensures \result.length == arrSize;
  @*/
private static Object[] cArrayNotWorking(int arrSize){
   return new Object[arrSize];
}

/*@ public normal_behavior
  @ requires arrSize >= 0;
  @ ensures \result != null;
  @ ensures \result.length == arrSize;
  @*/
private static /*@ nullable @*/ Object[] cArrayWorking(int arrSize){
   return new Object[arrSize];
}
```

Listing 5: Two Identical Methods with Almost Identical Specifications

```
public class ExceptionalConstructor {
  public static final List<ExceptionalConstructor> created = new ArrayList<>();
  public boolean initialized;

  public ExceptionalConstructor(boolean throwing) {
    super();
    created.add(this);
    if (throwing) {
      throw new RuntimeException();
    }
    initialized = true;
  }
}
```

Listing 6: Example of a Constructor that Saves the Reference in a Static Field

Invariants in Constructors

As mentioned before, invariants are checked in constructors even when object construction fails. Despite the fact that this behavior does not seem to be intuitive, we depict a minimal example in Listing 6, where we believe that this behavior is indeed reasonable. Although the custom constructor of class ExceptionalConstructor may fail, its reference is saved in a static field. In this case, the constructor of

```
public class ArrayStoreValid{
    /*@ invariant arr != null;
      @ invariant arr.length == 1;
      @*/
    private transient /*@ nullable @*/ Object arr[];

    /*@ public normal_behavior
      @ ensures true;
      @*/
    public void set(Object o) {
        arr[0] = o;
    }
}
```

Listing 7: Minimal Example of an Array Access

ExceptionalConstructor would not return the reference, but saving the reference is possible, because, according to Java's semantics, the object is created even before the constructor and its superconstructors are executed. The question is, again, whether all invariants must hold in any given scenario.

A solution would be to provide a new keyword for excluding invariants in specific cases or to use a boolean *ghost field* (similar to a model field) which is set to true once the invariant is established. The invariant can then be rewritten as an implication (i.e., *@ invariant ghost ==> arr != null; @*\).

Array Access - Data Types

Given the specification of ArrayStoreValid.set(Object) illustrated in Listing 7, this method appears to be easily verifiable. However, KeY fails to verify this method and provides an ArrayStoreException. The reason is based on array arr's type; arr can take on any subtype of Object[]. Moreover, arr can hold elements of any subtype of Object. Both types may be incompatible, which is recognized by KeY. This issue can be resolved by adding the invariant *@ invariant \typeof(arr) == \type(Object[]);@*\ to explicitly inform KeY that array arr will always be of type Object[]. However, again, resolving such issues should be supported by additional tooling during the specification phase.

5 Related Work

In the following, we discuss differences to related research that also focuses on specification and verification of software systems with contracts.

A survey on different languages for behavioral contracts was done by Hatcliff et al. [24]. Besides JML, there exist alternatives for specifying Java source code, such as C4J [10] or Contract4J [51]. Other examples for languages with support for contract are Eiffel [39] and Spec# [2]. We consider our results to be generally applicable to other languages, as the specification and semantics of those contract languages is similar to JML.

For the purpose of this paper, we chose KeY 2.6.1 [1] as the primary verification system. There are a number of verifiers that have been used in substantial verification efforts. Dafny was employed in IronClad and IronFleet [25, 26], Autoproof has been used in the development of a verified Eiffel library [23, 43], and F* has been used in Microsoft's project Everest [8]. For object-oriented programming in general, there exist a number of alternatives. The KIV system [19] can be employed for the development of safety-related software, is also based on a dynamic logic, and primarily focuses on strong proof support. ESC/Java2 [16] is a static checker that finds common runtime-errors in JML-specified programs. Other systems are Krakatoa [21] for Java programs and Jesse [37] for C programs, which are both based on the Why platform for deductive reasoning [52]. Both systems require high expertise, as proof scripts are manually written by users and also appear to be highly brittle [50]. Typically, verification of imperative languages following the design-by-contract paradigm are based on first-order logic, such as KeY that is based on Java dynamic logic. Examples of interactive theorem provers for higher-order logic are Coq [7], Isabelle/HOL [40], and PVS [41]. Nevertheless, for our real-world case study we chose KeY, because none of these systems is directly designed to support verification of mainstream programming languages by mainstream software developers.

