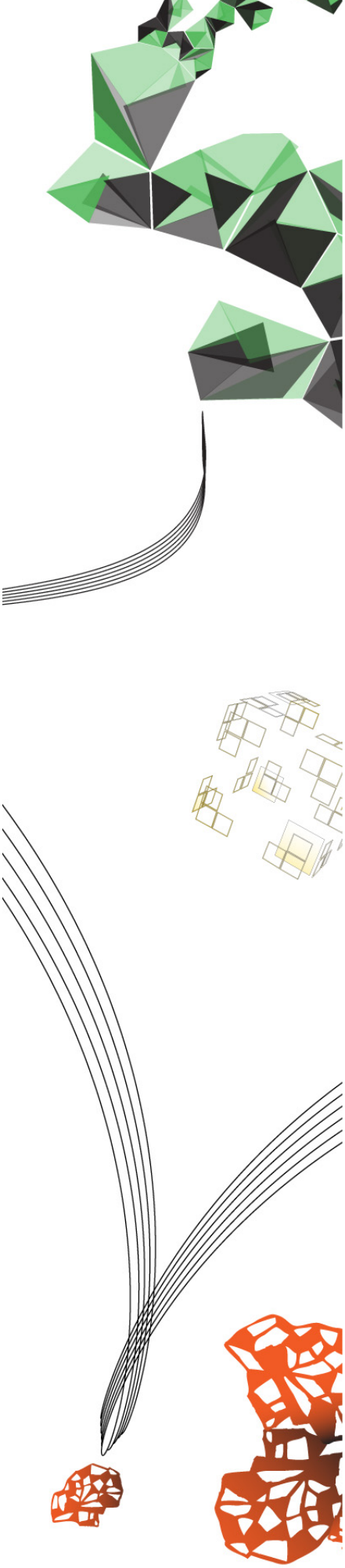


Specification and verification of selected parts of the Java Collections Framework using JML* and KeY

Jelmer ter Wal

Master's Thesis

Department of Computer Science
University of Twente

Graduation Committee:
Dr. Marieke Huisman
Dr. Wojciech Mostowski
Lesley Wevers, MSc.

October 9, 2013

UNIVERSITY OF TWENTE.

Abstract

In this thesis JML* (an extension for JML used by the static verification tool KeY) is used to formalize the behavior of several interfaces and classes, part of the Java Collections Framework. For these specifications several specification styles were used, e.g., making use of model fields, ghost fields, pure methods and abstract data types, which aids in getting an understanding of which style would contribute to better understandability, extensibility and verifiability of JML* specifications. After specifying several interfaces and classes, a large part of the specifications has been verified with KeY.

To see whether the specifications made are understandable, a group of people only familiar with basic JML has been asked to fill out a questionnaire. The questionnaire asked whether certain methods – that represent the overall style of specifications done – would be verifiable. Most of the answers provided, suggested that not just the specifications provided, but JML in general is not straightforward to understand, e.g., people have the idea specifications are not verifiable when they do not cover the complete behavior of a method, and additionally the use of model fields as well as invariants is not completely clear. The use of abstract data types (i.e., `sequences`) as well as framing of methods – specifying the set of locations that might be changed by a method – did not cause a lot of additional confusion for the participants in general.

To validate findings about understandability, extensibility and verifiability, a group of experts in the fields of JML* and KeY (i.e., the KeY developers) have been asked to fill out a questionnaire. Depending on one's experience it might be the case that a specific construct is better for understandability as well as extensibility than the other. All things considered, ghost fields seem to be worse for understandability and extensibility compared to other constructs as they need to update state for every method that affects them. Ghost fields are, however, easier for verifiability though, since – at least for KeY – they can be treated like actual code. Higher forms of abstraction, i.e., using model fields or model methods seem to be more problematic for verification as they can be very complex, and verification tools need to be provided with manual instantiations to reason about them. Moreover, higher abstraction also leads to improved understandability and extensibility.

Acknowledgements

I want to thank everyone who contributed in enabling me to finish this thesis.

I would like to thank my supervisors; Marieke Huisman, Wojciech Mostowski and Lesley Wevers. They provided me with vital feedback while writing this thesis, and gave me helpful suggestions on how to tackle different problems along the way. I would especially like to thank Wojciech who helped me understand the KeY tool better and guided me when I got stuck on verification using this tool.

Also, I would like to thank all the participants who took the effort to fill out either the understandability or specification styles questionnaire, even though it was during summer holidays.

Last but not least, I would like to thank my parents and sister, who motivated me when I needed it most, and provided a nice environment at home.

Contents

Introduction

Nowadays software is mingled in almost every aspect of life, from non-critical - but still important - software on a mobile phone, up to highly critical software for automotive and medical applications. For safety critical systems it is important that software does not contain bugs that can do harm or cause serious losses, i.e., costs lives, money or time.

One of the most cited bugs is that of the Ariane 5 launcher [26]. On June 4th 1996 the first flight of the Ariane 5 launcher ended in a crash. Within 40 seconds after lift-off the Ariane 5 deflected its flight path, broke up, and exploded. The primary cause of the crash was an overflow exception when converting the horizontal bias variable, and the lack of protection of the conversion of this variable. This eventually ceased the system responsible for calculating angles and velocities, and transmitting findings about altitude and movements to the on-board computer that executes the flight program and controls the steering mechanism, i.e., in case of an exception this system was programmed to be shutdown. As the backup system had identical software, it also got an overflow exception at a certain point and was shutdown too. Thereupon the launcher disintegrated and ended up in destruction, as designed.

That bugs like the one just described are not wanted is obvious, i.e., they have large expenses in money and time. To prevent this kind of bugs from happening, many studies have been performed. This thesis focusses on the direction of formally specifying behavior of an Application Programming Interface and verifying these specifications with a interactive verifier. Programs build on these Application Programming Interfaces can take advantage of the specifications, since these programs only need to specify additional behavior and have less proof obligations when being verified. This makes it less likely that bugs will occur in these programs.

API specification

An Application Programming Interface (API) can be used as a foundation for programmers to build programs on. Müller indicates some technical problems of APIs in [32]. He mentions that clients have to rely on documentation that tends to be imprecise and incomplete. Especially proprietary APIs – APIs for specific devices - tend to be that way as the company behind the API does not want to burn their fingers by providing too much detail about their implementation. Another problem is that there are no quality certifications for libraries available yet, and companies might not see the importance of good documentation.

As many programmers use APIs as the foundation of their software, there is a need for precise specifications. Three benefits one gets when formal specifications are applied to APIs – and next verified – are:

- Ambiguity or inconsistency that comes with normal documentation will disappear. This way, a programmer exactly knows what to expect from a specific method.
- Secondly, the programmer has the guarantee that the API will behave like specified since specifications have been verified.
- Costs and time can be spared on developing software, as only the programmer's own code has to be verified, i.e., the API is guaranteed to be correct. Programmers can use existing specifications and complement them for their own programs to also prove these programs correct.

Several attempts on verifying selected parts of the Java API have been performed. In [16, 33, 37] two attempts on specifying and verifying the Java `Collection` interface and `Iterator` interface have been performed. Peters as well as Huisman describe that they encountered unclear specifications in the informal specification. Furthermore, Peters had problems with the selected verifier tool KeY [5], in that the tool could not handle JML specifications like `\sum`, `\min` and `\max` at the time.

That it is possible to verify a complete API is shown by Mostowski, who presents a formally verified reference implementation of the Java Card API in [31]. The complete implementation has been first formally specified and next verified with KeY. Mostowski describes that only minor interactions were needed due to loop invariants that KeY could not prove itself. Although KeY can cope with the verification of the Java Card API, this does not mean that it is also possible to use KeY's power to verify part of the Java API,

because the Java Card API is substantially smaller and simpler than the full regular Java API.

Goals

The main goal of this master thesis is to gain inside in the understandability, extensibility and verifiability aspects of specifications written in JML* – an extension to JML used by KeY for modular static verification.

One of the most used parts of the Java API is the Java Collections Framework. Any substantial program made in Java will at least use a few of its classes. It has two root interfaces, `Collection` and `Map` with a number of sub-interfaces, abstract classes and classes. There will always be something that fits the needs of the programmer, and if that is not the case a programmer can easily extend a class or implement a specific interface.

This led to the following concrete goals of this master thesis;

1. Provide specifications for selected parts of the Java Collections Framework, and hereby gain insight on different specification constructs of JML* for understandability and extensibility.
2. Verifiy the specifications made, and hereby gain insight on different specification constructs of JML* for verifiability.
3. Validate the findings of the first two goals.

To accomplish the last goal, questionnaires have been made to retrieve information about understandability of the specifications made, and, understandability, extensibility and verifiability of JML* specifications in general. The different specifications constructs considered for this thesis are;

- ghost fields;
- model fields;
- abstract data types (e.g., sequences);
- pure methods; and,
- model methods.

Contributions

Concretely, this thesis describes the following contributions:

- a specification of selected parts of the Java Collections Framework;
- a verification of most part of the specifications made;
- an evaluation of the understandability of the specifications made;
- an evaluation of different specification styles (e.g., model and ghost fields) on
  - understandability;
  - extensibility; and
  - verifiability.

Thesis outline

The first part of this thesis describes the background of this study, i.e., what kind of tools do we have on writing working and correct programs. The basics of JML, and a derivative – called JML* – will be explained. Several techniques will be treated that use JML – or the derivative – to achieve checking and/or verifying correctness of specifications.

In the second part of the thesis the contributions are described. First, specifications for selected parts of several interfaces from the Java Collections Framework have been described in Chapter2, as well as providing alternative ways of writing some of these specifications. Chapter 3 discusses the verification of some classes based on the interface described in Chapter 2, after describing the usage of the tool KeY and encountered limitations during the verification. Additionally, both chapters describe findings about understandability, extensibility and verifiability of the different JML* specification constructs used.

The last part of the thesis provides an evaluation of the contributions. First Chapter 4 provides the approach that led to two questionnaires, i.e., one to retrieve information about the understandability of the created and verified specification, and one to validate the results found about understandability, extensibility and verifiability of JML* specification constructs. Chapter 5 provides the results and discussion of the two questionnaires. After which the final chapter presents conclusions and provide directions of future work.

Part I.

Background

1. Making software better

Several techniques have been conceived to rule out bugs in software. This thesis focusses on programs written in Java, a popular mainstream object-oriented language. Therefore, tools and specification languages for Java are given as an example, however similar tools and specification languages do exist, for example, for C# (NUnit [17], Pex [39], Spec# and Boogie [2]). This part of the thesis describes the background of this study, i.e., what influence do we have on writing working and correct programs, and explains where KeY comes from and what it is. Also, the basics of JML and a derivative of JML – called JML* that is used by KeY – are explained. Several techniques are treated that use JML – or the derivative – to check and/or verify the correctness of the specifications.

This chapter briefly describes unit testing in Section 1.1, Section 1.2 continues by explaining the basics of JML, which are used in Section 1.3 to describe several techniques to prevent bugs in software.

1.1. Unit testing

One of the most popular techniques for preventing bugs in software is unit testing, where every unit of code, i.e., a class or group of classes, is exposed to a series of tests, and results of invocations of methods are compared to expected results of these methods given an input. The technique gets its popularity due to the easiness of use for programmers, however, the technique – in general – lacks completeness of code coverage, and cannot be relied upon for safety critical systems. For Java there exists JUnit [4] – a unit testing framework for Java – and Hamcrest [36], that provides a matcher library to make writing unit tests easier. Another testing framework for Java is TestNG [6], which is inspired

by JUnit, but introduces additional functionality to make it more powerful and easier to use.

1.2. JML

The Java Modeling Language (JML) is a specification language for Java, and allows one to formaly specify the behavior of Java code. JML is an outgrowth of the principle of Design by Contract (DbC) introduced by Bertrand Meyer for the Eiffel programming language in 1986 [28]. With DbC the behavior of the components of a program are formally described by a so-called contract. A contract describes for each method under what conditions it may be called, and what is guaranteed about the return value and side-effects of a method call. This way a user can study a component's contract, which explains exactly what the component expects and does. Implementers are free to choose any implementation for components, as long as they adhere to the component's contract.

JML has become a large language when several projects [5, 40, 14, 27] – that targeted tool-support for the verification of Java programs – started supporting it. Because JML is a large language, not all language constructs and their semantics are totally agreed upon. Tools that support JML therefore support only a subset of the language. To prevent losing perspective, several levels of the language have been defined. There exist four basic levels, from level 0 – which contains the most common constructs of JML, for which the semantics are well understood – up to and including level 3, where each level extends on the previous level. Furthermore, there is level C for concurrent features and level X for experimental features, that might end up in one of the basic levels at a later stage. Tools that support JML are expected to support at least level 0. More information on all levels that have been defined can be found in Section 2.9 of the JML Reference Manual [24]. This thesis first explains basic features of JML – which are part of the first levels – and next addresses additional features used in the specifications later on.

JML specifications can be added to Java files with a special comment-like style. Namely, lines starting with `//@` for single line JML specifications and lines starting with `/*@` and ending with `*/` for multi-line specifications. Often you will also see lines starting with `@` between the first and last multi-line JML specification, and `@*/` for ending a multi-line specification. This is not required but helps in providing a clear seperation between specification and normal comments.

As JML is added to source code in a comment-like style, it is ignored by the Java compiler, so it is only there to formally relate specifications with a program. Tools that support JML can use these formal specifications to either validate properties at runtime, or verify whether specifications comply with the source code statically, i.e., without executing the source code. In Section 1.3 examples of these tools are discussed.

1.2.1. Method contracts

A method contract specifies the behavior of a method with *pre-* and *postconditions*. Preconditions should hold before invoking a method and specify for instance restrictions on the arguments of the method, or in what state an object should be. A precondition on a method could be that an argument of type integer is restricted with a lower and an upper bound. In JML, the keyword `requires` is followed by a precondition expression. Postconditions specify guarantees about the method, and describe how the object's state is changed by the method, or what the expected return value of a method is. Postcondition expressions are preceded by the JML keyword `ensures`, and are only guaranteed when the corresponding precondition holds.

Pre- and postconditions in JML are basically just Java expressions of Boolean type. The expressions should not have side effects and may not terminate exceptionally. The idea behind this is that specifications are written in a language familiar to the programmer, so that writing specifications has a low threshold and reading them is not difficult.

```
1  public interface Employee {
2
3    public static final int maxYearSalary = 193000;
4    public static final int retirementAge = 67;
5
6    //@ ensures \result == getAge() >= 67;
7    /*@ pure */ public boolean retirementEarned();
8
9    //@ ensures \result >= 0;
10   /*@ pure */ public int getAge();
11
12   //@ requires !retirementEarned();
13   //@ ensures \result > 0 && \result <= maxYearSalary;
14   /*@ pure */ public int getSalary();
15
16   //@ requires inc > 0 && inc <= maxYearSalary;
17   //@ requires getSalary() + inc <= maxYearSalary;
18   //@ ensures getSalary() == \old(getSalary()) + inc;
19   public void increaseYearSalary(int inc);
20 }
```

Listing 1.1: JML specified interface `Employee`

An example of JML

Listing 1.1 provides an example specified with basic JML for the methods in the interface `Employee`. Different aspects of the specifications are explained in detail below.

- `retirementEarned` specifies that one has earned a retirement when one is above the age of 67.
- The specification of `getAge` defines that its return value should always be greater than or equal to 0.
- The method `getSalary` only specifies what should be guaranteed when the `Employee` has not yet earned its retirement, namely the `Employee` should then have a salary greater than zero and less than or equal to the `maxYearSalary`.
- `increaseYearSalary(double inc)` specifies that the argument `inc` should be positive and less than or equal to the `maxYearSalary`. Also, when the amount is added to the salary it should not exceed the `maxYearSalary`. When the precondition holds the postcondition states that the return value of `getSalary` will

now hold the value of inc added to the result of `getSalary` before invocation of the `increaseYearSalary` method. To allow reasoning about the pre-state of an object the JML operator `\old(e)` is used, where `e` is an expression evaluated in the context of the pre-state.

The methods `retirementEarned`, `getAge` and `getSalary` are all specified with the keyword `pure`. This means that these methods are not allowed to have any side effects, the state allocated before the method call may not change. However, this needs elaboration as it does not exclude that methods create new objects and instantiate them. Technically speaking it is even possible to change a heap location (i.e., memory operated on by the method) during the flow of a method and changing it back to its pre-state before the method is finished, which will not result in non-purity of the method. Only pure methods are allowed to be part of a specification expression. In JML, pure methods do not change the part of the heap memory known prior to the invocation. However, they are allowed to create fresh objects on the heap and assign to fresh locations that belong to them. In a subsection on purity – see page 29 – an additional form of purity is explained.

A solution to completely specify changes made by a method is described in Section 1.3.3 when the idea of dynamic frames is explained . To denote the return value of a method, JML uses the reserved keyword `\result`.

Although not used in the example above, reference values are implicitly assumed to be non-null in JML. When one needs the fact that these values may be `null`, the dedicated JML keyword `nullable` must be used.

Comprehension constructs

Besides traditional side-effect-free Java expressions, and pure methods – which can be used for predicates in the specifications – JML defines additional constructs. `\old` and `\result` are two of them, as explained above. Also worth mentioning are the constructs `\forall`, `\exists`, `\sum`, `\min`, and `\max`. The `\forall` and `\exists` constructs can, e.g., be used for stating that an argument of a method needs an ordered array like so;

```
//@ requires (\forall int i; 0 < i && i < a.length; a[i-1] <= a[i]);
public void arrayModification(int[] a) { ...
```

Listing 1.2: Example usage of `\forall`

or the same restriction making use of `\exists`;

```
//@ requires !(\exists int i; 0 < i && i < a.length; a[i-1] > a[i]);
public void arrayModification(int[] a) { ...
```

Listing 1.3: Example usage of `\exists`

Summing and counting can be done with `\sum`. E.g., `\sum` can be used to check whether there is an equal amount of positive and negative numbers in an array;

```
//@ ensures \result <==> (\sum int i; 0 < i && i < a.length; a[i] > 0 ? 1
    : 0) == (\sum int i; 0 < i && i < a.length; a[i] < 0 ? 1 : 0);
public boolean equalPosNegAmount(int[] a) { ...
```

Listing 1.4: Example usage of `\sum`

Lastly, `\min` and `\max` work similar to `\sum` and could, e.g., be used for finding the lowest and highest value in an array. `\exists` and `\forall` both have a boolean result as opposed to `\sum`, `\min`, and `\max` which produce an integer. As the attentive reader might have seen, an additional logical operator was introduced, namely logical equivalence `<==>`. Together with implication `==>` they complement the standard logical operators from Java. JML does not differentiate between the use of `|` and `||`, or `&` and `&&`. In Java `|` and `&` are bitwise operators and have different semantics. For clarity this thesis uses the `||` and `&&` variants.

Specification declarations

Specifications seen so far, are so called lightweight specifications, i.e., they do not contain any of the following keywords `behavior`, `normal_behavior` or `exceptional_behavior` which are heavyweight specifications. With `normal_behavior` and `exceptional_behavior` one can specify that under certain conditions a method will always terminate without an exception or will always terminate with some specific exception under some condition, respectively. With `behavior`, normal and exceptional behavior can be combined.

Heavyweight specifications tell JML that the method specification is intended to be complete, as opposed to lightweight specifications which tell JML that the specification is incomplete and only contains some of what the specifier had in mind. When one uses lightweight specifications omitting the clauses `requires`, `ensures` or `signals` for a method results in the default specification of `\not_specified`. The meaning of `\not_specified` may vary between different usage of JML specifications, i.e., it is possible that one static checker[1] translates requires `\not_specified` to `requires true`

[1]Static checking will be described in section 1.3.2

and another to `requires false`.

Exceptions

The keyword `signals` can be used in conjunction with an exception type and an expression. When a particular exception – the one between parentheses – is thrown, the condition expressed by the expression should hold. An example is given for an `ArithmeticException` for a method that calculates a division. If the method throws the exception it must be the case that `b == 0`;

```
1 //@ ensures \result == a/b;
2 //@ signals_only ArithmeticException;
3 //@ signals (ArithmeticException e) b == 0;
4 public float divideBy(int a, int b) {
5   return a/b;
6 }
```

Listing 1.5: Example usage of `signals`

For lightweight specifications the signals clause defaults to `\not_specified` and the heavyweight specification to `(Exception) true`, i.e., it is always possible that there will be an exception. The `signals_only` keyword is used to indicate which exceptions may occur during execution of the method. When omitted, the specification defaults to the exceptions given by the `throws` clause, for both light- and heavyweight specifications. For heavyweight specifications the `signals` and `signals_only` clauses only apply for `behavior` and `exceptional_behavior`.

Behaviors of a method can be combined in different ways, namely specifying behavior of the method with `behavior`, which entails both exceptional and normal behavior, or separating and combining them with the keyword `also`. The difference is illustrated in Listing 1.6 and Listing 1.7;

```
1 /*@
2   @ public normal_behavior
3   @   requires b != 0;
4   @   ensures \result == a/b;
5   @ also
6   @ public exceptional_behavior
7   @   signals_only ArithmeticException;
8   @   signals (ArithmeticException e) b == 0;
9   @*/
```

```
10 public float divideBy(int a, int b) {
11   return a/b;
12 }
```

Listing 1.6: Example usage of combining normal with exceptional behavior

The same can be specified using only `behavior`;

```
1 /*@
2   @ public behavior
3   @   ensures \result == a/b;
4   @   signals_only ArithmeticException;
5   @   signals (ArithmeticException e) b == 0;
6   @*/
7 public float divideBy(int a, int b) {
8   return a/b;
9 }
```

Listing 1.7: Example usage of `behavior`

The difference is that with `behavior`, when the `requires` clause holds, the method can either end normally or exceptionally. When the method ends exceptionally, it should be the case that `b == 0`.

Note that, in general it is possible to transform preconditions to postconditions. In Listing 1.6 the precondition of the normal behavior can be moved to the postcondition by changing the postcondition to `(b != 0) ==> \result == a/b`. This thesis attempts to keep pre- and postconditions separated, and only uses the later if it improves readability.

Specifications for constructors

Constructors are somewhat different from regular methods in that they do not have a pre-state, i.e., the object does not yet exist. That is why a precondition of a constructor can only put restrictions on the arguments of the constructor. The postcondition of a constructor will typically relate the object state to the constructor's parameters.

```
1 //@ requires age > 0 && age < 67;
2 //@ requires salary > 0 && salary <= maxYearSalary;
3 //@ ensures getAge() == age;
4 //@ ensures getSalary() == salary;
5 CEmployee(int age, double salary) {
6   this.age = age;
7   this.salary = salary;
```

```
8 }
```

Listing 1.8: A constructor for the class `CEmployee`

For example, Listing 1.8 shows a possible constructor specification for the class `CEmployee` that implements the aforementioned interface `Employee`. The preconditions only specify restrictions on the arguments `age` and `salary` of the constructor method.

1.2.2. Class specifications

The specification of the interface `Employee` above makes an implicit assumption about the property of `getSalary` that should hold throughout, namely the salary should always lay within the range of zero and `maxYearSalary`. Any method in `Employee`, or implementation of `Employee` might potentially break this property when it is not explicitly mentioned in the method contracts. This means every method should add a `requires` and `ensures` clause like;

```
//@ requires getSalary > 0 && getSalary() <= maxYearSalary;
//@ ensures getSalary > 0 && getSalary() <= maxYearSalary;
public void someMethod(..) { ..
```

Listing 1.9: The burden of repeating yourself

To overcome the burden that specifications could get very large this way, and make it possible to describe additional properties over the lifetime of an object, JML provides class-level specifications. These class-level specifications, such as invariants, constraints and initially clauses specify properties over the objects internal state and describe the object's restrictions over time. Listing 1.9 shows an example where the pre- and post-condition of the method could be replaced with an invariant.

Invariants

An object invariant is a predicate that specifies a condition that should hold on all visible states of the object. Visible states are all states in which either a method call to the object starts or terminates. Object invariants can be used to remove the overhead of adding `requires` and `ensures` clauses for each method in a class, as they are implicitly added to the method contracts. Constructors are a little different in that they only need to ensure that the invariant is established in the post-state of the method. Invariants

have the neat feature that one does not need to write the same pre- and postconditions for every method. Invariants also contribute to a nice separation of concerns, i.e., invariants are inherited by subclasses. This way any method that overrides a method from a superclass, or methods added to a subclass, also needs to respect the invariant. An example invariant for the `Employee` interface would be the following;

```
//@ invariant getSalary() > 0 && getSalary() <= maxYearSalary;
```

Listing 1.10: Example usage of `invariant`

One exception where invariants do not need to hold is for so-called helper methods or helper constructors, which are private methods that aid methods that can be called by the programmer. These methods are annotated with the JML keyword `helper`.

Initially clauses

Initially clauses are like object invariants, only instead of specifying properties about every state, initially clauses specify what should hold in a state after creation of an object. Each non-helper constructor of an object has to establish the predicate specified by the initially clause. Like invariants, initially clauses can also be specified differently, namely by adding the wanted conditions to every postcondition of all the non-helper constructors. Using initially clauses will ensure that also subclasses, and any additional constructors specified in subclasses respect the initially clause.

Constraints

Constraints limit changes to an object. A constraint for the `Employee` interface could be that an employee is only allowed to increase in age. This could be specified like;

```
//@ constraint \old(getAge()) < getAge();
```

Listing 1.11: Example usage of `constraint`

However, this specification would be too strict. It should be possible to respect any constraint without actually changing the object's state. In particular, this means that also any pure method should be able to adhere to the specification. Therefore, the specification above should be changed to;

```
//@ constraint \old(getAge()) <= getAge();
```

Listing 1.12: Better usage of `constraint`

This way the method `getSalary` also adheres to the specification. Obviously the method `getSalary` should not change the age of an employee.

Variable declarations

Until this point, specifications did not specify anything about the values of an object's instance variables. Usually, these are declared private, and private elements cannot be accessed within the specifications. For specifications we need instance variables to be either public or protected. Whenever it is not possible or inconvenient to specify methods with pure get-methods, JML provides the option to make instance variables `spec_public` or `spec_protected`. This way, instance variable names can be utilized – without the need of pure-get methods to address them – by specifying the visibility of the instance variables for specifications.

Just like reference values within a method contract, fields can be specified with `non_null` or `nullable`. When omitted, fields are declared `non_null` implicitly.

Model and ghost variables

Model and ghost variables are specification-only variables, and do not occur during execution of a program. Model variables provide an abstract representation of an object's state. If the underlying state of a model variable changes, implicitly the model variable also changes. This relationship is often captured with an explicit translation. Specifications for model variables are split into two parts, specifying the type of the model field and specifying what it represents. Listing 1.13 shows an example of a model field `isSquare` that represents whether or not the rectangle is a square by comparing `length` and `width` of `Rectangle`.

```
1  public class Rectangle {
2
3    public int length;
4    public int width;
5
6    //@ model private boolean isSquare;
7    //@ represents isSquare = length == width;
8
9    ...
10 }
```

Listing 1.13: Example usage of a model field

Ghost variables extend the state by providing additional information that cannot be directly related to the state of the object. Ghost variables are often used to keep track of events that have happened on an object, e.g., which methods have been invoked, and how often. An example is given in Listing 1.14, where the ghost variable `countA`, counts the invocations of `methodA`. The corresponding JML construct `set`, can update ghost variables.

```
1 //@ ghost public int countA
2 //@ initially countA == 0;
3
4 public void methodA(..) {
5   //@ set countA = countA + 1;
6   ..
7 }
```

Listing 1.14: Example usage of a ghost field

Inheritance of specifications

In JML subclasses inherit class-level specifications, e.g., invariants, initially clauses and constraints. Method specifications are also inherited, which means that every class that implements an interface or extends another class has to respectively respect the interface or its superclass. Any additional specification made in a subclass or implementing class is implicitly combined (with `also`) with its inherited specifications.

1.2.3. Further reading

At this point the reader should be familiar with the basis of JML. However, as mentioned earlier there is a lot more to say about JML, other constructs will be explained when needed for specifications, or for better understanding of this thesis. When one wants to know more about JML, the reader is advised to take a look at the JML Reference Manual [24] or one of the following papers [23, 22, 35].

1.3. JML and verification

Chalin et al. [8] mention two approaches to verification, namely runtime assertion checking and static verification. Static verification can be further classified into static checking

and static verification.

1.3.1. Runtime assertion checking

With runtime assertion checking source code is checked during program execution, violations noticed by the checker are reported back to the user. A fundamental problem with runtime assertion checking is that it cannot be used for all applications – as the program actually needs to be executed to get feedback – whereby checking, e.g., a driverless car would be problematic. Although common programming mistakes can be found easily with this technique, code coverage is limited. To be sure that the program behaves correctly for all its executions, the programmer still needs to deal with practically an infinite amount of test cases for substantial programs. The main runtime assertion checking tool for JML is *jmlc* [10]. *JMLUnit* [21] can be used to generate JUnit tests automatically for JML annotated Java code. *JMLUnitNG* [43] generates TestNG tests automatically. JMLUnitNG has also been substantially improved over JMLUnit in terms of supported features (e.g., data generators) and performance.

1.3.2. Static checking

To reason about programs – without the need of executing them – program logics have been developed. Floyd was the first to introduce the concept of pre- and postconditions to reason about program logic in such a non-executional way. In 1969, this led Hoare to come up with a set of rules to reason about programs [15]. These rules, and variations of these rules, are often called Hoare logic. Static checking, which is one step further than assertion checking, makes use of this technique. A JML annotated Java program is compiled and checked for correctness using an automated theorem prover.

In Java the most popular static checking tool is *ESC/Java2* which stands for Extended Static Checking tool for Java [19]. The tool aids in providing the programmer with feedback about runtime exceptions and violations that are likely to occur.

Loop invariants

Loop invariants are used to guide a static checker or verifier when checking correctness of a method. Listing 1.15 shows an example of a Java method `contains` annotated with JML, that searches for a given `int` and returns `true` when the array contains the `int`.

```
1  /*@ requires a != null;
2    @ ensures \result ==
3    (\exists int i; 0 <= i && i < a.length; a[i] == val);
4    @*/
5  public boolean contains(int[] a, int val) {
6    boolean found = false;
7    int i = 0;
8    /*@ loop_invariant found ==
9      @   (\exists int j; 0 <= j && j < i; a[j] == val);
10     @ loop_invariant 0 <= i && i <= a.length;
11     @ loop_invariant a != null;
12     @*/
13   while(i < a.length && !found) {
14     if(a[i] == val) found = true;
15     i++;
16   }
17   return found;
18 }
```

Listing 1.15: Example of JML loop invariants

Loop invariants are predicates that should be preserved by every iteration of the loop. Loop invariants are needed to abstract from the loop, like method specifications do for the method body. Tools like ESC/Java2 can find loop invariants automatically when they are simple, but most of the time the user should specify them as tools are not able to find them.

1.3.3. Static verification

To actually give guarantees that code is correct, another category of tools is needed, i.e., verification tools. With enough specifications attached, correctness of a program could be formally verified, i.e., proven that the source code complies to the formal specification. A few program verification tools that support JML are *KeY* [5], *JACK* [9] and *Jive* [29]. However, proving correctness might not always be feasible due to incomplete tool support. For example Java generics are still poorly covered in almost any verification tool, while Java already supports generics since 2004 with Java 5. To cope with this fact, generics are stripped out and wherever possible, specifications are added to describe restrictions on these types. The specifications discussed later on in this thesis are verified with the static verifier KeY. KeY is chosen here since it is a standalone

prover for Java which has some additions to JML to make modular verification easier. The tool can handle most of Java 1.4. Besides that KeY can cope with additional JML constructs, KeY is also actively developed. Furthermore, since this thesis is supervised by one of the developers of the tool, KeY is an obvious choice.

A few constructs that KeY can cope with besides the standard JML covered so far are explained below: dynamic frames, abstract data types, model fields and an additional form of purity that is more strict. Prior to that, data groups are explained, which is JML's default way of specifying frames over non-static heap locations. Although dynamic frames, data abstraction and purity are terms not strictly bound to KeY, KeY-specific implementations are explained below. The extended version of JML that KeY uses is called JML*.

Data groups

For modular static verification, where individual program parts are checked for correctness, i.e., without considering the program as a whole, demands on specifications as well as the specification languages are higher than for example for runtime checking. One important aspect when modular static verification is done, is specifying the memory frame operated on. Frame is the part of a state operated on when executing a program.

One way to specify framing is by using data groups. Data groups can be used in JML's `assignable` clauses to state which part of the heap is affected by a method. When model fields are being used in JML for `assignable` clauses, they are used as data groups [38], references to a set of memory locations. In JML, model fields can be used to abstractly represent data which will be evaluated to a value, but at the same time also represent data groups which will be evaluated to a set of locations. Data group interpretations for model fields are defined by declaring locations to be part of a data group with the keyword `in`. The `in` annotation must be placed directly after the declaration of the field to be added. It is called static inclusion when a field of object `x` becomes part of a data group for object `x`. However, when fields of another object become part of a data group, this is dynamic inclusion, which can be accomplished with the keywords `maps` and `\into`. An example of both, static and dynamic inclusion, is given in Listing 1.16.

```
1 public interface List {
2   //@ public model instance JMLDataGroup footprint;
3   ...
```

```
4  }
5
6  public class ArrayList implements List {
7    private /*@ nullable @*/ Object[] array = new Object[10]; //@ in
         footprint;
8    //@ maps array[*] \into footprint;
9    ...
10 }
```

Listing 1.16: Data group example

The `ArrayList` implementation has a model field `footprint` used as a data group, declared in `List`. `ArrayList` does a static inclusion for the object array and a dynamic inclusion for the elements in array. The dynamic inclusion might have been placed at another place in the code, and does not necessarily have to come after the object from which fields will be included. The type `JMLDataGroup` object is part of JML's model library. The JML model class library is a result of the goal to stay as close to Java as possible. Therefore, e.g., mathematical notions or data groups are not introduced in the language itself as additional primitive types, but come with a library of so-called model classes. In this library, mathemathical concepts are sneaked in by modelling them as regular Java classes.

One of the major shortfalls using data groups is that most tool support for static/runtime checking is minimal. Those tools that do support data groups most of the time only support static inclusion. Dynamic inclusion is only formalised in *Coq* [25] at the moment. Furthermore, semantics of assignable classes differ greatly between different tools [25].

Dynamic frames

Kassios [18] proposes a solution for framing in the presence of data abstraction and calls it dynamic frames. With data abstraction, internal structure of program data is hidden by using getter methods and abstract data types, e.g., sequences that represent the actual data. Using dynamic frames it is possible to specify which set of memory locations is accessed or changed when executing a method. Furthermore, dynamic frames state that executing a method on one object does not necessarily change the state of another object.

In JML* a dynamic frame only represents a set of memory locations. Dynamic frames are called dynamic in the sense that they can evolve over time and simplify inheritance

of specifications [42]. An example of a simplification one may get, is when one provides a footprint for a `Collection`, which can then be used for implementing classes. For example, an `ArrayList` or a `LinkedList` can now use this footprint to specify by which locations they are framed. An implementation of an `ArrayList` will specify this footprint as its array, all locations in that array – `array[*]` – and the size of the list, as opposed to a `LinkedList` which will have a footprint containing all nodes and the size. Here the footprint can be specified as `size` in `AbstractCollection`, whereupon `LinkedList` and `ArrayList` get a revised specification.

The operations set membership, set union and set intersection are all defined for dynamic frames. Furthermore Kassios also describes a preservation operation, a modification operation and a swinging pivot requirement. The preservation operator, indicated with Ξf holds true if no execution changes frame f. Δf, the modification operator holds true when the execution only changes frame f. The swinging pivot requirement Λf is satisfied when frame f did not increase in any other way than allocation of new memory. With frame f, and m a specification variable – which could be a model field for instance, f frames m means that when the values in f does not change, then also m does not change. When f frames itself, it means that when no values in f change, f itself does not change either. Dynamic frames, as described here, can be seen as a proper implementation of data groups with sound logical theories.

Using dynamic frames with JML* has the benefit that location sets and model fields can be decoupled. This way data groups and data groups inclusions are not needed for specification. Data groups are used in JML to accomplish similar specifications but have a few shortcomings compared to dynamic frames. One of these shortcomings is that dynamic inclusions complicate modular reasoning about data groups significantly. Without using additional measures, it is not possible to determine locally whether a given location may be part of a given data group, as an applicable dynamic inclusion might occur in any subclass of the class or interface that declared the model field [42]. Another great advantage of dynamic frames is that it is already supported by KeY, which makes it possible to statically check programs annotated with dynamic frames.

JML* introduces additional specification operators and a primitive type called `\locset`. With `\locset` a set of memory locations can be specified. Dynamic frames in JML* are instances of model and ghost fields with type `\locset`. `\singleton(o.f)` holds the singleton set of the (ghost) field `f` of object `o`. `\subset(s1, s2)`, `\intersect(s1, s2)`, `\set_minus(s1, s2)`, `\set_union(s1, s2)` and `\disjoint(s1, s2)` can all be

JML*	mathematical meaning
`\subset(s1, s2)`	$s1 \subseteq s2$
`\disjoint(s1, s2)`	$s1 \cap s2 = \emptyset$
`\intersect(s1, s2)`	$s1 \cap s2$
`\set_minus(s1, s2)`	$s1 \setminus s2$
`\set_union(s1, s2)`	$s1 \cup s2$

Table 1.1.: Mathematical meaning

used in JML* and have the mathematical meaning shown in Table 1.1, where `s1` and `s2` represent location sets. The expressions `\subset(s1, s2)` and `\disjoint(s1, s2)` are boolean expressions. `\intersect(s1, s2)`, `\set_minus(s1, s2)` and `\set_union(s1, s2)` result in a new set representation.

Below one can find an example that uses dynamic frames for the interface of a `Coordinate` and implementation thereof. Afterwards a few additional specification constructs used in the example will be explained.

```
1  interface Coordinate {
2    //@ public model int h;
3    //@ public model int v;
4    //@ public model \locset footprint;
5    //@ public accessible h: footprint;
6    //@ public accessible v: footprint;
7    //@ public accessible footprint: footprint;
8
9    //@ assignable footprint;
10   //@ ensures h == hor;
11   //@ ensures v == ver;
12   //@ ensures \new_elems_fresh(footprint);
13   void setCoordinate(int hor, int ver);
14
15   //@ accessible footprint;
16   //@ ensures \result == h;
17   int /*@ pure @*/ getX();
18
19   //@ accessible footprint;
20   //@ ensures \result == v;
21   int /*@ pure @*/ getY();
22 }
23
24 class CoordImpl implements Coordinate {
```

```
25   private int x; //@ represents h = x;
26   private int y; //@ represents v = y;
27   //@ represents footprint = x, y;
28
29   //@ ensures \fresh(footprint);
30   public /*@ pure @*/ CoordImpl() { .. }
31
32   ..
33 }
```

Listing 1.17: Example that makes use of dynamic frames

In `Coordinate` first some model fields are declared, a frame is created, the model fields `v` and `h` get framed by the footprint and thereafter the footprint gets framed by itself. Framing in JML* is done with `accessible m:f` and has the meaning f frames m. The `setCoordinate` method indicates the method makes changes to the footprint with the `assignable` clause. `\new_elems_fresh` states that all members of footprint belong to freshly created objects, or were already part of the footprint, which is the case here as `h` and `v` were already part of the footprint. The construct `\new_elems_fresh` represents the swinging pivot requirement. The `accessible` clause in `getX` and `getY` tells that these methods read from the set of locations within `footprint`.

The implementation `CoordImpl` first defines represents clauses for `h` and `v`. Next the footprint gets specified as holding both the locations of `x` and `y`. The ensures for the constructor introduces another operator, `\fresh(footprint)` which tells locations in footprint were not allocated before the constructor call.

Assume `CoordImpl` is a straightforward implementation of `Coordinate`. A program that makes use of `CoordImpl` will benefit from this specification when verified. Consider the following main method;

```
1 public static void main(String[] args) {
2   Coordinate c1 = new CoordImpl();
3   Coordinate c2 = new CoordImpl();
4   c1.setCoordinate(1, 2);
5   c2.setCoordinate(4, 5);
6   //@ assert c1.getX() == 1;
7 }
```

Listing 1.18: Example program making use of dynamic frames

When this program is being verified, the verifier can see that `c1` and `c2` have distinct footprints. Furthermore, when the method `setCoordinate` is called on `c1` and `c2` the

verifier sees that respectively the footprint of `c1` and `c2` will be changed and the assertion `c1.getX()==1` passes.

An example where dynamic frames are very useful is for the method `retainAll(Collection c)` in the interface `Collection` (Section 2.1.13). This method removes all the elements from the collection operated on that are not in `c`. This means that, if `c` equals the collection operated on, it should return the collection before the call, and if that is not the case elements in `c` that are not in the collection itself should be removed. Like the informal behavior, also the formal specifications can distinguish between the case where `c` is the collection itself or not – with two `normal_behavior` specifications. The distinguishing is done with JML's `requires` clause.

```
// @   requires this == c || footprint == c.footprint;

// @   requires \disjoint(footprint, c.footprint);
```

Listing 1.19: Distinguishing `requires` clauses

Listing 1.19 shows the two different `requires` clauses. The first `requires` clause describes the case where the footprints are the same, hence the collections are equal. The second clause does the opposite and state that the collections should be disjoint.

Abstact data types

Sequences are an example of abstract data types, which are used to abstract from an implementation. This is accomplished by specifying the structure of an object as a known mathematical structure. For example, sequences can be used to capture the structure of a collection or a hierarchical structure like trees. Sequences, like in mathematics, are ordered lists of objects and can be used as JML model or ghost fields for KeY.

KeY supports the `\seq` data type with the operations `\seq_empty`, `\seq_singleton(obj)`, `\seq_concat(s1, s2)`, `\seq_sub(seq, from, to)` , `\seq_reverse(s1)` and `\indexOf(s1, obj)`, where `s1` and `s2` are sequences. Also, `seq[x]` can be used to return the value of the object at index `x` and `seq.length` to get the length of the sequence. The construct `\seq_empty` can be used to indicate an empty sequence. With `\seq_singleton(obj)` one can indicate a single object `obj` as sequence with one element. To concatenate two sequences `\seq_concat(s1, s2)` can be used, with two sequences that should be concatenated. The construct `\seq_sub(seq, from, to)` can be used like substring works on Java String objects, to get only that part of a sequence, where `from` and `to` are included. The construct `\seq_reverse(s1)` will result in the reversal of a sequence `s1`.

```
1  public final class Tree {
2    int value;
3    /*@ nullable @*/ Tree left;
4    /*@ nullable @*/ Tree right;
5
6    /*@ invariant left == null <==> right == null;
7      @ invariant left != null ==> (left.\inv && right.\inv);
8      @
9      @ ghost int height;
10     @ invariant height >= 0;
11     @ invariant left != null ==>
12     @   height > left.height && height > right.height;
13     @
14     @ ghost \seq values;
15     @ invariant values == \seq_concat(\seq_singleton(value), (left==null)
16         ? \seq_empty : \seq_concat(left.values, right.values));
17     @*/
18
19   /*@ normal_behavior
20     @  ensures (\forall int z;
21           \indexOf(values, z) != -1; z <= \result);
22     @  ensures \indexOf(values, \result) != -1;
23     @  measured_by height;
24     @  strictly_pure
25     @*/
26   int max () {
27     int res = value;
28     if (left != null) {
29       res = maxHelper(res, left.max(), right.max());
30     }
31     return res;
32   }
33
34   /*@ normal_behavior
35     @ ensures \result >= x;
36     @ ensures \result >= y;
37     @ ensures \result >= z;
38     @ ensures \result == x
39         || \result == y
40         || \result == z;
41     @ strictly_pure helper
42     @*/
```

```
43   int maxHelper(int x, int y, int z) {
44     if (x > y)
45       return (x > z ? x : z);
46     else
47       return (y > z ? y : z);
48     }
49   ...
50 }
```

Listing 1.20: Example usage of sequences

The example class `Tree` in Listing 1.20 illustrates a scenario where sequences can be used. The sequence `values` contains every `value` of all its subtrees, expressed by the invariant for `values`. The postconditions for the method `max()` expresses that the return value should be contained within the sequence `values`, and every `value` in `values` should be less than or equal to the return value of the method. As the method `max()` makes recursive calls, there is an additional `measured_by` clause, which states that every call to the function, the value `height` should strictly decrease each time the method is invoked by itself. Constructors and methods that modify the `Tree` should contain the specification operation `set` to update the representation – `values` – accordingly.

Besides sequences, KeY has also the option to extend the tool with other constructs. Rules can be added to the language to also support other mathematical properties. For instance KeY can be extended to have an understanding of sets or maps. Version 2.0.0 of KeY actually comes with constructs for sets built in, however, these are not yet officially part of the constructs that KeY supports. These constructs start with `\dl_` and may not be part of future versions of KeY. The `\dl` stands for dynamic logic, more about dynamic logic can be found at Section 3.1.2.

The `Tree` class above could be easily modified to use the set constructs, however, then no duplicates elements are allowed. In Table 1.2 the replacements needed in comparison to using sequences are emphasized.

Sequences are more appropriate when indices are needed, whereas sets might be used when one demands more abstraction. With sets it is not possible to have duplicates, whereas with sequences this is no problem.

sequence representation	set representation
`ghost \seq values`	`ghost \set values`
`\seq_singleton(Object)`	`\dl_single(Object)`
`\seq_empty`	`\dl_emptySet()`
`\seq_concat(s1, s2)`	`\dl_cup(s1, s2)`
`\indexOf(s, Object) != -1`	`\dl_contains(s, Object)`

Table 1.2.: Comparison between sequence and set representation

Model methods

Although not completely supported, KeY allows the use of model methods for specification. Model methods are essentially KeY's way of declaring and using abstract predicates. For this thesis it is enough to know that model methods are yet another specification constructs, that allows for further abstraction. Model methods – like regular methods – can have arguments but, can only be used for specification and do not change heap locations. Since model methods are in an experimental state they have not been used for specification and verification in this thesis.

Purity

Besides the purity modifier in standard JML, KeY introduces additional notations to specify a different kind of purity. As opposed to the modifier pure in standard JML, which allows heap modifications by creating fresh objects on the heap and to assign to these fresh locations, KeY comes with representations for so-called strictly pure methods, i.e., where methods do not modify the heap at all. In Listing 1.21 three different ways of specifying strictly purity are given;

```
1 /*@ assignable \strictly_nothing;
2   @*/
3 int strictlyPureMethod() { ... }
4
5 /*@ strictly_pure @*/ int anotherStrictlyPureMethod() { ... }
6
7 /*@ ensures \dl_heap() == \old(\dl_heap));
8   @*/
9 in finalStrictlyPureMethod() { ... }
```

Listing 1.21: Strict purity

The first method uses the new keyword `\strictly_nothing`, the second method uses a newly introduced modifier keyword `strictly_pure`, and finally, the last specification is done with an ensures clause which tells that the heap before and after invocation of the method should be the same. The three specifications are equivalent.

1.4. Discussion

Since unit testing is not enough for all projects, specification languages like JML arose to be precise about the behavior of a program. These specification languages are used together with several kind of tools, e.g., runtime checkers, static checkers and verification tools. Verification tools use data groups, dynamic frames and purity to allow one to specify the heap locations a program operates on. The downside of data groups is that they are minimally implemented. KeY uses dynamic frames – which can be seen as a proper implementation of data groups with sound logical theories – and strict purity to formally capture change of heap locations. Abstract data types can be used to make higher-level specifications, e.g., instead of specifying precise behavior, behavior is captured by an abstract representation. The next part of this thesis will use the specification constructs described in this chapter to formally specify behavior of parts of the Java Collections Framework.

Part II.