Despite being a research topic for decades, formal methods are still not widely applied by industrial software developers. An often in research overlooked challenge is the difficulty of formally specifying real-world software. Thus, only a few publications exist that either discuss necessities for formal specifications to become widely applicable or discuss real-world case studies. Beckert et al. [4] discuss strategies and requirements for contract-based specification and post-hoc verification of imperative legacy code. They draw their experience from two case studies, namely the PikeOS microkernel [28] verified with VCC [14] and the sElect voting system [31] verified with KeY. Baumann et al. [3] report on their experience of the verification tasks in the Verisoft XT project. They also verified the PikeOS microkernel using VCC and discuss challenges they encountered in bridging informal and formal specification. Gouw et al. [18] investigated the correctness of TimSort with KeY. They indeed discovered a bug in its implementation and derived a bug-free implementation that was proven correct. Beckert et al. [6] conducted another case study by formally specifying JDK's dual pivot quick sort method with JML and proving it correct in KeY. Estler et al. [20] present a study that investigates how contracts are used in the practice of software development. They analyzed a total 21 projects in the programming languages Java, Eiffel, and C#, which all were following the design-by-contract methodlogy to some extent. Pariente and Ledinot [42] conducted a case study on formal verification of industrial C code using the verification system Frama-C [17]. Their results are in-line with ours, as they vividly exhibit that deductive verification of industrial source code requires considerable expertise.

6 Conclusion and Future Work

Driven by research, formal verification of highly complex software systems made considerable progress in the last decades. Beyond its purpose to increase trust in the correctness of a program, it also prevents safety-related bugs, where life or missions are at stake. Furthermore, better tool support and advanced automation push contract-based verification even in the range of industrial software developers, which do not have to be highly specialized to apply verification techniques. However, although scalability of verification techniques is addressed significantly, scalability of the specification phase is seldom investigated and discussed.

In our real-world case study, we specified parts of OpenJDK's Collections-API with JML and verified them with the deductive verification system KeY. Our approach was to take the perspective of an inexperienced user to gain insights about challenges that a typical software developer would face. We described issues that occurred during the specification with JML and the verification with KeY and tried to present ideas that would resolve them. Our vision here is to aid developers of deductive verification tools to make them applicable to industrial software developers.

One disillusioning insight of this case study was that deductive verification still requires high expertise in the underlying proof theory. The reasons are manifold. First, if a proof cannot be closed, identifying whether the problem lies in the specification or implementation was notoriously hard, even for simple methods. Second, there exist a considerable amount of parameters to set for verification tools and particularly for KeY, each with the possibility to influence provability and verification effort. However, finding the right configuration without feedback and tool support is impractical. Third, based on Java's underspecification of the maximal array length, verification systems may verify a method that can also fail when exploited by malicious software. This issue raises awareness of the fact that formal verification and software testing should be applied in concert to increase trust in the correctness.

For deductive verification tools to become applicable by typical software developers, we believe that raising awareness of the challenges in the specification and verification process is necessary. Hence, a community effort is needed to specify widely-used APIs such as OpenJDK that users can verify their own software against. There are several directions to extend this work.

- Besides specifying and verifying a larger part of OpenJDK to gain more experiences, it is necessary to also employ other deductive verification systems, preferably for numerous programming and specification languages. This allows us to generalize some of our findings and identify common challenges in terms of scalability of the specification process and usability of the verification environment.
- Accordingly, usability of the numerous IDEs should be investigated in experimental user studies. For instance, success of verifiying source code often depends on a user-chosen parameterization, but specific parameters are hard to understand and sometimes even negligible. Tool builders need to be aware of the user's challenges.