Contributions

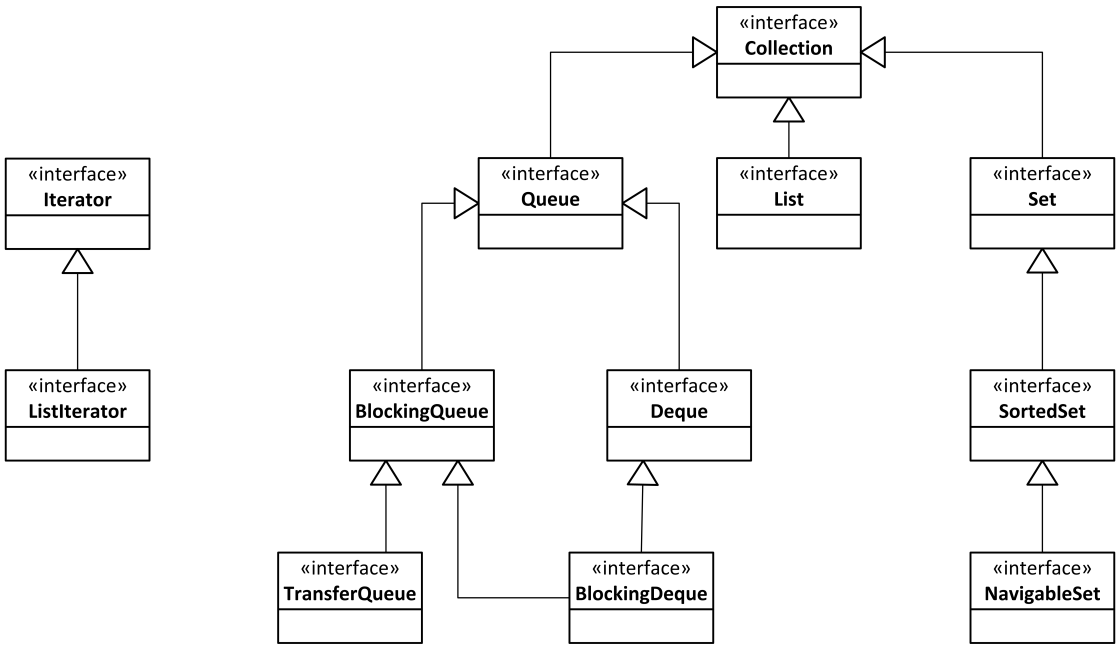
2. Specifications

This chapter describes specifications made for the interfaces `Collection`, `List`, `Iterator` and `ListIterator`. Section 2.5 concludes the chapter with findings made during the elaboration of aforementioned interfaces. Specifications described in this part are based on earlier findings of Peters's work [33] and try to improve wherever possible. Therefore, first Peter's work was stripped and data abstraction is used to represent the actual collection, instead of using a query method to get an array representation. Another difference is the fact that the specifications in this thesis use the annotation of `strictly_pure` whenever possible. Since KeY – the tool used for verification later on – does not support generics, these have been stripped out. Specifications are based on Java 7u6 Build b24[1] and the documentation located at http://docs.oracle.com/javase/7/docs/api/.

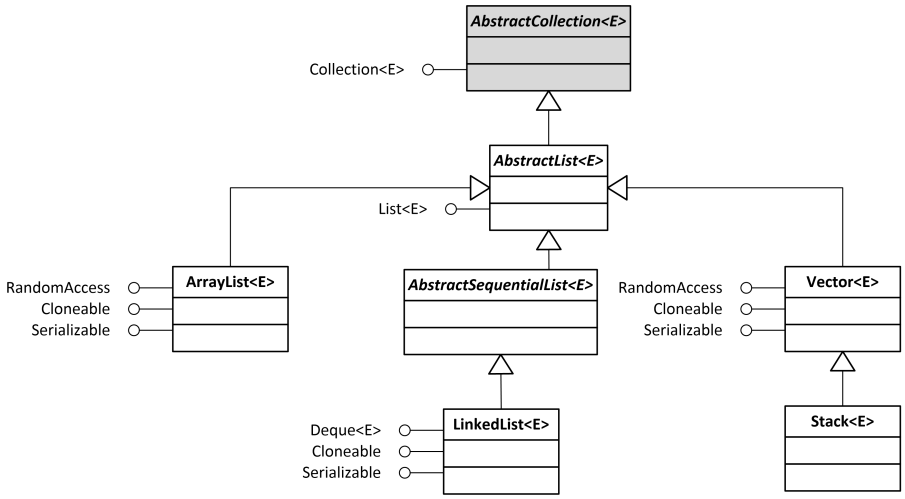
The interfaces `Collection`, `List`, `Iterator` and `ListIterator` all have a dedicated section describing their formal specifications. Each section starts with general specifications that state invariants, constraints and abstract representations of the corresponding interface. After the general specification, specifications about the methods the interface contains are described. Each section starts with an informal part describing behavior, followed by the formal specification using JML* and an explanation where necessary.

The interfaces described in the following chapters are chosen since they are the basic interfaces for the Java Collections Framework of which selected parts are verified. Figure 2.1a provides the interfaces for the collection classes. The left part of the figure shows the iterator interfaces, which can be returned by a collection to iterate over the collection. Figure 2.1b shows the collection classes based on the list part. The specifications described in this chapter are the latest version, as used for verification in Chapter 3. The arrows in the figures indicate that an interface extends from another interface.

[1] Publicly available at http://jdk7.java.net/source.html



(a) `Collection` interfaces



(b) `List` hierarchy

Figure 2.1.: Hierarchy of part of the Java Collections Framework

2.1. `Collection` specifications

This section describes the specifications for the interface `Collection` located at `java.util.Collection`. First class-level specifications about `Collection` will be explained, afterwhich the remainder of the section will explain the specification of methods from the `Collection` interface.

2.1.1. Class-level specifications

From the informal description it can be seen that the `Collection` interface is the root interface for the collection hierarchy. A collection represents a group of objects, known as its elements. Possible restrictions on collection implementations are whether or not it is allowed to have duplicate elements, null elements and whether or not elements should be ordered within the collection.

Methods that modify a collection are specified to throw an `UnsupportedOperationException` when the collection operated on does not provide support for the given operation. However, it is not required that they throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. Additionally a `NullPointerException` or `ClassCastException` could be thrown whenever there is an attempt to add an ineligible element to the collection. This is the case when the collection does not allow null elements and there is an attempt to add `null` to the collection, or when there is an attempt to add an object that is not a (sub)type of the type which the collection contains.

Several formal specifications can be deduced from these informal specifications, which are provided in Listing 2.1 and are described in the remainder of this section.

```
1  //@ public instance model boolean addSupported;
2  //@ public accessible addSupported: \nothing;
3  //@ public instance model boolean removeSupported;
4  //@ public accessible removeSupported: \nothing;
5  //@ public instance model boolean clearSupported;
6  //@ public accessible clearSupported: \nothing;
7
8  /*@ public model instance \locset footprint;
9    @ public accessible \inv: footprint;
10   @ public accessible footprint: footprint;
11   @
```

```
12  @ public model instance boolean supportNullElements;
13  @ public model instance boolean supportDuplicates;
14  @ public model instance boolean elementsHaveOrder;
15  @ public accessible supportNullElements: \nothing;
16  @ public accessible supportDuplicates: \nothing;
17  @ public accessible elementsHaveOrder: \nothing;
18  @
19  @ public nullable ghost instance \seq repr;
20  @
21  @ public model instance int seqLength;
22  @ public accessible seqLength: footprint;
23  @ public instance invariant seqLength >= 0;
24  @ public instance invariant seqLength == repr.length;
25  @
26  @ public model instance boolean sorted;
27  @ public accessible sorted: footprint;
28  @ public represents sorted = true;
29  @ public instance invariant elementsHaveOrder ==> sorted;
30  @
31  @ public model instance boolean noNullElements;
32  @ public accessible noNullElements: footprint;
33  @ public represents noNullElements =
34      (\forall int i; 0 <= i && i < seqLength; repr[i] != null);
35  @ public instance invariant
36      !supportNullElements ==> noNullElements;
37  @
38  @ public instance invariant !supportDuplicates ==> (
39      \forall int i; 0 <= i && i < size();
40      \indexOf(\seq_concat(\seq_sub(repr, 0, i),
41        \seq_sub(repr, i+1, size())), repr[i]) == -1);
42  @*/
```

Listing 2.1: Class-level specifications for the interface `Collection`

The model field `footprint` defines a dynamic frame. The boolean model fields are not framed, `footprint` and `\inv` – which is a model field that represents all invariants – are framed by `footprint`. Several boolean properties are specified about `Collection`. A model field `seq` is used to represent the collection and `seqLength` indicates the size of the collection. A boolean flag `sorted` can be used to represent whether all elements are ordered, normally this would be done by specifying the ordering over elements in the collection with the method `compareTo` for comparable objects, however, as explained later on page 76 and on page 77 `compareTo` cannot be used as restrictions were needed

on objects. The invariant about `supportNullElements` states that it should not be the case that there exists a `null` value in the collection when `null` elements are not supported. The last invariant states that it should not be the case that there exists a second object equal to one of the other elements from the collection. This is defined as taking all elements in front and after the element checked for, combining them, and then check if the result still contains the element left out.

The model fields `addSupported`, `removeSupported` and `clearSupported` are there to be able to specify exceptional behavior in case an `UnsupportedOperationException` should be thrown. This is the same as Peters did in [33].

Alternative specification for supportDuplicates

Another way of specifying what should hold when duplicates are not allowed, is checking the number of times an element is contained within the collection, which should be equal to one, in case duplicates are not allowed. Listing 2.2 shows this alternative specification. The choice whether the alternative should be used at the end depends on which of the specifications makes verification easier. Both specifications need a little inspection in order to completely understand what they do.

```
@ invariant !supportDuplicates ==> (\forall int i;
    0 <= i && i < size(); (\sum int j; 0 <= j && j < seqLength;
    (repr[i] == repr[j] && i != j) ? 1 : 0) == 1);
```

Listing 2.2: Alternative invariant for supportDuplicates

2.1.2. method `size`

The method `size` only needs a small amount of specification. The size of a collection is specified as the length of the sequence, which abstractly represents the collection. As no changes are made to the collection the method is specified `pure`. The state of the colection only depends on `footprint`, hence a corresponding accessible clause. Listing 2.3 shows the specification of the method `size`.

```
1 /*@ public normal_behavior
2   @   accessible footprint;
3   @   ensures \result == seqLength;
4   @*/
```

```
5  /*@ pure @*/ int size();
```

Listing 2.3: Specification of the method `size`

2.1.3. method `isEmpty`

The method `isEmpty` also has a simple specification, i.e., when the size of the collection is `0` the collection is empty. Furthermore, the method also only accesses `footprint` and it can be specified `pure`. Listing 2.4 shows the specification of the method `isEmpty`.

```
1  /*@ public normal_behavior
2    @   accessible footprint;
3    @   ensures \result == (size() == 0);
4    @*/
5  /*@ pure @*/ boolean isEmpty();
```

Listing 2.4: Specification of the method `isEmpty`

2.1.4. method `contains(Object o)`

The method `contains` is less trivial. The method should return `true` if and only if the collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`. When the method throws a `ClassCastException` – the type of the specified element is incompatible with the collection – or a `NullPointerException` - when the specified element is `null` and the collection does not permit `null` elements.

From the informal specifications, the specification in Listing 2.5 is deduced.

```
1  /*@ public normal_behavior
2    @   requires \typeof(o) == \type(java.lang.Object);
3    @   ensures \result == (\exists int i; 0 <= i
4          && i < seqLength; repr[i] == o);
5    @   assignable \strictly_nothing;
6    @ also
7    @ public exceptional_behavior
8    @   requires \typeof(o) == \type(java.lang.Object);
9    @   requires !supportNullElements && o == null;
10   @   signals (NullPointerException)
11         !supportNullElements && o == null;
12   @   signals_only NullPointerException;
13   @   assignable \nothing;
```

```
14   @*/
15 boolean contains(/*@ nullable @*/ Object o);
```

Listing 2.5: Specification of the method `contains(Object o)`

The method `contains` returns `true` whenever there exists an element `o` in the collection. A `ClassCastException` or `NullPointerException` might be thrown by an implementing class, however, as explained later (on page 76) `ClassCastExceptions` are not considered for now. Furthermore, as these exceptions are optional, a design choice might be to not trigger a `NullPointerException` when `null` elements are not allowed, but instead just return `false`.

2.1.5. method `iterator`

The method `iterator` returns an `Iterator` over the elements in the collection. The informal specification only states that there are no guarantees concerning the order in which the elements are returned, unless the collection is an instance of some class that does provide a guarantee. Therefore specifications are kept to a minimum and only state that the method is `pure`, only depends on `footprint` and a number of properties about the iterator should be set. Explanations about the different properties can be found in Section 2.3, where specifications about `Iterator` are explained. The specifications for the method `iterator` can be found in Listing 2.6.

```
1 /*@ public normal_behavior
2   @   ensures \result.position == -1;
3   @   ensures \result.collection == this;
4   @   ensures \result.\inv;
5   @   ensures \fresh(\result);
6   @   assignable \nothing;
7   @*/
8 Iterator iterator();
```

Listing 2.6: Specification of the method `iterator`

2.1.6. method `toArray`

From the informal specification attached to the API the method `toArray` the following specifications are derived. The method should return an array containing all the elements in the collection. When the collection makes any guarantees about ordering, the returned

array should adhere to this ordering. Furthermore the returned array will be a freshly allocated array, any changes made to the array will not result in changes to the collection.

```
1 /*@ public normal_behavior
2   @   ensures \typeof(\result) == \type(java.lang.Object[]);
3   @   ensures \result != null;
4   @   ensures \result.length == size();
5   @   ensures (\forall int i; 0 <= i && i < seqLength;
6   @         \result[i] == repr[i]);
7   @   ensures \fresh(\result);
8   @*/
9 /*@ pure nullable @*/ Object[] toArray();
```

Listing 2.7: Specification of the method `toArray`

Listing 2.7 shows the specification for the method `toArray`. The method `toArray` is `pure` and should return a newly created array. The length of the resulting array should be equal to the size of the collection and all elements should be part of the result. Since no verification is performed on methods with ordering, this is also not considered here. When ordering will be introduced later the specification should somehow differentiate on ordered collections and collections that are not ordered.

2.1.7. method `toArray(Object[] a)`

The method `toArray(Object[] a)` is very similar as the method `toArray` described above. In particular the calls to `toArray(new Object[0])` and `toArray()` are identical. `toArray(Object[] a)`, like `toArray`, returns an array containing all of the elements in the collection. Furthermore, the runtime type of the returned array is that of `a` (the object given as argument). When the collection fits in `a`, it is returned therein. Otherwise, a new array is allocated with the runtime type of `a` with the size of the collection. When the collection fits `a`, but `a` has more elements than the collection, the first element following the end of the collection will be set to `null`. Ordering guarantees should be handled in the same fashion as the method `toArray`. The method throws an `ArrayStoreException` - if the runtime of `a` is not a supertype of the runtime type of every element in the collection - or a `NullPointerException` when `a` is `null`.

```
1 /*@ public normal_behavior
2   @   requires \typeof(a) == \type(java.lang.Object[]);
3   @   requires a.length >= seqLength;
4   @   requires a != null;
```

```
5    @   ensures (\forall int j; 0 <= j && j < seqLength;
6          \result[j] == repr[j]);
7    @   ensures (a.length > seqLength) ==> a[seqLength] == null;
8    @   ensures \result == a;
9    @   assignable a[0 .. seqLength];
10   @ also
11   @ public normal_behavior
12   @   requires \typeof(a) == \type(java.lang.Object[]);
13   @   requires a.length < seqLength;
14   @   requires a != null;
15   @   ensures (\forall int j; 0 <= j && j < seqLength;
16         \result[j] == repr[j]);
17   @   ensures  \fresh(\result) && \result.length == seqLength;
18   @   ensures \typeof(\result) == \type(java.lang.Object[]);
19   @   assignable \nothing;
20   @ also
21   @ public exceptional_behavior
22   @   requires a == null;
23   @   signals (NullPointerException) a == null;
24   @   signals_only NullPointerException;
25   @   assignable \nothing;
26   @*/
27 /*@ nullable @*/ Object[] toArray( /*@ nullable @*/ Object[] a);
```

Listing 2.8: Specification of the method `toArray(Object[] a)`

The formal specification can be found in Listing 2.8. The normal behavior requires that the array is not `null`. The method is allowed to change every element in `a`. When the length of `a` is greater than or equal to `seqLength` – which represents the size of the collection – `a` is used as result of the method call. If `a` is larger than `seqLength` the additional `null` element is set. If `a` was too small to fit the collection, a freshly allocated array with length `seqLength` is used for the result.

The exceptional behavior specifies when a `NullPointerException` should be thrown. No changes to the heap should be made in case of an exception. Again, as described later (on page 76) there are limitations on what to express about objects, and therefore, `ArrayStoreException`s are not regarded for this method.

2.1.8. method `add(Object o)`

The method `add(Object o)` is an optional operation, if implemented, it ensures the collection contains `o` afterwards. The method returns `true` if the collection changed as a result of a call to `add(Object o)`. When a collection does not support duplicates and `o` is already a member of the collection the method returns `false`. Implementations that support the operation may place limitations on what elements may be added to the collection. If a collection refuses to add a particular element for any reason other than that it already contains the element, it must throw an exception rather than returning `false`. If this was not the case, the invariant that a collection always contains the specified element after `add(Object o)` returns becomes invalid.

The method specifies five possible exceptions. When the `add(Object o)` operation is not supported an `UnsupportedOperationException` should be thrown. If the class of `o` prevents it from being added to the collection a `ClassCastException` should be thrown. A `NullPointerException` should be thrown when the collection does not allow `null` elements and `o` is `null`. An `IllegalArgumentException` or `IllegalStateException` should be thrown when some property prevents `o` to be added to the collection, or when `o` cannot be added because of insertion restrictions at this time, respectively.

```
1  /*@ public normal_behavior
2    @   requires \typeof(o) == \type(java.lang.Object) && o.\inv;
3    @   requires !supportNullElements ==> (o != null);
4    @   requires !supportDuplicates ==> !contains(o);
5    @   ensures seqLength == \old(seqLength) + 1;
6    @   ensures (\forall int i; 0 <= i && i < seqLength-1;
7          repr[i] == \old(repr[i]));
8    @   ensures repr[seqLength-1] == o;
9    @   ensures \result;
10   @   assignable footprint;
11   @ also
12   @ public normal_behavior
13   @   requires \typeof(o) == \type(java.lang.Object) && o.\inv;
14   @   requires !supportNullElements ==> (o != null);
15   @   requires !supportDuplicates && contains(o);
16   @   ensures !\result;
17   @   assignable \nothing;
18   @ also
19   @ public exceptional_behavior
20   @   requires \typeof(o) == \type(java.lang.Object);
21   @   requires (!supportNullElements && o == null) || !addSupported;
```

```
22    @   signals (NullPointerException) !supportNullElements && o == null;
23    @   signals (UnsupportedOperationException) !addSupported;
24    @   signals_only NullPointerException, UnsupportedOperationException,
25          ClassCastException, IllegalArgumentException,
                IllegalStateException;
26    @   assignable footprint;
27    @*/
28  boolean add( /*@ nullable @*/ Object o);
```

Listing 2.9: Specification of the method `add`

Listing 2.9 gives a formal specification. The specification for the normal behavior should meet the requirements of `o` not being `null` – when `null` elements are not supported – and there should not be an element equal to `o` present in the collection when duplicates are not supported. The field `footprint` must be assignable as `o` needs to be added. The field `seqLength` should be incremented, the elements before the call should still be part of the collection after the call, and additionally `o` should also be a member of the collection afterwards. When the above requirements are met, it is still possible that the method throws an exception due to unknown implementation details. This is the case for `UnsupportedOperationException`, `IllegalArgumentException` and `IllegalStateException`.

When a `NullPointerException` is being thrown by this method it should be the case that `null` elements were not allowed and `o` equals `null`, and a `ClassCastException` is being thrown if `o` does not fit the collection – which is the case when `o` has a different type the collection expects, as stated earlier this is left out for now.

When the collection does not support duplicates and `o` is already contained within the collection this will likely result in an `IllegalArgumentException`, however, the API documentation leaves this open for interpretation.

2.1.9. method `remove(Object o)`

When implemented, the optional method `remove(Object o)` removes a single instance of the specified element from the collection if it is present. The method returns `true` if the collection is changed by the operation. The method can throw two optional exceptions, `ClassCastException` and `NullPointerException`. A `ClassCastException` should be thrown when `o` is incompatible with the collection and a `NullPointerException` when

`o` is `null` and the collection does not allow `null` elements. When the remove operation is not supported an `UnsupportedOperationException` should be thrown.

```
1  /*@ public normal_behavior
2    @   requires !supportNullElements ==> (o != null);
3    @   requires contains(o);
4    @   ensures repr == \seq_concat(
5          \seq_sub(\old(repr), 0, \indexOf(\old(repr), o)),
6          \seq_sub(\old(repr), \indexOf(\old(repr), o), \old(seqLength))
7        );
8    @   ensures seqLength == \old(seqLength) - 1;
9    @   assignable footprint;
10   @ also
11   @ public normal_behavior
12   @   requires !contains(o);
13   @   ensures !\result;
14   @   assignable \nothing;
15   @ also
16   @ public exceptional_behavior
17   @   requires (!supportNullElements && o == null) || !removeSupported;
18   @   signals (NullPointerException) !supportNullElements && o == null;
19   @   signals (UnsupportedOperationException) !removeSupported ==> true;
20   @   signals_only NullPointerException, UnsupportedOperationException;
21   @   assignable \nothing;
22   @*/
23 boolean remove( /*@ nullable @*/ Object o);
```

Listing 2.10: Specification of the method `remove(Object o)`

Listing 2.10 gives the formal specification. The first normal behavior specifies that when the requirements of not having a `null` element when `null` elements are not supported is met, and there exists an element `o` in the collection, this will result in a collection with one element less and a return value `true`. The new representation of the collection, is a sequence that exists of the part before and the part after the removed element (which might both have a length of zero).

An additional specification for normal behavior is specified, which requires that the same condition is met as before, but this time `o` should not be a member of the collection. The return value should be `false` in this case and no changes should be made to memory.

The method might throw a `NullPointerException` or `UnsupportedOperationException`. A `NullPointerException` might be thrown when `null` elements are not allowed and `o` equals `null`, however, this might also result in that the method returns `false`.

2.1.10. method `containsAll(Collection c)`

The method `containsAll(Collection c)` returns `true` if the collection contains all of the elements specified in `c`. Two optional exceptions might be thrown, `ClassCastException` and `NullPointerException`. As before these should be thrown when either one (or more) of the types of the elements in `c` is incompatible, or when `c` contains one or more `null` elements and the collection does not allow `null` elements. Additionally the `NullPointerException` should be thrown when `c` itself is `null`, which is not optional. This nails down to the formal specification in Listing 2.11.

```
1  /*@ public normal_behavior
2    @   requires c != null;
3    @   requires c.\inv && \subset(footprint, c.footprint) &&
4          \subset(c.footprint, footprint);
5    @   ensures \result == true;
6    @   accessible footprint, c.footprint;
7    @   assignable \strictly_nothing;
8    @ also
9    @ public normal_behavior
10   @   requires c != null;
11   @   requires c.\inv && \disjoint(footprint, c.footprint);
12   @   requires !supportNullElements ==> c.noNullElements;
13   @   ensures \result <==> (\forall int i; 0 <= i && i < c.seqLength;
14         (\exists int j; 0 <= j && j < seqLength;
15         (Object)repr[j] == (Object)c.repr[i]));
16   @   accessible footprint, c.footprint;
17   @   assignable \strictly_nothing;
18   @ also
19   @ public exceptional_behavior
20   @   signals (NullPointerException) c == null ||
21         c != null && (!supportNullElements ==> !c.noNullElements);
22   @   assignable \nothing;
23   @*/
24 boolean containsAll( /*@ nullable @*/ Collection c);
```

Listing 2.11: Specification of the method `cointainsAll(Collection c)`

The normal behavior is split into two parts, the first behavior specifies when `c` is the collection itself, and the second specification defines behavior when `c` and the collection are disjoint. The exception `UnsupportedOperationException` might be thrown.

Alternative specification

The ensures part in the first normal behavior specification could also be formulated making use of the method `contains(Object o)`. This is depicted in Listing 2.12.

```
@  ensures \result == (\forall int i; 0 <= i && i < c.seqLength;
     contains((Object)c.repr[i]));
```

Listing 2.12: Alternative specification using `contains`

Using this alternative makes an inner `exists` like in Listing 2.11 unnecessary, which might increase understandability.

2.1.11. method `addAll(Collection c)`

The method `addAll(Collection c)` is also an optional operation and adds all of the elements in `c` to the collection. The behavior of this operation is undefined when the collection is modified while the operation is in progress, which implies that adding `c` to the collection where `c` is the collections itself gives undefined behavior. When the collection changed as a result of the call it returns `true`. The method could throw an `UnsupportedOperationException`, `ClassCastException`, `NullPointerException`, `IllegalArgumentException` or an `IllegalStateException`. The `ClassCastException` should be thrown when the class of an element of `c` prevents it from being added to the collection. A `NullPointerException` should be thrown when either one of the elements of `c` is `null` and `null` elements are not allowed by the collection, or `c` is `null` itself. The `IllegalArgumentException` and `IllegalStateException` should be thrown when some property of an element of `c` prevents it from being added to the collection, or if one or more of the elements of `c` could not be added at this time due to insertion restrictions, respectively.

```
1  /*@ public normal_behavior
2    @   requires c.\inv && \disjoint(footprint, c.footprint);
3    @   requires c != null;
4    @   requires !supportNullElements ==> c.noNullElements;
5    @   requires addSupported;
6    @   ensures (\forall int i; 0 <= i && i < \old(seqLength);
7          repr[i] == \old(repr[i]));
8    @   ensures supportDuplicates ==> (\forall int j;
9          \old(seqLength) < j && j < \old(seqLength) + c.seqLength;
10         repr[j] == c.repr[j - \old(seqLength)]);
```

```
11    @   ensures seqLength == \old(seqLength) + c.seqLength;
12    @   ensures seqLength > \old(seqLength) <==> \result;
13    @   ensures \new_elems_fresh(footprint);
14    @   assignable footprint;
15    @ also
16    @ public exceptional_behavior
17    @   requires (!supportNullElements && !c.noNullElements) || !
          addSupported;
18    @   signals (NullPointerException) (!supportNullElements && !c.
          noNullElements);
19    @   signals (UnsupportedOperationException) !addSupported;
20    @   signals_only UnsupportedOperationException, ClassCastException,
21          IllegalArgumentException, IllegalStateException,
              NullPointerException;
22    @   assignable \nothing;
23    @*/
24  boolean addAll( /*@ nullable @*/ Collection c);
```

Listing 2.13: Specification of the method `addAll(Collection c)`

Listing 2.13 gives a formal specification. The normal behavior up to the `ensures` part is identical to what is specified in Section 2.1.10 for the method `containsAll(Collection c)`. The `ensures` part first states that every element in the old representation of the collection should still be in the collection after the operation is performed. Next, it states that also the elements from `c` should be in the updated collection, and the method should return `true` if and only if the representation after the operation is larger than that is was before the operation. Freshly allocated memory is used if the collection is updated. Under certain circumstances specific exceptions might be thrown by implementations of `Collection`.

The exceptional behavior consists of a `NullPointerException` that if being thrown, it should be case that either `c` is `null`, or one or more of `c`'s elements are `null`. Additionally, a `UnsupportedOperationException`, `IllegalArgumentException` or `IllegalStateExcption` might be thrown, which have no additional formal constraints at this point.

2.1.12. method `removeAll(Collection c)`

`removeAll(Collection c)`, an optional operation, removes all of the elements in the collection that are also contained in `c`. No elements that are in `c` will be in the collection

after the method returns. The method will return `true` if the collection has changed by the operation. An `UnsupportedOperationException`, `ClassCastException` and `NullPointerException` might be thrown by the method, whereby the second is optional. The `NullPointerException` might optionally be thrown when one of the elements of `c` is `null`, but should be thrown when `c` itself is `null`.

```
1  /*@ public normal_behavior
2    @   requires c.\inv && \disjoint(footprint, c.footprint);
3    @   requires c != null;
4    @   requires !supportNullElements ==> c.noNullElements;
5    @   ensures \result <==> \old(seqLength) > seqLength;
6    @   assignable footprint;
7    @ also
8    @ public normal_behavior
9    @   requires c.\inv && \disjoint(footprint, c.footprint);
10   @   requires c != null && !supportNullElements && !c.noNullElements;
11   @   ensures \result == false;
12   @   accessible footprint, c.footprint;
13   @   assignable \strictly_nothing;
14   @ also
15   @ public exceptional_behavior
16   @   requires (c == null) || !addSupported ||
17         (!supportNullElements && !c.noNullElements);
18   @   signals (NullPointerException) c == null ||
19         (!supportNullElements && !c.noNullElements);
20   @   signals (UnsupportedOperationException) !removeSupported;
21   @   signals_only UnsupportedOperationException,
22         ClassCastException, NullPointerException;
23   @   assignable \nothing;
24   @*/
25 boolean removeAll( /*@ nullable @*/ Collection c);
26 boolean removeAll( /*@ nullable @*/ Collection c);
```

Listing 2.14: Specification of the method `removeAll(Collection c)`

For the first normal behavior, the ensures part states that there should not be an element in the collection anymore that is also in c. The second `ensures` states that the method should return `true` if and only if the collection has shrunk because of the operation.

Several exceptions might be thrown. A `NullPointerException` might be thrown when one of the elements in `c` is `null` or `c` itself is `null`. The `UnsupportedOperationException` is not further specified.

Alternative specification

For this method it is also possible to specify what is contained in the collection - after the method returns - by specifying it using the method `contains(Object o)`. This can be done as shown in Listing 2.15;

```
@   ensures (\forall int i; 0 <= i && i < c.seqLength;
      !contains((Object)c.repr[i]));
```

Listing 2.15: Alternative specification using `Contains(Object o)`

When the specification is done this way, providing a proof might be easier since the proof obligation is a little shorter, and KeY may use the method contract of `contains(Object o)`.

2.1.13. method `retainAll(Collection c)`

The method `retainAll(Collection c)` is again an optional operation, and therefore might throw an `UnsupportedOperationException` when not implemented. Calling the method will result in removing every element from the collection that is not contained within `c`. If the method returns `true` the collection was modified by the operation. The method might throw an `UnsupportedOperationException`, `ClassCastException` or `NullPointerException`. Again, the `ClassCastException` is optional and the `NullPointerException` might optionally be thrown when one of the elements of `c` is `null`, but should be thrown when `c` itself is `null`. When `c` equals the current collection, the method should return `false` and all elements should remain the same.

```
1  /*@ public normal_behavior
2    @   requires c.\inv && \disjoint(footprint, c.footprint);
3    @   requires c != null;
4    @   requires !supportNullElements ==> c.noNullElements;
5    @   ensures seqLength == (\sum int i; 0 <= i && i < \old(seqLength);
6          (\exists int j; 0 <= j && j < c.seqLength;
7          (Object)c.repr[j] == \old((Object)repr[i])) ? 1 : 0);
8    @   ensures \result <==> seqLength < \old(seqLength);
9    @   ensures (\forall int i; 0 <= i && i < seqLength;
10         c.contains((Object)repr[i]));
11   @   assignable footprint;
12   @ also
13   @ public normal_behavior
14   @   requires c.\inv && \subset(footprint, c.footprint)
15         && \subset(c.footprint, footprint);
```

```
16   @   ensures \result == false;
17   @   assignable \strictly_nothing;
18   @ also
19   @ public exceptional_behavior
20   @   signals (ClassCastException) true;
21   @   signals (NullPointerException) c == null ||
22         (!supportNullElements && !c.noNullElements);
23   @   signals (UnsupportedOperationException) !removeSupported ==> true;
24   @   signals_only UnsupportedOperationException,
25         ClassCastException, NullPointerException;
26   @   assignable \nothing;
27   @*/
28 boolean retainAll(/*@ nullable @*/ Collection c);
```

Listing 2.16: Specification of the method `retainAll(Collection c)`

The less obvious `ensures` clauses for the normal behavior seen in Listing 2.16 should be seen as follows; every element in `c` - which is also a member of the collection - is counted to get the new value for `seqLength`. The next `ensures` clause specifies that for every element in the collection after the operation it must hold, that the element is also part of `c`.

When `c` equals the collection, i.e., footprints of both `c` and the collection are a subset of each otherthe method should return `false`, the collection being unmodified.

2.1.14. method `clear`

The method clear is also an optional operation, it may throw an `UnsupportedOperationException` when not implemented. A call to clear results in the collection being empty.

```
1  /*@ public normal_behavior
2    @   ensures seqLength == 0;
3    @   ensures repr == \seq_empty;
4    @   assignable footprint;
5    @ also
6    @ public exceptional_behavior
7    @   requires !clearSupported;
8    @   signals (UnsupportedOperationException)
9          !clearSupported ==> true;
10   @   signals_only UnsupportedOperationException;
11   @   assignable \nothing;
12   @*/
```

```
13 void clear();
```

Listing 2.17: Specification of the method `clear`

As can be seen in Listing 2.17, the only post-condition the method has, is that `seqLength` will be equal to `0` afterwards.

2.1.15. Discussion

The representation of a collection is specified as a sequence. However, using a representation of a set would make the specification higher-level. The choice of using a sequence for data abstraction here has the following reason; subtyping a collection with a representation of a set would mean that also a list should be represented by a sequence, or otherwise, additional specifications should be added for the subclasses in a way that both representation for a set aswell as a sequence would be correct. To overcome the burden of having multiple specifications for representations, sequences are chosen as they seem to be usable for all kind of collections, i.e., they allow duplicates, `null` values, and ordering – although not considered – can be simulated by adding and removing from the representation the right way.

2.2. `List` specifications

This section describes the specifications for the interface `List` located at `java.util.List`. First the class-level specifications about `List` will be explained, afterwhich the specification of methods from the `List` interface will be explained. The specifications in this section for the methods in `List`, only consist of the methods that were not yet specified in the interface `Collection`, or were specified but has additional stipulations put on them by the `List` interface.

2.2.1. Class-level specifications

From the informal description it can be seen that the `List` interface is an extension on the root interface `Collection`. A list is an ordered `Collection`, which means that a programmer has precise control over where in a list an element should be inserted or removed. Lists typically allow duplicates and multiple `null` elements. The informal

description of the interface `List` places a note on the acceptance of having lists that have themselves as an element - which is allowed - as the methods `equals` and `hashCode` are no longer well defined on such lists. The only additional specification at the interface level is specified in the listing below.

```
1  //@ public instance model boolean setSupported;
2  //@ public accessible setSupported: \nothing;
3
4  /*@ public represents supportNullElements = true;
5    @ public represents supportDuplicates = true;
6    @ public represents elementsHaveOrder = false;
7  //    @ public model nullable instance List parentList;
8  //    @ public model instance int startIndex;
9  //    @ public represents parentList = null;
10   @*/
```

Listing 2.18: Class-level specifications

Besides previous *supported* boolean model fields, the `List` interface introduces `setSupported`. In case sublists are used several additional specifications are needed, e.g., model fields for a `startIndex` and a `parentList`. At the moment they are not supported by the specifications, and no verifications have been performed using `subLists` for this study. In case they would be implemented, every method gets additional specifications in case there is a `parentList` or not, i.e., if `parentList` is `null` or not. Default values for `supportNullElements`, `supportDuplicates` and `elementsHaveOrder` are set here as well.

2.2.2. method `add(Object o)`

The `add(Object o)` in `List` gives the additional specification that `o` should be placed at the end of the `List` when added. See section 2.1.8 for the specifications of `add(Object o)` as specified by `Collection`. Since already at the collection level `seq` is built up this way, no additional specifications are needed here.

2.2.3. method `remove(Object o)`

The method `remove(Object o)` is also already specified for `Collection` (Section 2.1.9) and has some additional specification details for `List`. That is, when `o` is removed from

a list it should be the first occurrence found in that list. The additional specification needed can be found in Listing 2.19.

```
1  /*@ public normal_behavior
2    @   requires !supportNullElements ==> (o != null);
3    @   ensures contains(o) ==>
4          (\result && seqLength == \old(seqLength)-1
5            && repr == \seq_concat(
6               \seq_sub(\old(repr), 0, indexOf(o)),
7               \seq_sub(\old(repr), indexOf(o)+1, \old(seqLength))
8            )
9          );
10   @   ensures !contains(o) ==>
11         (!\result && seqLength == \old(seqLength)
12           && repr == \old(repr));
13   @   ensures \new_elems_fresh(footprint);
14   @   assignable footprint;
15   @*/
16 boolean remove( /*@ nullable @*/ Object o);
```

Listing 2.19: Specification of the method `remove(Object o)`

2.2.4. method `addAll(int index, Collection c)`

The method `addAll(int index, Collection c)` places additional postconditions on the method `addAll(Collection c)`. Namely, this time the elements in `c` should be inserted starting at `index` of list, whereby old locations starting at `index` should be shifted to the right. As the method is not specified before it should contain the full specification, therefore also the `NullPointerException` and the additional `IndexOutOfBoundException` are considered.

```
1  /*@ public normal_behavior
2    @   requires index >= 0 && index <= seqLength;
3    @   requires c.\inv && \disjoint(footprint, c.footprint);
4    @   requires c != null;
5    @   requires !supportNullElements ==> c.noNullElements;
6    @   requires addSupported;
7    @   ensures (\forall int i; 0 <= i && i < index;
8          repr[i] == \old(repr[i]));
9    @   ensures supportDuplicates ==> (\forall int j;
10         0 <= j && j < c.seqLength; repr[index + j] == c.repr[j]);
11   @   ensures supportDuplicates ==> (\forall int k;
```

```
12          c.seqLength + index <= k && k < seqLength;
13          repr[k] == \old(repr[k - index]));
14    @     ensures seqLength == \old(seqLength) + c.seqLength;
15    @     ensures seqLength > \old(seqLength) <==> \result;
16    @     ensures \new_elems_fresh(footprint);
17    @     assignable footprint;
18    @ also
19    @ public exceptional_behavior
20    @     requires (!supportNullElements && !c.noNullElements) || !
          addSupported;
21    @     signals (NullPointerException) (!supportNullElements && !c.
          noNullElements);
22    @     signals (UnsupportedOperationException) !addSupported;
23    @     signals_only UnsupportedOperationException, ClassCastException,
24            IllegalArgumentException, IndexOutOfBoundsException;
25    @     assignable \nothing;
26    @*/
27  boolean addAll(int index, /*@ nullable @*/ Collection c);
```

Listing 2.20: Specification of the method `addAll(int index, Collection c)`

Listing 2.20 provides a specification. In case `index` lays outside the bounds of the list, an exceptional behavior has been specified for the conditions that should hold when either an `IndexOutOfBoundsException` or `NullPointerException` should be thrown.

2.2.5. method `get(int index)`

The method `get(int index)` returns the element located at `index` of the list. When `index` lays outside the bounds of the list an `IndexOutOfBoundsException` should be thrown. The Listing below specifies this behavior.

```
1   /*@ public normal_behavior
2     @     requires index >= 0 && index < seqLength;
3     @     ensures \result == repr[index];
4     @     assignable \strictly_nothing;
5     @ also
6     @ public exceptional_behavior
7     @     requires index < 0 || index >= seqLength;
8     @     signals (IndexOutOfBoundsException) true;
9     @     assignable \nothing;
10    @*/
```

```
11 /*@ nullable @*/ Object get(int index);
```

Listing 2.21: Specification of the method `get(int index)`

2.2.6. method `set(int index, Object element)`

The method `set(int index, Object element)` is an optional operation. When implemented it should replace the element specified at position `index` with `element`, and return the element located at `index` before the operation was performed. Possible exceptions for this method are `UnsupportedOperationException`, `ClassCastException`, `NullPointerException`, `IllegalArgumentException` and `IndexOutOfBoundsException`. The `ClassCastException` should be thrown when the class of `element` prevents it from being added to the list. `IllegalArgumentException` should be thrown when some property of `element` prevents it from being added to the list. `UnsupportedOperationException`, `NullPointerException` and `IndexOutOfBoundsException` should be thrown when the method is not implemented, `element` is `null` and the list prohibits `null` elements being added, or `index` lays not within the bounds of the list, respectively. Listing 2.22 specifies this behavior.

```
1  /*@ public normal_behavior
2    @  requires \typeof(element) == \type(Object);
3    @  requires index >= 0 && index < seqLength;
4    @  requires !supportNullElements ==> (element != null);
5    @   ensures repr[index] == element;
6    @   ensures \result == \old(repr[index]);
7    @  ensures \new_elems_fresh(footprint);
8    @  assignable footprint;
9    @ also
10   @ public exceptional_behavior
11   @  requires index < 0 || index >= seqLength ||
12        (!supportNullElements && element == null) || !setSupported;
13   @  signals (IndexOutOfBoundsException) index < 0
14         || index >= seqLength;
15   @  signals (NullPointerException)
16         !supportNullElements ==> element == null;
17   @  signals (UnsupportedOperationException) !setSupported ==> true;
18   @  signals_only UnsupportedOperationException, ClassCastException,
19         IllegalArgumentException, IndexOutOfBoundsException,
20         NullPointerException;
21   @  assignable \nothing;
```

```
22   @*/
23 /*@ nullable @*/ Object set(int index, /*@ nullable @*/ Object element);
```

Listing 2.22: Specification of the method `set(int index, Object element)`

The normal behavior specifies, when `index` is within bounds of the list and `element` is not `null` when `null` elements are not allowed, the element at `index` should become `element`, fresh memory is allocated, and the method returns the element at `index` before the operation. One of the following exceptions might be thrown; `UnsupportedOperationException`, `ClassCastException`, `IllegalArgumentException`, `IndexOutOfBoundsException` or `NullPointerException`. Only `IndexOutOfBoundsException` and `NullPointerException` have strict rules on when they should be thrown.

Alternative specification

Lines five and six of the above listing could alternatively be specified with `get(index)` instead of `repr[index]`.

2.2.7. method `add(int index, Object element)`

The method `add(int index, Object element)` is again an optional operation. The method inserts `element` at location index when implemented. All – if any – elements starting from index shift one location to the right. The same exceptions apply to this method as with the method `set(int index, Object element)` specified in Section 2.22, and are therefore not repeated here. Listing 2.23 gives a formal specification.

```
1  /*@ public normal_behavior
2    @   requires \typeof(element) == \type(java.lang.Object);
3    @   requires index >= 0 && index <= seqLength;
4    @   requires !supportNullElements ==> (element != null);
5    @   requires !supportDuplicates ==> !contains(element);
6    @   ensures seqLength == \old(seqLength) + 1;
7    @   ensures repr == \seq_concat(
8          \seq_concat(
9            \seq_sub(\old(repr), 0, index),
10           \seq_singleton(element)
11         ),
12         \seq_sub(\old(repr), index, \old(seqLength))
13       );
```

```
14   @   assignable footprint;
15   @ also
16   @ public exceptional_behavior
17   @   requires \typeof(element) == \type(java.lang.Object);
18   @   signals (IndexOutOfBoundsException) index < 0 || index > seqLength;
19   @   signals (NullPointerException) !supportNullElements && element ==
         null;
20   @   signals (UnsupportedOperationException) !addSupported;
21   @   signals_only UnsupportedOperationException,
22         ClassCastException, IllegalArgumentException,
23         IndexOutOfBoundsException, NullPointerException;
24   @   assignable \nothing;
25   @*/
26 void add(int index, /*@ nullable @*/ Object element);
```

Listing 2.23: Specification of the method `add(int index, Object element)`

2.2.8. method `remove(int index)`

The method `remove(int index)` removes the element at `index` of the list, when implemented. Any element located after `index`, shifts one element to the left. The method returns the element located at `index` before the operation was performed. Two exceptional cases exist, triggered when either the method is not implemented, or when the given `index` is out of bounds of the list, which should throw either an `UnsupportedOperationException` or `IndexOutOfBoundsException`.

```
1  /*@ public normal_behavior
2    @   requires index >= 0 && index < seqLength;
3    @   ensures repr == \seq_concat(
4          \seq_sub(\old(repr), 0, index),
5          \seq_sub(\old(repr), index+1, \old(seqLength))
6        );
7    @   ensures seqLength == \old(seqLength)-1;
8    @   ensures \result == \old(get(index));
9    @   ensures \new_elems_fresh(footprint);
10   @   assignable footprint;
11   @ also
12   @ public exceptional_behavior
13   @   requires index < 0 || index > seqLength || !removeSupported;
14   @   signals (IndexOutOfBoundsException) index < 0 || index > seqLength;
15   @   signals (UnsupportedOperationException) !removeSupported ==> true;
```

```
16   @   signals_only UnsupportedOperationException,
         IndexOutOfBoundsException;
17   @   assignable footprint;
18   @*/
19 /*@ nullable @*/ Object remove(int index);
```

Listing 2.24: Specification of the method `remove(int index)`

Listing 2.24 shows a formal specification. When the requirements are met, the post-conditions of the normal behavior specify that old elements before `index` should still be there after the operation is performed, the indexes after `index` should contain the elements previously located an index higher, the total length of the list should be decreased by one, the result is the element previously located at `index` and modifications should be done with freshly allocated memory.

The exceptional behavior specifies that an `IndexOutOfBoundsException` should be thrown when the index lays outside the bounds of the list.

2.2.9. method `indexOf(Object o)`

The method `indexOf(Object o)` returns the index of the first occurrence of `o` in the list, and returns `-1` if o does not occur in the list. A `NullPointerExcepion` might optionally be thrown if `o` is null, and the list does not allow null elements.

```
1  /*@ public normal_behavior
2    @   requires \typeof(o) == \type(Object);
3    @   ensures (\forall int k; 0 <= k && k < seqLength; repr[k] != o)
4          ==> \result == -1;
5    @   ensures (\exists int k; 0 <= k && k < seqLength; repr[k] == o)
6          ==> (\result >= 0 && \result < seqLength && repr[\result] == o);
7    @   ensures !(\exists int k; 0 <= k && k < \result; repr[k] == o);
8    @   assignable \nothing;
9    @ public exceptional_behavior
10   @   requires !supportNullElements && o == null;
11   @   signals (NullPointerException) !supportNullElements && o == null;
12   @   signals_only NullPointerException;
13   @   assignable \nothing;
14   @*/
15 int indexOf( /*@ nullable @*/ Object o);
```

Listing 2.25: Specification of the method `indexOf(Object o)`

2.2.10. method `lastIndexOf(Object o)`

The method `lastIndexOf(Object o)` is very similar to the method specified in the previous section. The only difference is that this method returns the last index instead of the first index where `o` occurs in the list. A formal specification can be found below.

```
1  /*@ public normal_behavior
2    @   requires !supportNullElements ==> (o != null);
3    @   ensures (\forall int k; 0 <= k && k < seqLength; repr[k] != o)
4          ==> \result == -1;
5    @   ensures (\exists int k; 0 <= k && k < seqLength; repr[k] == o)
6          ==> (\result >= 0 && \result < seqLength && repr[\result] == o);
7    @   ensures !(\exists int k; \result < k && k < seqLength; repr[k] == o)
       ;
8    @   assignable \nothing;
9    @ also
10   @ public exceptional_behavior
11   @   requires !supportNullElements && o == null;
12   @   signals (NullPointerException) !supportNullElements && o == null;
13   @   signals_only NullPointerException;
14   @   assignable \nothing;
15   @*/
16 int lastIndexOf( /*@ nullable @*/ Object o);
```

Listing 2.26: Specification of the method `lastIndexOf(Object o)`

Alternative specification

The ensures part could also be specified like depicted in Listing 2.27.

```
@   ensures \result == (\max int i; 0 <= i && i < seqLength;
      ((Object)repr[i] == o) ? i : -1);
```

Listing 2.27: Alternative specification for `\result`

However, as explained later, this was not allowed by KeY at this point (see page 84).