- Additionally, identifying reliable and scalable means for inferring specifications for API-like code (semi)-automatically can ease the specification process. We identified that certain specification aspects have to be spelled out explicitly that could also be synthesized from the code – either statically or dynamically. In particular, APIs evolve and new methods for avoiding unneeded reverifications under change become crucial.

Acknowledgments.

We gratefully acknowledge Richard Bubel, Reiner Hähnle, Dominik Steinhöfel, and Stefan Krüger for fruitful discussions and valuable feedback throughout this work. Work on this paper was partially supported by the DFG Research Unit *"Controlling Concurrent Change"*, funding number FOR 1800.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt & Mattias Ulbrich (2016): *Deductive Software Verification–The KeY Book: From Theory to Practice*. Springer, doi:10.1007/978-3-319-49812-6.

[2] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte & Herman Venter (2011): *Specification and Verification: The Spec# Experience*. *Comm. ACM* 54, pp. 81–91, doi:10.1145/1953122.1953145.

[3] Christoph Baumann, Bernhard Beckert, Holger Blasum & Thorsten Bormer (2012): *Lessons Learned from Microkernel Verification–Specification is the new Bottleneck*. *SSV*, doi:10.4204/EPTCS.102.4.

[4] Bernhard Beckert, Thorsten Bormer & Daniel Grahl (2016): *Deductive Verification of Legacy Code*. In: Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Springer, pp. 749–765, doi:10.1007/978-3-319-47166-2_53.

[5] Bernhard Beckert, Reiner Hähnle & Peter Schmitt (2007): *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Heidelberg.

[6] Bernhard Beckert, Jonas Schiffl, Peter H Schmitt & Mattias Ulbrich (2017): *Proving JDKâĂŹs Dual Pivot Quicksort Correct*. In: Working Conference on Verified Software: Theories, Tools, and Experiments, Springer, pp. 35–48, doi:10.1007/978-3-319-47846-3_5.

[7] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[8] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch et al. (2017): *Everest: Towards a Verified, Drop-in Replacement of HTTPS*. In: Leibniz International Proceedings in Informatics (LIPIcs), 71, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, doi:10.4230/LIPIcs.SNAPL.2017.1.

[9] Jonathan Bowen & Victoria Stavridou (1993): *Safety-critical Systems, Formal Methods and Standards*. Software Engineering Journal 8(4), pp. 189–209, doi:10.1049/sej.1993.0025.

[10] Hagen Buchwald & Florian Meyerer (2013): *C4J: Contracts, Java und Eclipse*. Eclipse Magazin 13(3), pp. 64–69.

[11] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino & Erik Poll (2005): *An Overview of JML Tools and Applications*. Int'l J. Software Tools for Technology Transfer (STTT) 7(3), pp. 212–232, doi:10.1007/s10009-004-0167-4.

[12] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (1999): *Model Checking*. MIT Press, Cambridge, Massachussetts.

[13] Edmund M Clarke & Jeannette M Wing (1996): *Formal methods: State of the Art and Future Directions*. ACM Computing Surveys (CSUR) 28(4), pp. 626–643, doi:10.1145/242223.242257.

[14] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In: Proc. Int'l. Conf. Theorem Proving in Higher Order Logics (TPHOLs), Springer, pp. 23–42, doi:10.1007/978-3-540-74591-4_15.

[15] David R. Cok (2011): *OpenJML: JML for Java 7 by Extending OpenJDK*. In: Proc. Int'l Conf. NASA Formal Methods (NFM), Springer, Berlin, Heidelberg, pp. 472–479, doi:10.1007/978-3-642-18070-5_13.

[16] David R Cok & Joseph Kiniry (2004): *ESC/Java2: Uniting ESC/Java and JML*. In: Proc. Int'l Conf. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), 3362, Springer, pp. 108–128, doi:10.1007/978-3-540-30569-9_6.