2.2.11. method `listIterator`

The method `listIterator` returns a `ListIterator` over the elements in the list. Specifications are kept to a minimum and only state that the method is `pure`, `footprint`

needs to be accessible and a number of properties about the iterator should be set. Explanations about the different properties can be found in Section 2.4, where specifications of `ListIterator` are explained. The specifications for the method `iterator` can be found in Listing 2.28 below.

```
1  /*@ public normal_behavior
2    @   ensures \result.collection == this;
3    @   ensures \result.position == -1;
4    @   ensures \fresh(\result);
5    @*/
6  /*@ pure @*/ ListIterator listIterator();
```

Listing 2.28: Specification of the method `listIterator`

2.2.12. method `listIterator(int index)`

`listIterator(int index)` also returns a `ListIterator` over the elements in the list, this time starting at `index`. The method should throw an exception when `index` is not within bounds of the list. A formal specification can be found below. Explanations about the different properties can be found in Section 2.4, where specifications, of `ListIterator` are explained.

```
1  /*@ public normal_behavior
2    @   requires index >= 0 && index < seqLength;
3    @   ensures \result.collection == this;
4    @   ensures \result.position == index -1;
5    @   ensures \fresh(\result);
6    @ also
7    @ public exceptional_behavior
8    @   requires index < 0 || index >= seqLength;
9    @   signals (IndexOutOfBoundsException) index < 0 || index >= seqLength;
10   @   signals_only IndexOutOfBoundsException;
11   @*/
12 /*@ pure @*/ ListIterator listIterator(int index);
```

Listing 2.29: Specification of the method `listIterator(int index)`

2.3. `Iterator` specifications

At `java.util.Iterator` the interface `Iterator` can be found, for which specifications in JML* are described in this section. First class-level specifications about `Iterator` wil be explained, then the specification of methods from the `Iterator` interface will be explained. The discussion at the end of this section describes an alternative approach to specify behavior of `Iterator` found in literature.

2.3.1. Class-level specifications

An iterator is used to iterate over a collection. Besides iterating, iterators allow the caller to remove elements from underlying collection during the iteration. However, removing elements is only allowed when either the iterator has just been constructed, or a call to `next()` has been performed and no subsequent calls to `remove()` have been performed.

Several formal specifications can be deduced from these informal specifications, which are provided in Listing 2.30 and are described in the remainder of this section.

```
1  //@ public model instance \locset footprint;
2  //@ public accessible \inv: footprint;
3  //@ public accessible footprint: footprint;
4
5  /// @ public ghost instance Object currentObject;
6  /// @ public instance invariant currentObject.equals(
7  ///     (Object)collection.repr[position]);
8
9  //@ public instance invariant collection != null && collection.\inv;
10 //@ public instance invariant position >= -1 && position <= seqLength;
11
12 /*@ public model instance boolean allowChange;
13   @ public model instance int position;
14   @ public model instance boolean modificationOkay;
15   @ public ghost instance Collection collection;
16   @ public accessible allowChange: footprint;
17   @ public accessible position: footprint;
18   @ public model instance int seqLength;
19   @ public accessible seqLength: footprint;
20   @*/
```

Listing 2.30: Class-level specifications for the interface `Iterator`

The first few lines define a dynamic frame with the name `footprint`. The ghost instance `currentObject` – as a comment here – is used to hold the current element the iterator has and is used to cover requirements for the method `remove()`, however, this ghost instance was not possible to use as the `equals` methods could not be used as intended (see Section 3.1.3). The model field `modificationOkay` was introduced, represents whether in the current state of the object it is allowed to remove or add an object.

The ghost instances `allowRemove` and `position` are used to allow to specify whether or not it is allowed to remove an element, and at what position the iterator is at, respectively. The model instance `collection` is used to represent the underlying collection. Next, framing conditions are specified, and an additional `model instance int seqLength` is added to directly represent the size of `collection`.

2.3.2. method `hasNext()`

The method `hasNext()` returns `true` if the underlying collection has more elements. As shown below, this method only needs a few lines of specifications.

```
1 /*@ public normal_behavior
2   @   ensures \result <==> position < seqLength -1;
3   @*/
4 /*@ strictly_pure @*/ boolean hasNext();
```

Listing 2.31: Specification of the method `hasNext()`

2.3.3. method `next()`

This method returns the next element in the iteration if the iterator has a next element, and throws an `NoSuchElementException` if there is no next element. Listing 2.32 gives a formal specification.

```
1 /*@ public normal_behavior
2   @   requires modificationOkay;
3   @   requires position < seqLength - 1;
4   @   ensures position == \old(position) + 1;
5   @   ensures allowChange == true;
6   @   ensures \result == collection.repr[position];
7   @   assignable footprint;
8   @ also
9   @ public exceptional_behavior
```

```
10    @   requires position >= seqLength - 1 || !modificationOkay;
11    @   signals (NoSuchElementException nsee) position >= seqLength - 1;
12    @   signals (ConcurrentModificationException) !modificationOkay;
13    @   signals_only NoSuchElementException, ConcurrentModificationException
          ;
14    @   assignable \nothing;
15    @*/
16  /*@ nullable @*/ Object next();
```

Listing 2.32: Specification of the method `next()`

Based on the current position of the iterator, it could be seen whether the method should return an actual object, or there is no next element. When this method is called – and returns an `Object` – the `remove()` method may be called next, that is why `allowRemove` is ensured to be `true` afterwards. A `NoSuchElementException` should be thrown when `position` of the iterator is greater than or equal to the size of the collection the iterator is iterating over.

2.3.4. method `remove()`

If implemented, the method `remove()` removes from the underlying collection the last element returned by the iterator. It is not allowed to call this method more than once after a call to `next()`. The API documentation states that behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any other way other than by calling this method, i.e., by directly modifying the collection or another iterator. A formal specification can be found in Listing 2.33.

```
1   /*@ public normal_behavior
2     @   requires position >= 0 && position < seqLength;
3     @   requires allowChange && modificationOkay;
4     @   ensures seqLength == \old(seqLength) - 1;
5     @   ensures collection.repr == \seq_concat(
6               \seq_sub(collection.repr, 0, position - 1),
7               \seq_sub(collection.repr, position + 1, \old(collection.
                    seqLength) - 1)
8           );
9     @   ensures allowChange == false;
10    @   assignable footprint;
11    @ also
12    @ public exceptional_behavior
13    @   requires !allowChange;
```

```
14    @   signals (IllegalStateException) !allowChange;
15    @   signals_only IllegalStateException, UnsupportedOperationException;
16    @   assignable \nothing;
17    @*/
18  void remove();
```

Listing 2.33: Specification of the method `remove()`

Clearly, the method should throw an `IllegalStateException` when a call to `remove()` is done when not allowed. When allowed the method should remove the element at `position` of the iterator.

2.3.5. Discussion

Cok describes specifying Java iterators with JML in [19], based on Java 1.5. His solution addresses the additional requirement that only behavior of single access by an iterator is defined by introducing specifications for the time methods were being called. A model method `isValid()` checks whether no modifications were done after the iterator has been created, and returns `true` if that is the case. Model fields `lastModifiedTime` and `iteratorTime` are used to specify the time a collection was modified and when an iterator was created, respectively. For this to work, every method that structurally changes a collection should have additional specifications that state an update of the field `lastModifiedTime`. Construction of a new iterator should also update the `lastModifiedTime` field, this way already existing iterators get a result of `false` for a call to `isValid()` and therefore behavior is unspecified.

Using this approach would require a lot of additional specifications, as every method that makes structural changes to the collection should update the `lastModifiedTime`, add a model method `isValid()` and an additional field `iteratorTime` would be needed. Therefore, the approach taken here makes use of a single ghost instance that reflects the `Object` at `position` after a call to next. Furthermore, the method `remove()` has only specified behavior when the ghost field equals the current object at `position`.

2.4. `ListIterator` specifications

This section describes the specifications for the interface `ListIterator` located at `java.util.ListIterator`.

2.4.1. method `hasPrevious()`

The method `hasPrevious()` has very similar behavior as the method `hasNext()`. There is only a previous element if position is greater than 0. Listing 2.34 gives a formal specification.

```
1 /*@ public normal_behavior
2   @   ensures (position > 0) <==> \result;
3   @*/
4 /*@ strictly_pure @*/ boolean hasPrevious();
```

Listing 2.34: Specification of the method `hasPrevious()`

2.4.2. method `previous()`

Like `hasPrevious()` is the opposite of `hasNext()`, `previous()` is the opposite of `next()`. The method returns the previous element in the list the iterator operates on, and moves the cursor position backwards. When there is no previous element – and this method is called – a `NoSuchElementException` should be thrown.

```
1  /*@ public normal_behavior
2    @   requires position > 0;
3    @   ensures \result == collection.repr[position];
4    @   ensures position == \old(position) - 1;
5    @   ensures allowChange == true;
6    @   assignable footprint;
7    @ also
8    @ public exceptional_behavior
9    @   requires position <= 0;
10   @   signals (NoSuchElementException) position <= 0;
11   @   assignable \nothing;
12   @*/
13 /*@ nullable @*/ Object previous();
```

Listing 2.35: Specification of the method `previous()`

Listing 2.35 specifies formal behavior of the method `previous()`. Normal behavior states that the result should hold the value of the representation for `list` at `position` of the pre-state, `position` is decremented and `allowRemove` is `true` afterwards.

2.4.3. method `nextIndex()`

The method `nextIndex()` is new for the `ListIterator` and returns the index of the element that would be returned by a subsequent call to `next()`. When the iterator is at the end of the list, this method will return the size of the list. Listing 2.36 specifies formal behavior for `nextIndex()`.

```
1 /*@ public normal_behavior
2   @   ensures \result == ((position + 1 <= seqLength)
3         ? position + 1 : seqLength);
4   @*/
5 /*@ strictly_pure @*/ int nextIndex();
```

Listing 2.36: Specification of the method `nextIndex()`

The method `nextIndex()` has only normal behavior which is specified with `seqLength` that represents the size of `list`. If the new value of `position` will be less than or equal to the size of `list`, that value will be returned, otherwise, the size of `list` will be returned.

2.4.4. method `previousIndex()`

The method `previousIndex()` returns the element that would be returned by a subsequent call to `previous()`. The method returns `-1` if the list iterator is at the beginning of the list. Formal behavior is specified in Listing 2.37.

```
1 /*@ public normal_behavior
2   @  ensures \result == position;
3   @*/
4 /*@ strictly_pure @*/ int previousIndex();
```

Listing 2.37: Specification of the method `previousIndex()`

The normal behavior specified is similar to that of `nextIndex()`.

2.4.5. method `set(Object e)`

The method `set(Object e)` replaces the last element returned by `next()` or `previous()` with `e`. The operation is optional and throws an `UnsupportedOperationException` if the method is not implemented. Normal behavior is only possible if neither a call

to `remove()` nor `add(Object e)` have been made after the last call to `next()` or `previous()`. The method throws a `ClassCastException` if the class of `e` prevents it from being added to this list. A `IllegalArgumentException` or `IllegalStateException` is thrown if some propery of `e` prevents it from being added to this list, or if a call to `set(Object e)` was not allowed at this time, respectively. Listing 2.38 gives a formal specification.

```
1  /*@ public normal_behavior
2    @   requires \typeof(e) == \type(java.lang.Object);
3    @   requires allowChange;
4    @   ensures allowChange == false;
5    @   ensures (\forall int i; 0 <= i && i < position;
6          \old(collection.repr[i]) == collection.repr[i]);
7    @   ensures collection.repr[position] == e;
8    @   ensures (\forall int i; position < i && i < seqLength;
9          \old(collection.repr[i]) == collection.repr[i]);
10   @   ensures \new_elems_fresh(collection.footprint);
11   @   assignable collection.footprint;
12   @ also
13   @ public exceptional_behavior
14   @   requires \typeof(e) == \type(java.lang.Object);
15   @   requires !allowChange;
16   @   signals (IllegalStateException) !allowChange;
17   @   signals_only UnsupportedOperationException,
18         ClassCastException, IllegalArgumentException,
19         IllegalStateException;
20   @   assignable \nothing;
21   @*/
22 void set(/*@ nullable @*/ Object e);
```

Listing 2.38: Specification of the method `set(Object e)`

When `allowRemove` is `true`, the normal behavior states that this value should be falsified afterwards, old elements of the list at other positions than `position` should remain the same, and `currentObject` as well as the representation of `list` at `position` should have been updated.

If a `IllegalStateException` is being thrown, it should be the case that `allowRemove` was `false` when `set(Object e)` was called.

2.4.6. method `add(Object e)`

If implemented, the method `add(Object e)` inserts `e` in the list the iterator operates on. `e` is inserted immediately before the element that would be returned by `next()`, if any, and after the element that would be returned by `previous()`, if any. A call to method `add(Object e)` will increase the values returned by `nextIndex()` or `previousIndex()` by one. Besides an `UnsupportedOperationException` being thrown when the method is not implemented, a `ClassCastException` or `IllegalArgumentException` might also be thrown by this method. A `ClassCastException` will be thrown in case the class of `e` prevents it from being added to the list, and an `IllegalArgumentException` will be thrown if some property of `e` prevents it from being added to the list. Listing 2.39 gives a formal specification.

```
1  /*@ public normal_behavior
2    @   requires \typeof(e) == \type(java.lang.Object);
3    @   requires collection.addSupported;
4    @   ensures (\forall int i; 0 <= i && i < position;
5          \old(collection.repr[i]) == collection.repr[i]);
6    @   ensures collection.repr[position] == e;
7    @   ensures (\forall int i; position < i && i < seqLength;
8          \old(collection.repr[i]) == collection.repr[i+1]);
9    @   ensures position == \old(position) + 1;
10   @   ensures \new_elems_fresh(collection.footprint);
11   @   assignable collection.footprint;
12   @ also
13   @ public exceptional_behavior
14   @   requires \typeof(e) == \type(java.lang.Object);
15   @   requires !modificationOkay || !collection.addSupported;
16   @   signals (ConcurrentModificationException) !modificationOkay;
17   @   signals (UnsupportedOperationException)
18         !collection.addSupported;
19   @   signals_only UnsupportedOperationException,
20         ConcurrentModificationException, ClassCastException,
21         IllegalArgumentException, IndexOutOfBoundsException;
22   @   assignable \nothing;
23   @*/
24 void add(/*@ nullable @*/ Object e);
```

Listing 2.39: Specification of the method `add(Object e)`

Normal specifications state that `allowRemove` should be `true` to be applicable. After the method returns, `allowRemove` should be `false`, old elements before `position` in the

representation of `list` should be the same afterwards, the element added should be at the position of `position` before the method was called, and all elements following the newly inserted element should still be in the representation of `list`, but with incremented position.

2.5. Findings

Loading the specifications this chapter into KeY gives no problems. Note that loading the four interfaces above does give problems when not also loading the mentioned `Exception` classes in the exceptional behaviors. One needs to provide dummy implementations for these `Exception` classes to let KeY pass loading, i.e., these classes should be in the same directory as the four interfaces.

Several methods described in previous sections have alternative ways of specifying them (see 2.1.1, 2.1.10, 2.1.12 and 2.2.10). Most of the time this boils down to describing methods with already formalized methods, e.g., using `contains(Object)` to describe the behavior of `containsAll(Collection c)`.

Often it is also possible to interchange ghost fields – an extension of state, model fields – an abstraction of state – and model methods – an extension to model fields – with each other [11]. Strictly taken it would not be fine to use ghost fields here, as the specifications are about interfaces, and ghost fields use JML `set` operations within method bodies to update state. Which means ghost fields would be conceptually wrong. However, this is not a problem for static checking, and would only result in failure using a runtime checker. The intended use of a model field is as a means to hold an abstract representation of the state of an object; in a concrete class each model field would be provided a representation. Model fields are fine to use within an interface, but also need a concrete representation if used for runtime checking. Finally, model methods are an alternate way of providing the functionality of a model field. Model methods can be either directly given an implementation at the interface level, or later on at the implementing classes of the interfaces. Unfortunately, model methods are not yet part of KeY's core functionality and therefore left out for now.

At several locations in the specifications model fields for ordering and duplication of elements are used, first at the class-level specification and later on at methods that structurally modify a collection. However, the specifications of these fields are not

always complete. This is on purpose, and could be expanded whenever collection implementations that actually need information about ordering or duplication need to be verified.

3. Verification

This chapter describes the verification of the specifications described in the previous chapter. Since interfaces themselves cannot be proved correct statically, verification is performed on the implementing (abstract) classes. Since only selected parts of the Java Collections Framework are verified, it can only be stated that these parts of the specified interfaces can be considered correct.

Before diving into the sample verifications in Section 3.2, a little background of KeY, how to use the KeY tool, and some limitations encountered during the verification will be discussed in Section 3.1.

3.1. KeY

KeY is a standalone prover and can statically verify JML*-annotated Java in a modular way, supporting a large part of Java 1.4. This chapter explains how KeY was used for modular static verification of Java for this project, explains the basic mechanisms of the tool, and discusses limitations encountered during verification. Furthermore, the approach taken for verification is explained in Section 3.1.4 on page 79.

3.1.1. Usage of KeY

When one starts KeY, the first thing to do is to load the JML*-annotated Java code. When no syntactic and semantic problems with the specifications are found, a contract for the behavior of a method, or dependency condition (see dynamic frames on page 22) can be chosen for verification. The proof obligation for normal or exceptional behavior of a method contains the chosen method, a pre- and postcondition, a modification

clause and a termination clause. A framing condition contains a precondition and a dependency clause. Once the proof obligation is loaded, the clauses from the obligations are transformed into first-order predicate logic with executional code fragments (also see Section 3.1.2 on the facing page), and one can either interactively prove the obligations by hand or let KeY try to prove the obligations automatically. Proving these obligations automatically is sometimes also possible using Satisfiability Modulo Theories (SMT) solvers. SMT solvers are able to reason over first-order theories and provide a verdict whether some first-order logic formula is valid, false – providing a counter example – or unknown [3]. However, most of the time one is more likely to prove obligations with KeY itself since the SMT solvers can only reason about a subset of what KeY itself can reason about, which was recommended by the KeY developers. KeY, however, does not provide counter examples, now it can utilise counter examples from SMT solvers.

KeY for this project

For the verification of selected parts of the Java Collections Framework – constantly – the latest nightly build of KeY was used, which is the current development version of KeY. Using the latest build gives benefits as well as drawbacks, e.g., new features will get introduced here first and bugs fixed sooner, but at the same time every iteration may introduce new bugs. Furthermore, semantics of features might slightly change or features can get removed over time. For example, a change in semantics is the specification construct `\seq_sub(sequence, lower, upper)`, which selects part of a sequence (also see abstract data types on page 26) and now behaves the same as the method `subList`, i.e., the lower bounds is included, but the upper bound is excluded, whereas before the upper bound was included as well. Note that the method `subList` was not yet considered for this thesis. The operation `\dl_array2seq` which transformed an array to a sequence was – although still present in an example that comes with KeY– unfortunately is not part of the latest builds anymore.

Running a local copy of KeY as opposed to running KeY from its webpage[1] makes it easier to spot a bug in the tool itself, since the command line from which KeY gets invoked displays the exceptions that KeY throws. This way, when an exception is thrown, the source code could be slightly modified and compiled again to gain more insight in what KeY has a problem with. Information on running KeY can be found on the same section

[1] http://www.key-project.org/download/

of the KeY website where additionally information on supported features can be found, and especially feature differences between the latest releases.

3.1.2. Basic mechanism of KeY

Loading JML*-annotated Java source into KeY results in a transformation from annotated source to proof obligations. These proof obligations consists of formulas whose logical validity corresponds with the correctness of the Java source code with respect to the specification. The logic used for that is called dynamic logic, an extension of first-order predicate logic, that uses modal operators which contain executable program fragments of some programming language [42]. In case of KeY the programming language is a large subset of Java 1.4, hence the logic is called Java Dynamic Logic or JavaDL. A feature of KeY is that it can transform JavaDL – with state updates – to formulas in first-order predicate logic with symbolic execution. Which means that the problem gets reduced to proving the logical validity of this logic with built-in theories, either by KeY itself or SMT solvers as explained in the previous section.

Modal logic introduces modal operators to propositional logic, namely *box* $\Box$ and *diamond* $\Diamond$ [7]. When doing model checking, e.g., using Linear Temporal Logic (LTL), $\Box p$ means for all future states the proposition p should hold, and $\Diamond q$ that at some future state q should hold [1]. In dynamic logic, as used by KeY, their usage is a little different. The box and diamond operators are still present, only this time they surround the body of a method that needs to be proved, i.e., $[b]\, p$ and $\langle b \rangle\, q$ where p and q stand for one or more properties and b for the body of some method in Java. Since sequential Java programs are deterministic it means either that b terminates or not, hence there is one or no final state. For the box expression this means that either b does not terminate and nothing has to hold for this specific expression, or b does terminate and p should hold, often indicated as partial correctness. For the diamond expression it should additionally hold that b terminates, often indicated with total correctness.

Dynamic logic is a generalisation of Hoare logic [15]. The formula $p \implies [b]\, q$, where p and q are first-order formulas, has the same meaning as the Hoare triple $\{p\}b\{q\}$. If b is executed and terminates, and p holds in the pre-state, q should hold in the post-state. In contrast to Hoare logic, dynamic logic is closed under logical operators, i.e., including the modal operators $[b]$ and $\langle b \rangle$ [42]. That dynamic logic is closed under logical operators means that statements in dynamic logic can be combined in bigger

formulas with arbitrary propositional operators or quantifiers, and even into nestings of formulas [34].

3.1.3. Encountered limitations

Several limitations have been encountered during verification, some of which are also mentioned on the KeY website, i.e., the limitations on `set` statements (same page) and the `\old` expression (on page 75). In the subsections below these limitations are addressed, explained how they were dealed with in this project and what this means for verification and specification.

`set` statements

The `set` statement in JML makes it possible to update the state of ghost fields. In KeY when used in a method body, `set` may not be the last statement of a code block, since otherwise KeY will ignore the statement. To bypass this problem – when the expression is actually used and is also the last statement of a block – an empty block statement can be inserted just after the statement. Using this technique might slightly reduce readability of the code. Verification or specification will not be different compared to when the `set` statement was actually working without adding an additional empty block statement.

`\min` and `\max` expressions

Although supported in the latest builds, the `\min` and `\max` expressions, which in JML go over a certain range and find respectively a minimum or maximum value on some expression, did not completely work as expected. A problem revealed itself when trying to use `\min` to find the minimum value where an object occurs in an array. Since when the object was not found in the array `-1` should be returned, an expression very similar to the following was used `(\min int i; 0 <= i && i < a.length; (a[i] == obj) ? i : -1)`. The specificaton did not get loaded into KeY giving an error about the last part of the expression, which compares `a[i]` with `obj` and returns either `i` or `-1`. Probably, having complex expressions inside the `\min` or `\max` construct are not yet part of KeY's features. A solution to bypass this problem in this case was to use different specification constructs, i.e., the specification in Listing 3.1 is used.

```
1 @ ensures (\forall int k; 0 <= k &&
2     k <= seqLength-1; repr[k] != o) ==> \result == -1;
3 @ ensures (\exists int k; 0 <= k &&
4     k <= seqLength-1; repr[k] == o) ==> (\result >= 0
5       && \result < seqLength && repr[\result] == o);
6 @ ensures !(\exists int k; 0 < k &&
7     k <= \result; repr[k] == o);
```

Listing 3.1: `\min` and `\max` alternative

The first clause states that when the `o` is not in the array, `-1` should be the result. The second clause states that when `o` is in the range of the array, the result should be as well, and furthermore, the result should also have the specific index where the array has the value `o`. The last clause makes sure that when there is more than one index which applies, the value of the array is `o`, it is indeed the minimum index which applies that is returned.

When one needs to use an inner expression for the `\min` or `\max` constructs, one should either make sure it is supported at the current build or use a different specification to express roughly the same. Replacing a `\min` construct the way described will decrease understandability of the specifications, i.e., it takes more time to understand what the specification actually does. However, verification possibly gets easier, instead of one relatively hard `\min` construct, three easier forall/exists constructs are used. Another option would be to use model methods, whereby potentially understandability as well as difficulty of verification can benefit. For example, using a slightly modified version of the specification above for the body of the model method, and giving it a name like `minIndex(Object o)`. Understandability will increase as the name suggests what the method does, and verification gets easier as explained before.

`\old` expression

The `\old` expression has the limitation that it cannot be used to reason about the arguments of a method from within the method, e.g., when used for specifying the behavior of loops. Essentially the problem is that some methods work on arguments directly and change them. KeY, however, assumes they would never change. This problem can be bypassed by copying this argument to an inner local variable. After providing an additional local variable, a loop can reason about this variable instead of the argument provided by the method. Of course, specifications should not alter

source code, but at this point there is no other way to work around this limitation. In principle verification gets a little harder, since you should know this limitation when using KeY and apply this solution, as otherwise it is not obvious why verification is not working. An example is provided in Section 3.7 on page 88 where lines 15 and 16 are added.

Sequences

The data abstraction of sequences used by KeY has the limitation that it is relatively new to the tool, and not everything one expects to work for it is implemented yet. An example is the `remove` operation – which could remove an index from a sequence. Since this operation is not yet implemented, one has to use a considerably bigger expression, i.e., express the remove operation by using subsequences and concatenation of these sequences. Addition can be specified in a similar way. The downside of this is that specifications get more complicated, hence understandability decreases. Difficulty of verification largely depends on the actual implementation of the `remove` operation. If the implementation directly translates to the aforementioned, difficulty of verification will stay the same.

Generics

As mentioned before, at the time being KeY supports a large subset of Java 1.4. Therefore – since generics needed to be removed from source – specifications about type restrictions that are related to generics are hard to make, and maybe even harder to verify at this point. Moreover, certain specification constructs like `\elemtype` and `\TYPE` [23] are not possible to use yet. This is why `ClassCastException`s are not considered. Normally a `ClassCastException` should be thrown when, e.g., a certain object is not allowed to be added to the collection as it is not a subclass of the elements of the collection. However, there is no way of storing information of the types of the elements in the collection. To prevent that a `ClassCastException` could actually be thrown, restrictions are made on the types of, e.g., the array with elements in `ArrayList` and objects that are added/removed. Making changes like these largely limits the expressiveness of the specifications, but also makes verification easier.

Since only little can be specified about types at this point, it was needed to put restrictions on the types of objects used in implementations. Here the choice has been

made to only allow `Object`, and `Object[]` as runtime types of respectively `Object` and `Object[]`. For instance the `elementData` array has the following restriction; `\typeof(elementData) == \type(java.lang.Object[])`.

With this restriction, it is at this point not possible to do, e.g., reasoning about ordered elements, since therefore objects should be `instanceof Comparable`, and plain objects are not instances of this or any other interface. More problems might reveal itself when a specific collection implementation requires an `IllegalArgumentException` to be thrown, e.g., when trying to add a certain element that does not satisfy a condition on the object. Although verification in general gets easier, specifications loses expressiveness.

Abstract classes

For abstract classes, KeY only allows static methods to be statically verified. This means that methods that have an actual implementation, but are part of an abstract class, are not subject to verification. When one wants to verify these non-abstract methods, one should extend the abstract class – and not override these methods – and next verify these methods for the class. For verification – in the end – the difficulty stays the same compared to verifying normal methods, however, it is more work to write additional classes and to get to a point where verification is actually possible.

Inner classes

Another problem encountered during the project is that KeY was not able to reason about inner classes. To make verification possible, inner classes have been made regular classes of the project, i.e., inner classes were extracted, constructor methods needed to be adjusted slightly and references within the classes were updated. If specifications of inner classes would work, extraction of these inner classes would not be necessary, but otherwise not much changes.

The methods `equals` and `compareTo`

Certain methods use the method `equals` from object. `Object` is one of the classes that KeY uses an own implementation for, the class is provided and a few specifications are given. However, the method `equals` only got the annotation that it is `pure`, which was not enough for this project. As a first try, specifications were added for

the `equals` method according to the informal specifications, i.e., the `equals` method should be *reflexive*, *symmetric*, *transitive*, *consistent* and for any non-null value of `x`, `x.equals(null)` should return `false`. This was done in the following way;

```
1  /*@ public normal_behavior
2    @   requires object != null;
3    @   ensures this.equals(this);
4    @   ensures (\forall Object o1;
5          this.equals(o1) <==> o1.equals(this));
6    @   ensures (\forall Object o1; (\forall Object o2;
7          this.equals(o1) && o1.equals(o2)
8          ==> o2.equals(this)));
9    @ also
10   @   requires object == null;
11   @   ensures !\result;
12   @*/
13 public /*@ pure @*/ boolean equals(/*@ nullable @*/ Object object)
```

Listing 3.2: Contract for method `Equals`

After compiling KeY again with the added specifications, methods that uses the `equals` method in it were still not able to close open goals of the proof. The `forall` constructs only introduced a lot of additional complexity. Changing the specification of the method using the following ensures clause `\result == (this == object)` and specifying that `object` is `nullable` did not do the trick either. Changing the method specification this way basically removes the equivalence relation conditions, and only checks for object references. In case the method calls to `equals` are replaced in place with `==`, methods are able to pass verification. The difference might come from the fact that the method `equals` is not specified as `strictly_pure` but just as `pure`, which places additional stipulations on verifcation. A downside of using this workaround as a solution is that methods that use the method `equals` all need to be changed for verification, and the expressiveness of the behavior is limited this way.

Similar reasoning holds for the method `compareTo` from the interface `Comparable`. This method is also provided by KeY itself, and this time has no specification at all. The method is used in the specification of the `Collection` interface, to make it possible to specify that an collection is ordered. Since – in the end – no class that uses ordering is being verified, the condition of the model field `elementsHaveOrder` has been simplified to represent `false`. Otherwise, KeY was not able to prove the specification of `elementsHaveOrder` as it seems to need more information on `compareTo` (also see the

specifications on page 35).

Exceptional behavior

This limitation is not specific to KeY and will be exposed by all verification tools that handle state changes for exceptional behavior. Just like normal behavior, exceptional behavior uses the `assignable` clause to specify changes to the heap. When inspecting an interface that explains occurence of exceptions, nothing is stated about modified heap in case an exception is thrown. Therefore, one could expect that in case of an exception the heap would stay the same. This is however not always the case, e.g., the `ArrayList` uses a variable `modCount` that counts the modifications performed on the object and is updated on every method that structurally modifies the object, even in case an exception has occured. Since `modCount` might also change on exceptional behavior, the assignable clause cannot be set to `\nothing` or `\strictly_nothing` for the exceptional behavior of these methods. To make it possible to specify the changes on the heap at interface level, either a different location set – beside `footprint` – or `footprint` itself should be provided as the assignable clause. Using `footprint` has the disadvantage that it overestimates the changes on the heap, and therefore will introduce difficulties for methods that need to use the specification. Both solutions make specifications less understandable, since one would expect exceptional behavior does not change heap locations. At least, when looking at informal specifications. However, both solutions make specifications verifiable and also improve extensibility, since `locsets` can be different for every implementation.

3.1.4. Verification approach

Several techniques are used to do verification systematically. The methods used are based on recommendations by a KeY developer and experimentation of using different settings. In general, the following approach is used. When a method behavior is loaded into KeY first *finish symbolic execution* – a proof macro – is performed, which evaluates the methods body as far as possible, i.e., depending on the proof search strategy options set, leaving a proof without executional code – if search strategy options allowed it to fully execute the source symbolically.

For this project the following search strategy options are used;

- *proof splitting* is set to *delayed*, which makes sure unnecessary proof splitting is limited, i.e., first other expansions are performed resulting in less goals in proofs.
- *Loop treatment* is set to *none*, which makes sure symbolic execution stops when a loop is encountered. This is particularly useful as loops create three proof obligations;
  - initially the invariants should hold;
  - the body of the loop should preserve the invariant; and,
  - an obligation that continues execution of the code after the loop.
- Often one of the proof obligations has problems closing automatically, as specifications are hard to get right for loop invariants, i.e., especially when you do not have a lot of experience with them. It is also possible to set loop treatment to expand, which ignores the loop invariants and expand the loop symbolically. In case a loop has a fixed number of iterations, *expand* might be the preferable option to set.
- *Method treatment* is set to use the *method contract*. It makes no sense for modular static proving to also expand methods used by the current method.
- *Query treatment* is set to *restricted*, which ensures queries in the specification are rewritten according to the method treatment, and a specific ordering on which query to treat next is used. Furthermore, specific occurences that may cause an infinite loop in the proof are not further expanded. This setting also results in less goals to prove.

Next, the proof goals that are still open are tried to close one by one, by chosing *close provable goals below* after selecting a goal. *Close provable goals below* is another macro that makes it possible to roll back the proof to the point where the macro was triggered. When all goals close the behavior and the framing is proven. It is, however, still possible that loops were not yet expanded, this is where a little interaction is needed, the loop needs to be selected and the step *loop invariant* should be chosen, after which the three aforementioned goals will replace the one goal the last step was performed on. Proving continues as described before, trying to close the goals one by one, starting with *finish symbolic execution*.

When a certain goal did not close, the goal underwent a close inspection on what particular aspect makes the goal not closable. Often this resulted in that a slight modification to the specifications was needed, but other times it was not that obvious what actually causes problems. In case it was not obvious, and the method was quite large, the

method was truncated and verification was done on a simplified method – of course updating specifications accordingly. Next, incrementally, parts were added back in, until the complete method was verified.

Sometimes it happens that proof obligations were so large that they were not able to be closed automatically. However, often large parts of the antecedent – part before the implication arrow, i.e., the assumptions – are unnecessary to proof the consequent. To make it possible to prove these larger goals as well, the part of the antecedent not needed to prove the consequent was hidden from the goal.

3.2. Example verifications

Working on this thesis, methods and dependency conditions from the classes `AbstractCollection`, `ArrayList`, `AL_Itr` and `AL_ListItr` (e.g., the `Iterator` and `ListIterator` classes extracted from `ArrayList`) underwent verification. From these methods 49 out of 54 – 91 percent – passed verification, and from the dependency conditions all 19 conditions passed verification. From the non-passing methods, the method `bachRemove` did not pass verification as it used a for each loop, not usable at the moment, as `ArrayList` did not inherit from `Iterable` directly, as was needed to be able to use the `for each` construct in KeY. A few methods from `AL_Itr` and `AL_ListItr`, namely, `remove`, `add`, `previous` and `set` were also not able to pass verification. It is not clear why exactly this is the case, but it seems to be related with possible insufficient invariants for the classes themselves, such that not enough information about heap locations is provided. That this is probably the case can be deduced from the fact that other methods from these classes had similar problems before they were able to pass verification.

Furthermore, some particular methods were harder or more interesting to get verified. The sections below will describe the harder and more interesting methods, i.e., the methods that did not close easily because they needed more interaction with KeY. The specifications the method got from interfaces above – and are relevant here – are added for convenience. The first section starts with a general remark on `nullable` values and type restrictions, which were used a lot in this project and can cause problems when not used correctly.

All specifications made for this thesis can be found on Bitbucket[2], anyone who is interested in using or improving the specifications is invited to fork or clone the repository.

[2]https://bitbucket.org/Outlay_JPtW/jml-annotations

Note that additional comments are provided that can aid in improving the specifications.

`nullable` values and type restrictions

For this project, `nullable` values were needed at a lot of locations in the specification. When a method could return, has an argument or uses an object or array of objects special care should be taken. Most of the time when you want to specify a condition on objects, you also want to make a distinction whether an object is `null` or not. However, JML defaults to having fields, arguments and return values to be `non_null`. In certain cases this is fine, but sometimes you want to be able to specify a little more. That is, for referencing arrays `non_null` is deep, which means that `non_null` can be used to state that elements from the array should not be `null` either.

For instance, the field `elementData` (an array) used by the `ArrayList` class itself should not be `null`, as this would throw exceptions at locations where they should not be thrown. However, the elements of `elementData` should allow `null` values as the `ArrayList` allows `null` elements. To make this possible, `elementData` should be specified to be `nullable`, which makes it both the array and the possible elements of the array could be `null`. Next, `elementData` itself should be restricted to not be `null`. An invariant that states this property should be put in the `ArrayList` class. Also, see the section about generics on page 76 that provides type restrictions for `Object` and `Object[]`.

`ArrayList(Collection c)`

The constructor method `ArrayList(Collection c)` creates an `ArrayList` based on a `Collection c`, using the `toArray` method to retrieve elements that needs to be added to the newly created `ArrayList`. Furthermore, the method sets the size of the `ArrayList`.

This constructor method was a bit problematic as the collection `c` is abstract and therefore nothing concrete is known about the representation for `seqLength` and `repr`, i.e., respectively the model and ghost field used for specifications by collections (as described on page 35). The specifications of `Collection` states that the method `size()` returns `seqLength`, but `seqLength` itself has no representation. This means it is not possible to use `set repr = c.repr` to update the representation. Furthermore, `repr` is used to represent the elements that are in a collection, but could be used differently when ordering is needed on some specific implementation of collection. The two invariants in `ArrayList` shown in Listing 3.3 are unable to be proved since `repr` is not known.

```
@ public instance invariant (\forall int i;
    0 <= i && i < repr.length; repr[i] == elementData[i]);
@ public instance invariant size == repr.length;
```

Listing 3.3: Invariants for `ArrayList`

To make it possible to verify the constructor method, it was needed to let KeY know how the mapping corresponds with the underlying elements. This was done by introducing a loop which updates values in the sequence representation of the `ArrayList`. Again, putting additional code in your methods is not preferable. However, there was no other way, as no specification construct that makes mapping the values to a sequence construct directly exists. The complete verified specification can be found in Listing 3.4.

```
1  /*@ public normal_behavior
2    @   requires c != null;
3    @   requires c.\inv;
4    @   ensures elementData.length == c.size();
5    @   ensures size == elementData.length;
6    @   ensures \fresh(footprint);
7    @   assignable footprint;
8    @*/
9  public ArrayList(/*@ nullable @*/ Collection c) {
10   elementData = c.toArray();
11   size = elementData.length;
12
13   //@ set repr = \seq_empty;
14
15   int i = 0;
16   /*@ loop_invariant 0 <= i && i <= elementData.length;
17     @ loop_invariant (\forall int j; 0 <= j && j < i;
18         (Object)repr[j] == (Object)elementData[j]);
19     @ loop_invariant repr.length == i;
20     @ assignable repr;
21     @ decreases elementData.length - i;
22     @*/
23   while(i < elementData.length) {
24     //@ set repr = \seq_concat(repr,
25     //@   \seq_singleton(elementData[i]));
26     i++;
27   }
28 }
```

Listing 3.4: Specification of `ArrayList(Collection c)`

Since additional specification is needed for the added while loop, understandability will decrease, as it will take a little longer to understand what is going on. A more elegant solution, making use of model methods (see the description on page 29) might be preferable in the future.

`lastIndexOf(Object o)`

The method `lastIndexOf(Object o)` has a few interesting points which made it harder to prove. The method checks, depending whether the argument `o` is `null` or an actual object, if an object is contained within the `ArrayList`. Since KeY has the construct `\indexOf`, which returns the index of an object in a sequence – when you provide it with a sequence and an object to search for – and `\seq_reverse` which you provide with a sequence and reverses the order of elements in the sequence this seemed trivial; because it could be specified like this: `\result == (\indexOf(seq, o)== -1 ? -1 : seqLength - \indexOf(\seqReverse(seq), o))`. This was not the case, using these two constructs KeY still left a few open goals, having trouble with the `\seq_reverse` construct it seemed, as very similar use without `\seq_reverse` worked for `indexOf(Object o)`. Therefore, another way of specifying the same was tried, i.e., searching for a maximum value over an array with the `\max` construct, giving `-1` if the current object is not the one being searched for, or the current index if it is. However, this expression was not able to load into KeY and an additional way had to be found. See the workaround described on page 74.

Additionaly, the tool had problems with the `equals` method as described on page 77. Replacing `equals` with `==` solved this. As a final note, the `decreasing` clause and `\forall` loop invariant were a little harder to specify, since they reversed the normal way of reasoning for loop invariants and `decreasing` clauses. The verified specifications can be found in the listing below;

```
1 /*@ public normal_behavior
2   @   requires !supportNullElements ==> (o != null);
3   @   ensures (\forall int k; 0 <= k &&
4         k <= seqLength-1; repr[k] != o) ==> \result == -1;
5   @   ensures (\exists int k; 0 <= k &&
6         k <= seqLength-1; repr[k] == o) ==> (\result >= 0
7         && \result < seqLength && repr[\result] == o);
8   @   ensures !(\exists int k; \result < k &&
9         k <= seqLength-1; repr[k] == o);
```

```
10    @   assignable \nothing;
11    @ also
12    @ public exceptional_behavior
13    @   requires !supportNullElements && o == null;
14    @   signals (NullPointerException)
15          !supportNullElements && o == null;
16    @   signals_only NullPointerException;
17    @   assignable \nothing;
18    @*/
19  public int lastIndexOf(Object o) {
20    if (o == null) {
21
22      /*@ loop_invariant -1 <= i && i <= size -1;
23        @ loop_invariant (\forall int j; i < j &&
24              j <= size - 1; repr[j] != o);
25        @ assignable \strictly_nothing;
26        @ decreasing i + 1;
27        @*/
28      for (int i = size-1; i >= 0; i--)
29        if (elementData[i] == null)
30          return i;
31    } else {
32
33      /*@ loop_invariant -1 <= i && i <= size -1;
34        @ loop_invariant (\forall int j; i < j &&
35              j <= size - 1; repr[j] != o);
36        @ assignable \strictly_nothing;
37        @ decreasing i + 1;
38        @*/
39      for (int i = size-1; i >= 0; i--)
40  // if (o.equals(elementData[i]))
41        if(o == elementData[i])
42          return i;
43    }
44    return -1;
45  }
```

Listing 3.5: Specification of `lastIndexOf(Object o)`

Since understandability increases when similar methods also have similar specifications, the method `indexOf` has been changed accordingly, i.e., also using three `ensures` clauses to specify the minimum index that should be returned. Of course, changing the last `ensures` clause with one that checks there does not exist an index with a lower index –

instead of a higher one. Again, model methods might increase understandability when they are available. For verification itself, difficulty decreases, e.g., less steps are needed to prove the goal, and the obligation splits in smaller proof obligations.

`set(int index, Object element)`

The method `set`, which sets an element on the given index of a list and returns the element on that index before the invocation illustrates the way sequences are used. Namely, by using concatenation. A single element is added by first making a singleton sequence from it and next concatenating it with the old elements. The old elements are selected by making two subsequences from the original sequence; one subsequence for the part before and are for the part after the index that needs the element to be replaced. Naturally, you want this update to be done as follows; `set repr[i] = element`, which is however not the way sequences work. That it does not work this way is because `repr[i]` is basically a shortcut for the statement `\seq_get(repr, i)`, which returns the element at `i` for the sequence `repr`, whereas `repr[i] = element` would mean `repr` is an array that updates index `i` with `element`. Sequences work this way because sequences are pure, and sequences are a stateless data type that operate by value only. The verified specification can be found in Listing 3.6.

```
1  /*@ public normal_behavior
2    @  requires \typeof(element) == \type(Object);
3    @  requires index >= 0 && index < seqLength;
4    @  requires !supportNullElements ==> (element != null);
5    @  ensures repr[index] == element;
6    @  ensures \result == \old(repr[index]);
7    @  ensures \new_elems_fresh(footprint);
8    @  assignable footprint;
9    @ also
10   @ public exceptional_behavior
11   @  requires index < 0 || index >= seqLength || !setSupported
12         || (!supportNullElements && element == null);
13   @  signals (IndexOutOfBoundsException)
14         index < 0 || index >= seqLength;
15   @  signals (NullPointerException) !supportNullElements
16         ==> element == null;
17   @  signals (UnsupportedOperationException)
18        !setSupported ==> true;
19   @  signals_only UnsupportedOperationException,
20         ClassCastException, IllegalArgumentException,
```

```
21          IndexOutOfBoundsException, NullPointerException;
22    @  assignable \nothing;
23    @*/
24  /*@ nullable @*/ Object set(int index, /*@ nullable @*/ Object element);
25    rangeCheck(index);
26
27    Object oldValue = elementData(index);
28    elementData[index] = element;
29
30    /*@ set repr = \seq_concat(
31          \seq_concat(\seq_sub(repr, 0, index), \seq_singleton(element)),
32          \seq_sub(repr, index + 1, seqLength)
33        );
34      @*/
35    return oldValue;
36  }
```

Listing 3.6: Specification of `set(int index, Object element)`

Although it is logical that setting values on an array-like-style does not work for sequences, understandability of specifications decreases because of this fact. Model methods might improve understandability, e.g., a model method `seqSet(sequence, int, object)` would be clearer and the implementation could be the same – or very similar – to the specification above which uses `\seq_concat`.

`finishToArray(Object[] r, Iterator it)`

The method `finishToArray(Object[] r, Iterator it)`, which can be found in the class `AbstractCollection` is a helper method which will – given an `Object[] r` and `Iterator it` – make sure the array `r` will be updated with the final values that `it` still has to provide for it. This method was particularly hard to get verified since it used several methods that needed to be correct as well before it was possible to get this method verified, i.e., `hasNext` from iterator, `copyOf` from the `Arrays` class and `hugeCapacity` from `AbstractCollection` itself. Note that, in the listing below the class `Arrays` is called `SelfArrays`. This change has been made to know KeY was actually using the correct specifications for the `copyOf` method and not silently used its own ones.

The hardest part was to get the `loop_invariant` specification correct. Initially it was even not possible at all, since it was unclear KeY could not use the `\old` construct to

refer to the arguments of the method too. Trying to automatically solve the obligations by KeY resulted in a freezing system due to memory consumption. After finding out the current approach was not working because of the `\old` construct, additional local variables were introduced that could aid with the verification. After providing enough `loop_invariants` KeY was able to proof the method. The implementation of `finishToArray` with specification used for the verification can be found in Listing 3.7.

```
1  /*@ public normal_behavior
2    @   requires r.length + (it.seqLength - it.position + 1)
3    @     < MAX_ARRAY_SIZE;
4    @   requires it != null && it.\inv;
5    @   requires r != null;
6    @   requires \disjoint(it.footprint, r[*]);
7    @   ensures \result == r || \fresh(\result);
8    @   assignable it.footprint;
9    @*/
10 /*@ helper @*/ private static /*@ nullable @*/ Object[]
11   finishToArray(/*@ nullable @*/ Object[] r, Iterator it) {
12
13   int i = r.length;
14
15   Object[] tmp_r = r;      /// added
16   int cap = tmp_r.length; /// added
17
18   /*@ loop_invariant \fresh(tmp_r) || tmp_r == r;
19     @ loop_invariant tmp_r != null;
20     @ loop_invariant i <= cap && cap <= tmp_r.length && i >= r.length;
21     @ loop_invariant i < MAX_ARRAY_SIZE;
22     @ loop_invariant it.collection.footprint ==
23         \old(it.collection.footprint);
24     @ loop_invariant it.\inv;
25     @ assignable it.footprint;
26     @ decreases it.seqLength - it.position;
27     @*/
28   while (it.hasNext()) {
29     cap = tmp_r.length; /// r -> tmp_r
30     if (i == cap) {
31       int newCap = cap + (cap / 2) + 1;
32       // overflow-concious code
33       if (newCap - MAX_ARRAY_SIZE > 0)
34         newCap = hugeCapacity(cap + 1);
35       tmp_r = SelfArrays.copyOf(tmp_r, newCap); /// r -> tmp_r
```

```
36      }
37      tmp_r[i++] = (Object) it.next(); /// r -> tmp_r
38    }
39    // trim if overallocated
40    return (i == tmp_r.length) ? tmp_r : SelfArrays.copyOf(tmp_r, i);
41  }
```

Listing 3.7: Specification of `finishToArray(Object[] r, Iterator it)`

`remove(int index)`, `add(int index, Object element)` and alike

Some methods, i.e., `remove(int index)`, `add(int index, Object element)`, `addAll(Collection c)`, and `addAll(int index, Collection c)` which all make structural changes to the `ArrayList` object were hard to prove initially. The methods have in common that they all make use of the method `arraycopy`. That they were so hard to prove came from the fact that after returning from one of these methods – of course – the invariants still needed to hold. Most of the invariants were easily proved. However, the following invariant was problematic;

```
1
2 (\forall int i; 0 <= i && i < repr.length; repr[i] == elementData[i])
```

Listing 3.8: Problematic invariant

The invariant states that the ghost field `repr` – a representation sequence of the `ArrayList` object – has to hold the same values as the underlying array `elementData`. Below the difficulty will be explained based on the `add(int index, Object element)` method which adds an element to the given `index` of the `ArrayList` and makes sure that all old elements starting from `index` move an index further. Since other specifications of the method are at this point not relevant they are left out in the listing below;

```
1 public void add(int index, Object element) {
2   rangeCheckForAdd(index);
3   ensureCapacityInternal(size + 1);
4   SelfSystem.arraycopy(elementData, index, elementData, index + 1, size -
        index);
5   elementData[index] = element;
6
7   /*@ set repr = \seq_concat(
8         \seq_concat(\seq_sub(repr, 0, index), \seq_singleton(element)),
9         \seq_sub(repr, index, size));
```

```
10      @*/
11
12    size++;
13  }
```

Listing 3.9: The method `add(int index, Object element)`

The `set` specification updates the sequence `repr` by adding `\seq_singleton(element)` to the sequence. At the moment this is done by concatenating the part of the sequence before `index`, the element that should be placed, and the part after `index`.