[17] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2012): *Frama-C*. In: Proc. Int'l. Conf. Software Engineering and Formal Methods (SEFM), Springer, pp. 233–247, doi:10.1007/978-3-642-33826-7_16.

[18] Stijn De Gouw, Jurriaan Rot, Frank S de Boer, Richard Bubel & Reiner Hähnle (2015): *OpenJDKâĂŹs Java.utils.Collection.sort() is Broken: The Good, the Bad and the Worst Case*. In: Proc. Int'l Conf. Computer Aided Verification (CAV), Springer, pp. 273–289, doi:10.1007/978-3-319-21690-4_16.

[19] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg & Wolfgang Reif (2015): *KIV: Overview and VerifyThis Competition*. Int'l J. Software Tools for Technology Transfer (STTT) 17(6), pp. 677–694, doi:10.1007/s10009-014-0308-3.

[20] H-Christian Estler, Carlo A Furia, Martin Nordio, Marco Piccioni & Bertrand Meyer (2014): *Contracts in Practice*. In: International Symposium on Formal Methods, Springer, pp. 230–246, doi:10.1007/978-3-319-06410-9_17.

[21] Jean-Christophe Filliâtre & Claude Marché (2007): *The Why/Krakatoa/Caduceus Platform for Deductive Program Verification*. In: *Computer Aided Verification*, Springer, Berlin, Heidelberg, pp. 173–177, doi:10.1007/978-3-540-73368-3_21.

[22] Robert W. Floyd (1967): *Assigning Meanings to Programs*. *Mathematical Aspects of Computer Science* 19, pp. 19–32, doi:10.1090/psapm/019/0235771.

[23] Carlo A Furia, Martin Nordio, Nadia Polikarpova & Julian Tschannen (2017): *AutoProof: Auto-Active Functional Verification of Object-Oriented Programs*. *Int'l J. Software Tools for Technology Transfer (STTT)* 19(6), pp. 697–716, doi:10.1007/s10009-016-0419-0.

[24] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller & Matthew Parkinson (2012): *Behavioral Interface Specification Languages*. *ACM Computing Surveys* 44(3), pp. 16:1–16:58, doi:10.1145/2187671.2187678.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty & Brian Zill (2015): *IronFleet: Proving Practical Distributed Systems Correct*. In: *Proc. Symposium on Operating Systems Principles (SOSP)*, ACM, pp. 1–17, doi:10.1145/2815400.2815428.

[26] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang & Brian Zill (2014): *Ironclad Apps: End-to-End Security via Automated Full-System Verification*. In: *Proc. USENIX Symposium Operating Systems Design and Implementation (OSDI)*, 14, pp. 165–181.

[27] C. A. R. Hoare (2003): *The Verifying Compiler: A Grand Challenge for Computing Research*. In: *Proc. Joint Modular Languages Conference (JMLC)*, Springer, Berlin, Heidelberg, pp. 25–35, doi:10.1007/978-3-540-45213-3_4.

[28] Robert Kaiser & Stephan Wagner (2007): *Evolution of the PikeOS Microkernel*. In: *Proc. Int'l. Workshop on Microkernels for Embedded Systems (MIKES)*, p. 50.

[29] John C Knight, Colleen L DeJong, Matthew S Gibble & Luis G Nakano (1997): *Why are Formal Methods not used more Widely?* In: *Fourth NASA Langley Formal Methods Workshop*, Citeseer, doi:10.1.1.2.3395.

[30] Alexander Knüppel, Thomas Thüm, Carsten I. Pardylla & Ina Schaefer (2018): *Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY*. In: *Proc. Int'l. Conf. Interactive Theorem Proving (ITP)*, Springer, doi:10.1007/978-3-642-29044-2.

[31] Ralf Küesters, Tomasz Truderung & Andreas Vogt (2011): *Verifiability, Privacy, and Coercion-resistance: New Insights From a Case Study*. In: *Proc. Symposium on Security and Privacy (SP)*, IEEE, pp. 538–553, doi:10.1109/SP.2011.21.