In case the given `index` was equal to the size of the `ArrayList`, there was no problem, since this is basically the same as the `add(Object element)` method, which was already proven. However, in case `index` has another value, the method `arraycopy` seemed to prohibit KeY from closing all goals. A close inspection of the goals revealed a problem, updating of the `elementData` array was not going as expected, i.e., instead of using an old and new state of `elementData` for updating each index, the current state was used for both sides of the update statement. The problem had to be at the specification of `arraycopy`, which was indeed the case. Listing 3.10 shows the `normal_behavior` of the `arraycopy` method (without the `requires` clauses) before and after the correction.

```
1  // BEFORE CORRECTION
2  /*@ public normal_behavior
3    @   ensures (\forall int i; 0 <= i && i < length;
4          dest[destPos + i] == src[srcPos + i]);
5    @   assignable dest[destPos .. destPos + length -1];
6    @*/
7  native public static void arraycopy(/*@ nullable @*/ Object[] src, int
       srcPos,
8    /*@ nullable @*/ Object[] dest, int destPos, int length);
9
10 // AFTER CORRECTION
11 /*@ public normal_behavior
12   @   ensures (\forall int i; 0 <= i && i < length;
13         dest[destPos + i] == \old(src[srcPos + i]));
14   @   assignable dest[destPos .. destPos + length -1];
15   @*/
16 native public static void arraycopy(/*@ nullable @*/ Object[] src, int
       srcPos,
17   /*@ nullable @*/ Object[] dest, int destPos, int length);
```

Listing 3.10: The method `add(int index, Object element)`

After correction the `\old` construct is used to specify the updated elements. Since initially the idea had not come to mind that `src` and `dest` arrays could be the same, this was not taken into account before.

Note that the method `arraycopy` is native and therefore verification of `arraycopy` itself is not possible, otherwise, verification of the method had shown the problem earlier. It is, however, possible to provide a (simple) Java implementation and verify this implementation.

Solving this problem however, was not all that was needed to let KeY close all open goals. It seemed KeY has some difficulties closing goals automatically that make use of `forall` and `exists` constructs. In order to close these goals the user has to instantiate these `forall` and `exists` for the current goal with the right variables that occur in the goal. The variable used for instantiation should be taken from the consequent part of the open goal, and next instantiate a `forall` in the antecedent of the goal. This is not always straightforward, and due to large proof obligations it might still take a while before KeY is able to close the goal.

`repr` as `model` vs. `ghost`

To see what difference it makes to specification and verification, `repr` – a sequence used to map the `ArrayList` object on – has been declared both as a `model` and as a `ghost` field. Two minimalistic classes based on the `ArrayList` class are created that only contain a constructor method, `size` and `add(Object o)`. Appendix A provides the implementations as well as the specification for the method `copyof` from `Arrays` which both implementations use.

For specification the most obvious difference between the implementations are that the model field needs only a `represents` construct to specify the representation, while the `ghost` fields need to be updated with the JML `set` construct for every change on the elements the list contains. Further `ghost` fields also need additional invariants to state what the elements of the sequence should hold, and what the length of `repr` is. This means that the `model` representation is a lot more brief, which will increase understandability of the specification. Furthermore, this means that when extending a class with additional methods, effort is needed only for the implementation using the `ghost` field.

However, doing verification using `\such_that` to make a model field binding causes difficulties. Every time a method changes `repr` the user should provide a representation –

in this case of a sequence – that corresponds with the update in the method body, essentially a `set` statement but in the proof and not the code. For the implementation that uses a `ghost` representation KeY was able to close all goals automatically. Whenever a method makes a more complex change which also affects `repr`, giving a representation for the `model` field will also be more difficult. However, when using a `ghost` representation, verifications also gets more difficult as can be seen in the previous section using sequences on page 89.

Findings

Undergoing verification, several specifications stood out that complicated verification, i.e., invariants or ensures clauses that used `\forall` and `\exists` constructs, as well as `model` fields specifying a binding with the `\such_that` construct. In case `model` fields could use the direct `represents` construct, they are easier to use than `ghost` fields, i.e., verification would not be any different, but specifications are much shorter which makes understandability better. For extensibility, `model` fields are preferred over `ghost` fields, since for `ghost` fields it is necessary to include specifications in the body of a method to preserve invariants. The use of `sequences` for specification provides a nice addition to former specification styles, in that they can be used to extend the state of an object, or used to provide a shorter representation. Therefore, `sequences` are considered to improve extensibility, however, understandability could at this point be improved as only minimal operations can be performed on `sequences` which causes the specifications to be longer.

Pure methods can also be used to specify method behavior in certain cases, which might be preferable for understandability. For example, after some method it might be necessary to ensure an element is part of a collection. This can be done by using a pure method `contains`, but also using an expression or a model method. The downside of using a pure method opposed to another representation lies in verification, whereby lookup is needed and either unfolding of the method or applying a corresponding contract needs to be performed. In case there are multiple contracts, this leaves a lot of additional goals in the proof tree, some of which might be hard to close.

Finally, model methods – although not yet supported – aid in keeping specifications understandable, i.e., they can provide complex expressions and next be used by their names in method contracts. The downside is that people should be familiar with the concept of model methods, and verification might be horrible since it is yet one further

abstraction compared with model fields, however, this is only speculation.

A final note, the findings here are all based on the workings of KeY, which means verification might differ when using another static verification tool. The findings about understandability might also differ for model methods and abstract data types (e.g., sequences), since these may or may not be implemented by other tools, and if they are implemented, they might be implemented in a different manner.

Part III.

Evaluation

4. Evaluation

This chapter describes the approach taken to get more insight in understandability of the specifications made, and understandability, extendibility and verificability for different specification constructs that can be used in JML* in general. First the road to the final questionnaires will be explained in Section 4.1, after which the design of the two distinct questionnaires will be explained in the remainder of this chapter.

4.1. Road to the questionnaires

First, understandability of specification and programming languages in general has been investigated. Barros et al. [12] found that for the Object Constraint Language (OCL [41]), which is a different formal specification language, understandability decreases when nested quantifiers are used, several constructs are used to describe the same behavior, or distinctions in specifications are made based on the actual type of, e.g., the parameters of a method. These results were found after first indicating possible constructs that might decrease understandability, and next doing an experimental study that evaluated the usefulness of refactoring these constructs. The results of the study indicated that the former constructs have a negative impact on both correctness and the time necessary to understand constraints written in OCL. Of course, OCL is not the same as JML*, but for the specifications done for this project the stumbling blocks for OCL understandability have been avoided here as well. In [20] two rules are stated about understandability for programming languages, i.e., smaller languages are easier to understand than bigger languages, and closely related languages are easier to understand than distantly related languages. The second part makes a lot of sense for JML as method contracts are designed to be familiar to Java programmers (see also Section 1.2.1 on page 9). However,

both rules cannot be used to derive statements about JML* specifications. Keeping specifications minimal could help, but that is not supported by any research. Misra and Akman [30] indicate that complexity and therefore understandability of code in an object oriented language depends on the architecture of methods and attributes in this code, and that time and effort spent on understanding code plays an important role in understandability.

To get more insights in understandability of the specifications made for this thesis, two approaches come to mind. A first approach is performing a questionnaire about part of the verified methods and ask people whether methods should indeed be able to pass verification. Of course, to make it more interesting also a few modifications should be inserted that do not pass verification. For each method the participants should motivate their answer, i.e., especially in case a participant has the feeling a method should not pass verification. The second approach would be to let a group of participants extend part of the specifications made during this project for a different implementing class, e.g., a `LinkedList` and let them answer questions on what parts are more difficult and which are easier. This way – since being able to extend some specification you first need to understand the original specification – one is able to determine whether the original specifications are understandable. Both of the approaches have advantages as well as disadvantages. The former approach cannot provide information whether all specifications are correct, whereas the second approach needs participants that are – at least – familiar with JML* to the extent of being able to work with it. The advantage of the first approach is that people familiar with the basics of JML already could provide useful feedback on understandability, which is a far larger group than people familiar with JML*. Furthermore, to make it more likely that people are willing to participate, it should not take too long to complete the questionnaire/assignment. Therefore the choice for the first approach has been made. The design of the understandability questionnaire is described in Section 4.2. Since participants should not have more than a basic knowledge of JML, students that followed the course *System Validation* at the University of Twente – which among other things teaches students to specify Java with JML – are chosen as a target audience.

As JML* leans itself to use different specification styles to specify the same behavior – using ghost fields, model fields, model methods, pure methods and abstract data types – this is also an interesting aspect where understandability, extendability and verifiability can differ. Taking an additional questionnaire could give more insight on these aspects. Since this time, specific constructs for specification are addressed, and

some of them are KeY-specific, as target audience the KeY developers are selected. For all aforementioned constructs, opinions on understandability, extendibility and verifiability should be retreived. The design of the questionnaire is described in Section 4.3.

Both questionnaires were available from Monday 29th of July until Saturday 10th of August 2013 and participants were addressed by e-mail. For the understandability questionnaire this means personal addressing to 70 students that participated in the System Validation course in the year 2012. For the specification styles questionnaire an invitation was sent to a central KeY developers e-mail address. Halfway a reminder for participation was sent. The goal was to get at least 20 per cent participation from the invitations.

Since both questionnaires were taken in a holiday period, additional measures were needed to get enough participants. By the end of the second week the questionnaires were open 7 (out of about 20) people participated in the specification styles questionnaire and only 5 (out of about 70) people particiated in the understandability questionnaire. For the specification styles this meant the goal of 20 per cent was reached, however, the understandability questionnaire still needed nine participants to reach the goal. Since a lot of people were probably on vacation, the deadline of the understandability questionnaire was extended by a week and a more personal approach to obtain participants was used, which helped to get enough participants. Another problem with the understandability questionnaire is that people who followed the System Validation course were asked to fill out the questionnaire. Some of these people might have graduated already and therefore they possibly cannot use their e-mail accounts from the university anymore.

4.2. Understandability questionnaire

For the understandability questionnaire, seven methods are provided in an accompanying file with JML* specifications attached, also found in Appendix C. The questionnaire asks for each of the methods whether it should pass verification or not, with an additional field to provide a motivation why a method did not pass, or a comment in case it passes. This way it is also possible to express doubt or to tell that a specification is hard to understand. This was also mentioned in the description of the questionnaire. The questions are ordered in such a way that the difficulty of the questions increases, i.e., shorter specifications or specifications that do not use other methods come prior to

larger specifications or specifications that use other methods to describe the behavior. At the end of the questionnaire it is asked to fill out time spent. The methods used for the questionnaire represent the general specification style used throughout the project, and are only slightly modified to include only minimal specifications/methods needed for the questionnaire. The questionnaire should take around 15 minutes to complete. The participants are asked not to be concerned about coding style or having multiple classes in one file. Furthermore, potential overflow of integer values, as well as `OutOfMemoryException`s need not to be considered for the questions, and the participants may assume the correctness of the `copyof` method (see Appendix C). Additionally it is given that the specifications load succesfully – there are no syntax errors – and minor changes were made to make some specifications fail verification, shortly to be described. Listing C.1 shows the file provided to the participants for answering the questionnaire. To make it easier for participants to see which method and question belong together, each method is preceded by a comment.

Before changing any of the specifications, it has been checked that all methods are still verifiable. After which two minimal changes were done to the specifications for method 4 (`clear`) and method 7 (`indexOf`). For method 4 the loop invariant has been changed from `(\forall int j; 0 =< j && j < i; elementData[j] == null)` to one leaving out the specification for the 0-index, additionally a post-condition that could not be satified is added, i.e., one that states all elements in `elementData` should be `null` afterwards. For method 7 the second post-condition is changed; a negation operator is added in front.

4.3. Specification styles questionnaire

The specification styles questionnaire is divided in a three page questionnaire, and is designed to take about 10 minutes to complete. The first two pages consider only questions regarding the specifications, whereas the third page considers verification. For the different specification styles distinguished for this research, i.e., using ghost fields, model fields, pure methods, abstract data types (e.g, sequences), abstract predicates (model methods), it is asked whether understandability, extensibility and verifiability improves or not. Participants may choose among a five-point scale or give the answer *not applicable*, and are asked to motivate their answer. Answers should all be based on the participants experience working with verification tools until the questionnaire.

The first page asks about perceived understandability of different specification styles. Participants should compare several styles with the general (perceived) understandability of specifications. The five-point scale for the answers on this page range from *far better* to *far worse* and have a neutral answer (*no difference*) in the middle. An example question looks as follows; *'Ghost fields' make understandability of specifications..*, after which a dropdown menu for the answer is provided and a field for motivation can be found.

The second page asks about perceived extensibility of different specification styles. Participants should compare several styles with the general (perceived) extensibility of specifications. Extensibility is explained as follows; when different specification styles are used for, e.g., some basic implementation class, how well can they be used by an extending class of this basic implementation. The same five-point scale as on the first page is used. The questions are similar to the ones on the first page, but this time state *extensibility* instead of *understandability*.

The third and last page of the questionnaire asked about perceived difficulty of verifications using the different specification styles. Of course, answers should again be given based on the participants experience working with verification tools until the questionnaire. This time the answers on the five-point scale range from *much easier* to *much harder*. The questions differ only little from the other pages and are similar to; *'Ghost fields' make verification..*, after which a dropdown for the answer is given.

With the answers of the participants it is possible to (in)validate findings described earlier in this thesis, and give possible (additional) explanations why specific specification styles are better for certain criteria than others. The questions for this questionnaire can also be found in Appendix B.

5. Questionnaire

This chapter provides and dicusses the results of the questionnaires described in the previous chapter. First Section 5.1 goes through and discusses the results of the understandability questionnaire. Next, Section 5.2 goes through and discusses the results of the specification styles questionnaire.

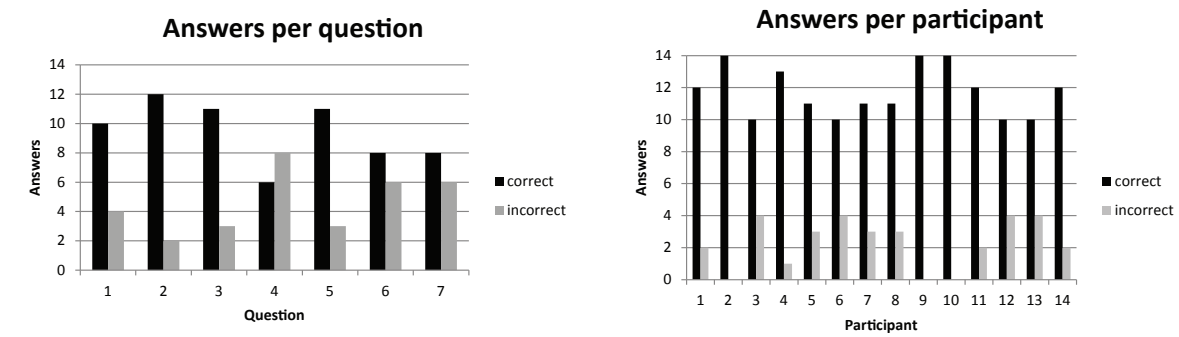
5.1. Understandability questionnaire

At the end of the extended period, 14 (out of 70) people participated in the understandability questionnaire, which means the goal of 20 per cent has been reached just. Below the results are described, followed by a discussion. For each question there is a corresponding method. Question `x` asks whether method `x` should pass verification. The methods can be found in Appendix C.

Results and discussion

Figure 5.1a provides the answers given to the understandability questionnaire questions that asked about whether verification would pass or not for seven methods. Not whether the answer *yes* or *no* was given is graphically represented, but if the answer was indeed correct. An average of 2.3 wrong answers was given, which means just over 70 per cent of the questions are correctly answered. The correct and incorrect answers provided by each participant are represented by Figure 5.1b.

Although mentioned in the introduction of the questionnaire that adjustment were made to falsify one or more of the method specifications, two participants answered that all



(a) per question (b) per participant

Figure 5.1.: Provided answers

methods would pass verification. As stated in the previous chapter, adjustments are made such that the methods for question four and seven would not pass verification.

In the sections below for every question the given answers and motivation are provided, followed by a short discussion about the question. Finally, the results of the time spent is provided and dicussed.

Question 1

Four participants answered that the constructor method – method for question 1 – was not correct. Three out of four motivations stated that either the exceptional behavior was not covered, or specifications were incomplete. One participant motivated that `size` would be undefined, and therefore the method would not pass verification.

It is interesting that participants answered that there was no exceptional behavior or specifications were incomplete, and therefore answered the specification was incorrect, as up front the question 'is the specification valid, considering the implementation based on your *experienced intuition*?' was asked. The motivation that `size` would be undefined, could be explained as that the participant either did not know Java would set default values, or otherwise an assumption that annotations should be added to also specify this behavior of Java.

Question 2

The method for question 2, which retrieves the element at a certain index is falsely assumed not verifiable by two participants. Both answers are motivated by the statement

that `seqLength` does not necessarily has to have the same value as `size`.

Since both times the same motivation is provided, it is possible that the participants did either not understand model fields, or did not find the specification that stated `represents seqLength = size`.

Question 3

For the third question, it is asked whether the method `trimToSize` would pass verification. Three participants incorrectly answered that verification would not pass. Motivations provided are; uncertainty of the post-condition `repr == \old(repr)` for the first behavior as `repr` was not understood, wrong requires clause as a participant believes it is only possible to have `size < elementData.length`, or a not understood specification of `copyOf`.

The motivation that `repr == \old(repr)` could not hold afterwards is interesting as the specification would not have been needed in the first place, i.e., the assignable clause does not state that `repr` can change. Since no further explanation is given, it is not clear what exactly makes the participant assume this post-condition is incorrect. The second motivation basically states that the second behavior is not needed, or unreachable, which would falsify the verification of the method, that is not the case, however. Why the `copyOf` method is hard to understand is not explained, it might be that the specification is too long.

Question 4

The fourth method (`clear`) should not pass verification, six out of eight participants indeed identified that there was a problem with the specification. From the participants that thought the specifications are just fine, one participant explained it was unclear what `decreasing size - i` means. Another participant does not get the point why the post-condition states that all elements in `elementData` are `null`, but thinks it is fine anyway. A third participant answered that the loop invariant should be different and include the 0-index element. Several motivations were given in case a participant indicated the method would not pass verification. One participant described both problems, i.e., it cannot be proven that all elements of `elementData` are `null` afterwards, and the loop should also include the 0-index element. The other participants all indicated one of the two problems.

Although two participants gave the right answer why the method should fail, they indicated that the method should pass verification anyway, which is strange.

Question 5

For the fifth question, it is asked whether the method `set` would pass verification. Three participants incorrectly answered the method would not pass verification. One participant thinks the method should not pass in case `\old(size) < \old(seqLength)`, another participant believes `elementData` should be assignable, and a last motivation states the `set repr` specification is not understood.

The first motivation basically boils down to the representation of `seqLength`, which is either overlooked or not understood – by the same person as before. Furthermore, the idea of having a footprint is not completely clear, as `elementData` is part of the footprint and therefore assignable. Also (at least) one of the operations on sequences is not clear, while `\seq_empty` did not seem to cause any problem for the participants.

Question 6

Question 6 asked whether the method `add` should pass verificaton, six out of 14 participants thought it did not, which is however not the case. Several participants indicated the assignable clause is incorrect and should contain `elementData` and `elementData[*]`. Again, the same participant answered that the combination of using `size` and `seqLength` will give problems, and therefore the method will not pass verification. Also, the same participant that answered the set construction for `repr` was not understandable in question 5, gave the same motivation for question 6.

Clearly, not all participants understand how the `\locset` is used to define a footprint. Additionally, sequences and model fields are not completely understood either.

Question 7

The question about the last method, i.e., `indexOf` where the method was modified and does not pass verification anymore was correctly identified as having an incorrect specification by six participants. Actually eight participants answered the method would fail verification, but two participants gave an incorrect motiviation. One of the motivations stated that `size` and `seqLength` would cause problems, and the other motivation

that not enough is known about `repr`. The correctly answered (and motivated) questions stated that having the exclamation mark in front of the second ensures clause fails verification, which is exactly the case.

Time spent

Participants were asked upfront to record time spent on the questionnaire. Most participants needed more than 20 minutes as indicated in Figure 5.2. Six out of 14 participants needed less time to complete the questionnaire.

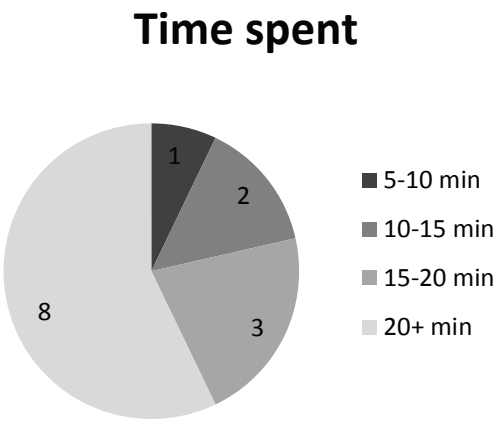


Figure 5.2.: Time spent distribution

In Table 5.1 it is indicated how much time it took each participant to fill out the questionnaire, furthermore it is indicated which of the provided answers are incorrectly answered by each participant.

From this table, it seemed that spending more than 20 minutes trying to correctly answering the questionnaire contributed to having a better understanding of the questionnaire. Two exceptions might be participants 1 and 14 who basically answered all questions with *yes*. Although it seemed they understood the specifications, it might also be the case that they only quickly scanned the methods for possible flaws, and could not find any. However, some of the specifications are JML* specific, not thought at the System Validation course, which makes it doubtful whether the participants were seriously participating.

participant	time spent	incorrect answers provided
1	10-15	4, 7
2	20+	-
3	10-15	3, 4, 6, 7
4	20+	7
5	15-20	4, 6, 7
6	20+	2, 4, 5, 6
7	20+	1, 6, 7
8	20+	1, 3, 4
9	20+	-
10	20+	-
11	20+	1, 4
12	15-20	1, 4, 5, 6
13	15-20	2, 3, 5, 6
14	5-10	4, 7

Table 5.1.: Time spent and provided incorrect answers

Some of the participants explained, either by the questionnaire or in person, that they spent plenty more time than they could indicate via the answering field of the questionnaire, namely 30 to 35 minutes. This gives the idea, that when one is more familiar with JML specifications, one is also quicker at understanding them. An estimated time of 15 minutes was given for filling out the questionnaire, which was obviously not enough for most of the participants. Furthermore, some of the people that were asked to participate gave the answer they were not familiar with JML or that they did not finish the System Validation course yet, and therefore were also not able to participate. Furthermore, some of the motivations contain a remark that the difference between `\nothing` and `\strictly_nothing` was not clear. The requirements `e.\inv` and `\typeof(element)` was also not understood by all participants. However, differences between in purity, invariants as a single group and the `\typeof(element)` construct are not taught on the System Validation course. This tells two things, i.e., an abstract memory model is difficult to understand, and in general abstraction is not easy.