[32] Gary T. Leavens & Yoonsik Cheon (2006): *Design by Contract with JML*. Available at `http://www.jmlspecs.org/jmldbc.pdf`.

[33] Gary T. Leavens & Peter Müller (2007): *Information Hiding and Visibility in Interface Specifications*. In: *Proc. Int'l Conf. Software Engineering (ICSE)*, IEEE, Washington, DC, USA, pp. 385–395, doi:10.1109/ICSE.2007.44.

[34] Gary T Leavens & David A Naumann (2006): *Behavioral Subtyping is Equivalent to Modular Reasoning for Object-oriented Programs*.

[35] Barbara Liskov & John Guttag (1986): *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, USA.

[36] Barbara H. Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Programming Languages and Systems (TOPLAS)* 16(6), pp. 1811–1841, doi:10.1145/197320.197383.

[37] Claude Marché & Yannick Moy (2012): *The Jessie Plugin for Deductive Verification in Frama-C*. *INRIA Saclay Île-de-France and LRI, CNRS UMR*, doi:10.1.1.229.3233.

[38] Bertrand Meyer (1988): *Object-Oriented Software Construction*, 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[39] Bertrand Meyer (1992): *Applying Design by Contract*. *IEEE Computer* 25(10), pp. 40–51, doi:10.1109/2.161279.

[40] Tobias Nipkow, Markus Wenzel & Lawrence C. Paulson (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Heidelberg, doi:10.1007/3-540-45949-9.

[41] Sam Owre, Sreeranga P. Rajan, John M. Rushby, Natarajan Shankar & Mandayam K. Srivas (1996): *PVS: Combining Specification, Proof Checking, and Model Checking*. In: Proc. Int'l Conf. Computer Aided Verification (CAV), Springer, Berlin, Heidelberg, pp. 411–414, doi:10.1007/3-540-61474-5_91.

[42] Dillon Pariente & Emmanuel Ledinot (2010): *Formal Verification of Industrial C Code using Frama-C: A Case Study*. Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS), p. 205.

[43] Nadia Polikarpova, Julian Tschannen & Carlo A Furia (2015): *A Fully Verified Container Library*. In: Proc. Int'l Symposium Formal Methods (FM), Springer, pp. 414–434, doi:10.1007/978-3-319-19249-9_26.

[44] Robby, Edwin Rodríguez, Matthew B. Dwyer & John Hatcliff (2006): *Checking JML Specifications Using an Extensible Software Model Checking Framework*. Int'l J. Software Tools for Technology Transfer (STTT) 8(3), pp. 280–299, doi:10.1007/s10009-005-0218-5.

[45] Hartley Rogers & H Rogers (1967): *Theory of Recursive Functions and Effective Computability*. 5, McGraw-Hill New York.

[46] John Rushby (1997): *Formal Methods and their role in the Certification of Critical Systems*. In: Safety and Reliability of Software Based Systems, Springer, pp. 1–42, doi:10.1007/978-1-4471-0921-1_1.

[47] Donald Sannella (1988): *A Survey of Formal Software Development Methods*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

[48] Johann Schumann (2001): *Automated Theorem Proving in Software Engineering*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-22646-9.

[49] Johann M Schumann (2001): *Automated Theorem Proving in Software Engineering*. Springer Science & Business Media, doi:10.1007/978-3-662-22646-9.

[50] Thomas Thüm, Ina Schaefer, Martin Kuhlemann & Sven Apel (2011): *Proof Composition for Deductive Verification of Software Product Lines*. In: Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST), IEEE, Washington, DC, USA, pp. 270–277, doi:10.1109/ICSTW.2011.48.

[51] Dean Wampler (2006): *Contract4J for Design by Contract in Java: Design Pattern-like Protocols and Aspect Interfaces*. In: Fifth AOSD Workshop on ACP4IS, pp. 27–30, doi:10.1.1.115.2281.

[52] Why Development Team: *Why: A Software Verification Platform*. Website. Available online at `http://why.lri.fr/`.