The possibility that participants were not participating seriously, together with the fact that filling out the questionnaire took more time than expected, as well as a few points indicated by the participants that were not clear, i.e., the relation of `seqLength` and `size`, assignable clauses using a footprint, and sequences, give an indication that specifications using JML (or JML*) are not really understandable for people only just starting with JML. This is at least the case when JML specifications are used for any

substantial Java class, i.e., the specifications here could at some points be a little harder than they might be in general when being taught, however, for any substantial Java class, specifications might and probably will be much more complicated.

5.2. Specification styles questionnaire

For the specification styles questionnaire, seven members of the KeY development team participated. Since the questionnaire is basically divided into three parts, there are three subsections, i.e., one for the understandability, extensibility and one for verifiability. Each of these sections is divided in two parts again, each providing the answers the participants gave, and a reflection.

Understandability

The first part of the questionnaire asked about influence of JML* specifications on understandability. Figure 5.3 shows for several specification styles the answers given on perceived improvement or worsening for understandability by the expert users. The figure shows that abstract data types (e.g., sequences) score better on understandability. Motivation given by the experts is that abstract data types allow to abstract away from actual Java representations, which makes reasoning and problem understanding easier, and in case of linked structures, technicalities are removed. The only downside given is one has to be familiar with the concept.

Besides abstract data types also model methods, pure methods and model fields are considered generally positive regarding understandability. Model fields are declaritive expressions which can be used to abbreviate complex structures and repeating expressions, and also have less overhead compared to ghost fields according to the experts. Pure methods make it possible to query anything that is also possible in Java itself, while guaranteeing no state changes have occured. Model methods or abstract predicates can be used to avoid complicated expressions that are used repeatedly and express the same thing. Furthermore, they make it possible to combine the advantages of model fields, abstract data types and pure methods. A possible downside given is that when they are used a lot they can be overwhelming. In case the expert users chose *not applicable* this was because that they did not use model methods themselves, or because they

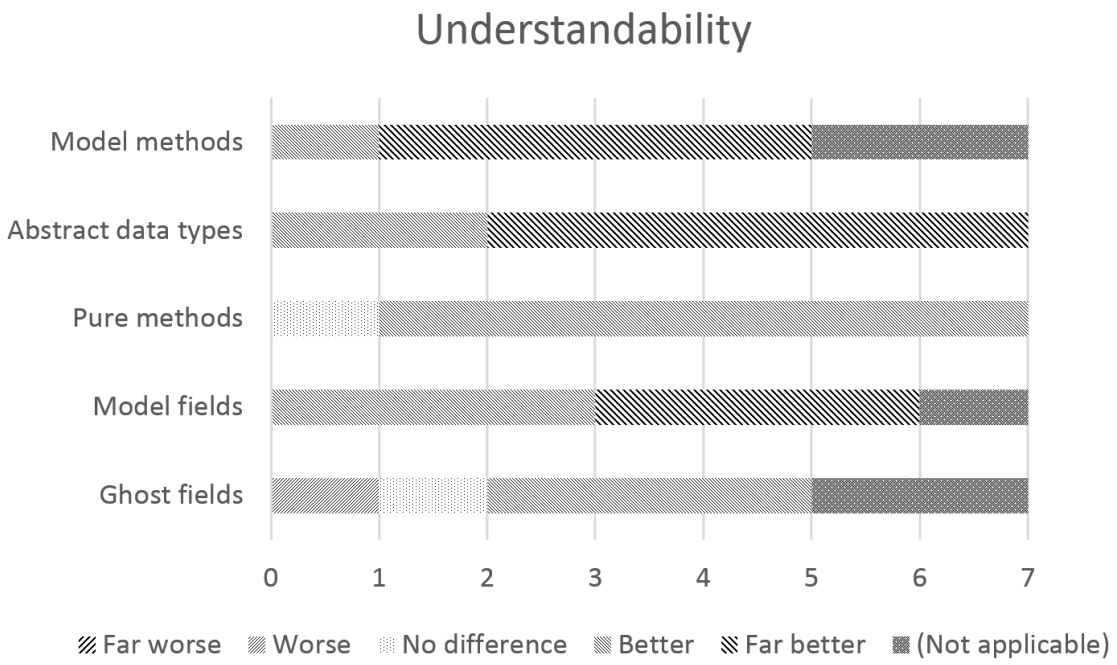


Figure 5.3.: Understandability results

were not sure if they would improve understandability. However, the overall motiviation given was still positive.

Regarding ghost fields participants were less positive. Their motivation is that additional elements are needed in specification, the specification has to be kept consistent with executed code, and specifying this way is error prone. As a positive motivation, the fact that ghost fields aid in formulating facts explicitly that are only implicitly in the code is given. Furthermore, the motivation that the usage is more like actual programming is given.

Reflection

It seems the expert users have a similar conception as described in the last chapters. Ghost fields being a little harder for understandability as additional measures have to be taken, i.e., updating the state for any method that is linked with the ghost field, whereas other specification styles do not have this need. Something not mentioned in the motivation explicitly is the need of invariants to restrict the ghost fields somehow.

Also, pure methods are a little lower on the understandability scale than model fields

and abstract data types here, as they possibly provide distinguishing behavior based on, e.g., the arguments of a method, whereas the other specifications are stricter.

The motivation given for model methods make it sound logical they would increase understandability for specifications, however, this is something that still needs to be checked later on when model methods are completely supported.

Extensibility

The second part of the questionnaire asked about influence of JML* specifications on extensibility. Figure 5.4 shows for several specification styles the answers given on perceived improvement or worsening for extensibility by the expert users. Compared to the understandability results, the results given by the experts are very similar, but slightly less positive for all specification styles. Ghost fields are considered worse than other specification styles for extensibility. However, the results are a bit misleading, i.e., considering the given motivations. One of the experts has only little experience with extensibility of JML* and therefore answered all questions with *not applicable*. Another participant did not get the idea that any of the specification styles contributes to better or worse extensibility, and therefore answered all questions with *no difference*.

Starting with ghost fields, the argument of having to update ghost fields in the bodies of methods of extending classes is given as argument for worsening extensibility. The argument given in favor of ghost fields is that they make it possible to extend the state of a class. Additionally one participant answered this question with not applicable, with a motivation of having no experience with it.

For model fields the arguments given for improved extensibility is that model fields provide abstraction from state, and there is no need to worry about implementation details. Furthermore, it is possible to specify properties about a model field already at the interface level whereafter implementing classes can indicate the way a model field represents a certain property. As a disadvantage a participant indicated represents clauses of model fields may be overriden in JML.

The argumentation given for pure methods is that they abbreviate from object state like model fields, whereby additionally implementations can be overriden by subclasses.

Abstract data types are considered better for extensibility since higher abstraction is in favor of extensibility, and only the connection between an abstract data type and source needs to be established for an extending class. Some participants indicated doubts in the

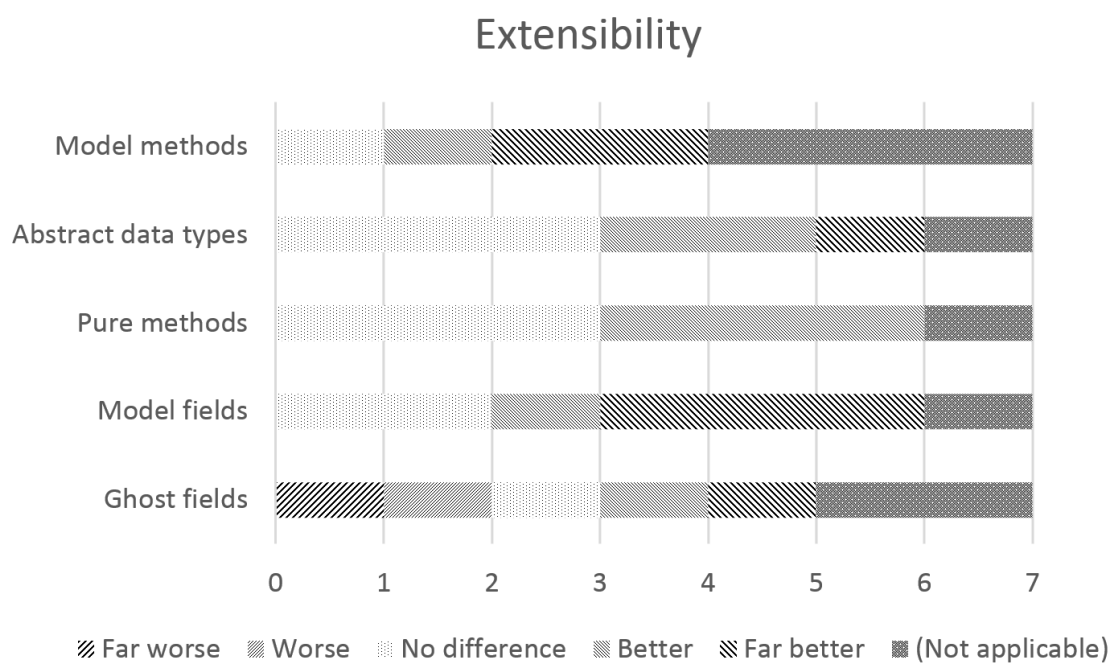


Figure 5.4.: Extensibility results

motivations, since implementing classes of some interface may need different abstract data types, e.g., a set or some ordered queue implementing a collection might need a different abstract data type.

Since model methods are not yet part of KeY two more participants answered with *not applicable*, as they had no experience with model methods. The remaining two answers are considerably positive based on expectations. A motivation given is that one idea behind model methods is to bring full specification inheritance that has the same flexibility as implementation inheritance, which should clearly make them extensible.

Reflection

Although participants provided similar motivation for some of the answers for the same question, the answers were different. This might be related to how much participants actually use a specific specification style. Using one specification style (more) often might give the perception one style is better in a certain aspect than another.

An interesting motivation is given for the question about how much model fields improve or worsen extensibility, whereby it is stated that represents clauses of model fields may

be overridden, which actually caused a loading problem in KeY during the project. It might be the case that this was actually possible at a certain version of the tool.

From the answers given for abstract data types, it seems that it might be the case that is not just the usage of abstract data types, but more likely the context in which they are used that affects extensibility. This would explain the motivation of providing a connection between abstract data types and source that improves extensibility, which would be the case when using an abstract data type together with a model field.

The overall result of ghost fields being the least extensible specification construct was also the perception during the project.

Verifiability

The last part of the questionnaire asked about influence of JML* specifications on verifiability. Figure 5.5 shows for several specification styles the answers given on perceived difficulty of verification by the expert users. This time, model fields scores worst and abstract data types scores best.

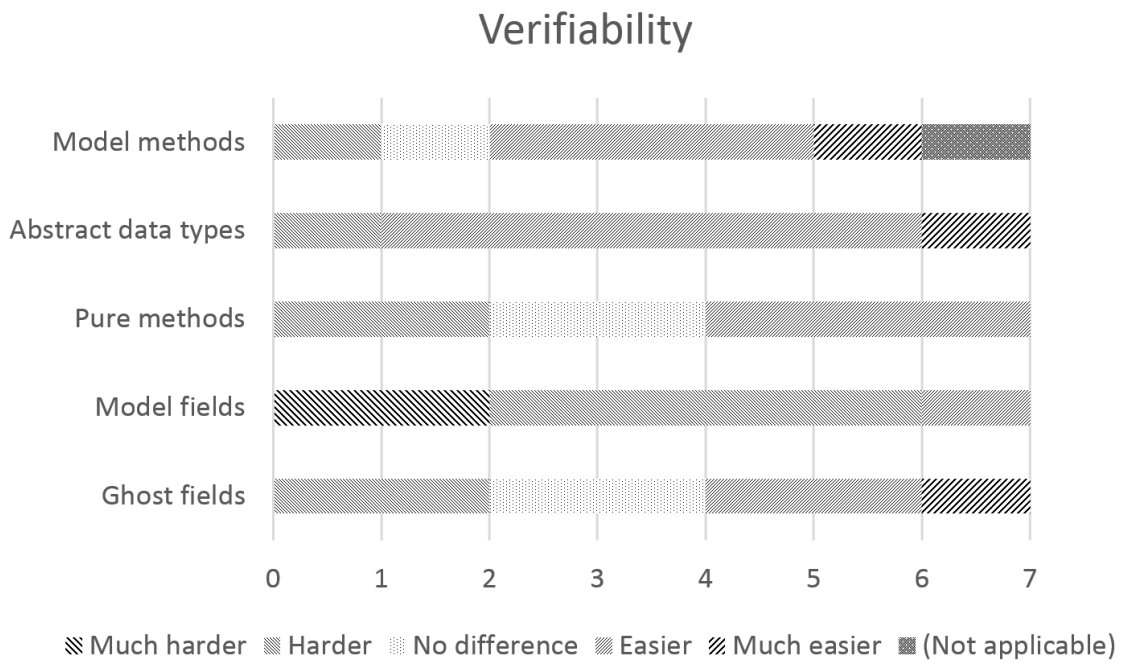


Figure 5.5.: Verifiability results

To start with abstract data types, the expert users were quite positive regarding ver-

ifiability of abstract data types. However, verification might become harder, which is motivated with the argument that another entity is introduced in the proof. Motivation for easier verification explains that proofs will become smaller due to a higher level of abstraction. Furthermore, dealing with reachability is easier compared to dealing with reachability on reference structures. The level of easiness also depends on provided proof automation for abstract data types, which lacks in certain aspects at the moment, e.g., concatenation of sequences and reasoning about sequences within sequences might be problematic at the time being.

For pure methods and ghost fields, opinions are somewhat divided to whether it contributes to making verification either easier or harder. For ghost fields the argument of having more heap changes states that it contributes to harder verifiable proofs. Others state, since ghost fields are very similar to symbolic code execution that verification becomes easier, whereas yet another participant provides this reason stating verifiability will not be affected compared to the general complexity of verification.

That pure methods make verification easier is only motivated by one of three answers, stating structure is introduced for the proof using this specification style. Participants that indicate pure methods complicate verification back their answers by stating method resolution and lookup is needed, and that either unfoloding of the method body or applying a corresponding contract is necessary. One participant answered *no difference* and motivates that it actually depends on the difficulty of the contract. Additionally a problem is that the quantifier instantiation in KeY does not work if the quantified formula contains a pure method.

Model fields make verification according to all but one expert harder. The motivation given for the answer that verification will be easier states that model fields should be handled the right way to help. Furthermore, they help abstracting complicated issues. Several motivations are given in case either the answer harder or much harder was given. Depends clauses as well as using a difficult representation for model fields involving the usage of a `\such_that` construct, e.g., in case of a recursive representation, seem to be the hardest parts of model fields. Two participants were not able to indicate what actually makes usage of model fields harder for verification.

On model methods the experts differ widely in opinion. One expert states model methods bring structure in the proof like pure methods, and thereby are also strictly pure, making model methods much easier for verification. The participants that answered proofs would be easier basically state that this is the case for certain usage of model methods, i.e.,

in case they are used with lemmata. These lemmata are basically the contracts model methods themselves can have. This is somewhat backed by the participant that answered *no difference* and gives the motivation that it depends on the usage as application of contracts come with an overhead which makes the proof situation harder to understand. Additionally, quantifier instantiation does also not work for model methods. Again, in case of usage with recursive structures difficulty largely increases.

Reflection

During the project several specification styles have been used, and different styles seemed better for specific usage, e.g., in most situations ghost fields suffice, but in case it is possible model fields were nicer, as they provide additional abstraction and less overhead maintaining as no set statements are needed. It was not always possible to use model fields however, e.g., the `repr` field as a model field made use of the `\such_that` construct necessary, which made proving not possible in the end.

During the project model methods were not yet at a usable stage. When sequences were used for the `ArrayList`, this indeed gave a bit harder to prove specifications when concatenation was used.

6. Conclusions

This thesis explored and specifies selected parts of the Java Collections Framework using JML*, and statically verifying these specifications using KeY. Different specification styles and constructs are used to see what the effects are on understandability, extensibility and verifiability. These effects are also evaluated by asking expert users (i.e., KeY developers) about their findings and experiences using JML* and KeY for verification in the past years. Furthermore, feedback about the understandability of the specifications made in this project was also gathered by means of a questionnaire – filled out by participants that only have a basic understanding of JML.

In this chapter, first it is discussed how the contributions achieve the goals set out in the introduction part of the thesis, and how results are evaluated. Next, overall limitations are indicated. Finally, possible directions for further research are discussed.

6.1. Goals and contributions

The introduction of this thesis described the goals to achieve with this thesis. In this section, it is described how the contributions of this thesis achieve these goals.

Specifying selected parts of the JCF to gain insight on understandability and extensibility

The first goal of this thesis is to gain insight on understandability and extensibility by specifying selected parts of the Java Collections Framework.

Chapter 2 describes the specifications made for several interfaces (i.e., `Collection`, `List`, `Iterator` and `ListIterator`) as well as specifications needed for the implementing classes `ArrayList` and the iterator used by the `ArrayList`. The specifications are based on the informal specifications of the Java API. Modifications have been performed to strip out generics, as at the time no tool could properly cope with generics. Furthermore, inner classes were extracted and made first class citizens as KeY was not able to reason about inner classes as support is broken at the moment. Here, it was found that ghost fields need additional specifications, i.e., invariants to specify restrictions as well as updating the state of the ghost fields, and therefore make understandability and extensibility worse. The opposite holds for the more abstract specifications of model fields. Pure methods seems to be the most understandable and extensible way of specifying. In case model methods would be usable, they would provide even more abstraction than model fields and probably further improve understandability and extensibility of specifications. Abstract data types provide a nice way to reason abstractly about implementations, however, they could be improved by, e.g., providing additional operations like `add` and `remove` for sequences.

Verifying selected parts of the JCF to gain insight into understandability

The second goal of the thesis is to gain insight into verifiability by verifying selected parts of the Java Collections Framework.

Chapter 3 first descibes how KeY was used to perform static modular verification on the specifications made, next, provides limitations for verification, and discusses difficulties encountered when doing verification. Among a few other limitations, it was not possible to use the full expressiveness of `\min` and `\max` constructs (see Section 3.1.3 on page 74) – as it was not possible to count different values depending on a complex inner expression – or provide meaningful specifications for the methods `equals` and `compareTo` (see Section 3.1.3). Furthermore, specifications seem to be harder to verify when `\forall` and `\exists` constructs are used. Additionally, it was found that using model fields might be particularly hard in case the `\such_that` construct was needed to specify what the model field should represent. The easiest construct for verification seems to be ghost fields, as they are very similar compared to regular fields in Java. Under certain conditions it was not possible to use pure methods for specification, after evaluation by expert users this seems to be caused by the fact that pure methods cannot yet be used

together with `\forall` or `\exists` constructs.

Evaluation

After specifying and verifying selected parts of the Java Collections Framework, the results found needed to be evaluated. Therefore, two questionnaires were made. First, a questionnaire to see if the specifications made are understandable for people only familiar with basic JML. And second, a questionnaire to see if the perceived understandability, extensibility and verifiability for different specifications constructs is also perceived by expert KeY users.

The questionnaire about the understandability of the specifications made, indicated that, primarly, JML itself was not always well understood, and not just the additional specifications used by JML*. For example, people have the perception that specifications are not verifiable when they do not cover the complete behavior of a method. Furthermore, the use of model fields as well as invariants is not completely clear. The use of abstract data types (i.e., `sequences`) as well as framing of methods – specifying the set of locations that might be changed by a method – did not cause a lot of additional confusion for the participants in general.

The second questionnaire, that asked about perceived understandability, extensibility and verifiability by expert users, indicated that the findings during specification and verification are very similar. The expert users, in general, also state that ghost fields are easier for verification, and that depending on the difficulty of pure methods these can be either easier or harder to verify. Additionally the motivation is provided that pure methods cannot be used within `\forall` or `\exists` constructs yet. Furthermore, model fields are indicated as potentially harder for verification, e.g., in case of representing recursive structures or a `\such_that` representation. Understandability and extensibility, is also by the expert users indicated better in case of a higher abstraction, i.e., model fields and abstract data types should provide better understandability and extensibility in general. A final note, depending on the participant, answers differ since one has more experience with one construct compared to another construct.

6.2. Limitations

Not limited to the use of KeY – probably verification tools in general – have a steep learning curve, i.e., it is hard to start using them and know what they can and cannot do. At this point, KeY and similar tools, are not ready for incorporating them for business (e.g., by software development companies) and their use is limited to academic research. Although the support of verification tool increases, only little verification has been performed on API's, not to mention frameworks that are used on top of API's. Not only is there a steep learning curve, but also only a small group of people is able to use verification tools at this point. This means there needs to be a large shift in software engineering, will software verification become part of the process of any substantial software product in the near future.

6.3. Future work

In this section several directions for further research towards gaining insight on understandability, extensibility and verifiability of JML* specifications are indicated.

Specification and verification: Only a limited amount of specifications and verifications could be examined, it would be interesting to cover bigger parts of the Java Collections Framework in the future and provide more complete specifications. Furthermore, it would be nice if there was no need to make inner classes first class citizens anymore.

Introduction of generics: For this thesis, several adjustments to the source were needed to cope with the fact it was not possible to handle generics, i.e., specifications cannot be type parametric. It would be nice to have this in the future, which would probably aid a lot in making specifications more understandable as well as extensible.

Concurrency: During the project concurrency is not taken into consideration, as JML* and JML do not support concurrent execution of Java (yet), however, this would be an interesting aspect to look at next. Especially in case the Java Collections Framework of Java 8 will be considered, as the framework will then be optimized for concurrent execution using lambda expressions, e.g., to filter and map collections [13]. However, this will probably introduce further problems regarding understandability and verifiability of specifications.

Understandability of specifications: It would be nice to gain more insight in what actually make specifications hard to understand. The understandability questionnaire indicated that invariants and model fields are not well understood. Maybe interviewing people, instead of letting them take a questionnaire, would aid in providing more precise results in what makes JML specifications difficult to understand.

Abstract Data Types: Abstract data types provided a nice way to abstract from implementations. Future work could consist of exploring additional constructs, i.e., besides location sets and sequences, that could aid making specifications more understandable. Also, it could be explored how the existing abstract data types could be changed to make them contribute more to a better understanding of specifications.

Model Methods: Model methods were not completely implemented, and therefore not yet used for specification and verification. It would be nice to explore whether model methods could improve on the understandability, extensibility and verifiability of specifications.

Incorporating Static Verification for Business: Another interesting project for future work would be to explore what would be needed to make it possible to eventually incorporate static verification into software development.

Bibliography

[1] Baier, C., Katoen, J.-P., et al. *Principles of model checking*. MIT press Cambridge, 2008.

[2] Barnett, M., Leino, K. R. M., and Schulte, W. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., 3362 ed. Springer Berlin Heidelberg, Berlin, 2005, pp. 49–69.

[3] Barrett, Clark W and Sebastiani, Roberto and Seshia, Sanjit A and Tinelli, C. Satisfiability Modulo Theories. *Handbook of satisfiability 185* (2009), 825–885.

[4] Beck, K., Gamma, E., and Staff, D. JUnit, 2008.

[5] Beckert, B., Hähnle, R., and Schmitt, P. H., Eds. *Verification of Object-Oriented Software. The KeY Approach*. Springer Berlin Heidelberg, 2007.

[6] Beust, C., and Suleiman, H. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[7] Blackburn, P., van Benthem, J. F., and Wolters, F. *Handboook of modal logic*. Access Online via Elsevier, 2006.

[8] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. 1–22.

[9] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. *Beyond assertions: Advanced specification and verification with JML and ESC/Java2*. 2006.

[10] Cheon, Y., and Leavens, G. T. *A runtime assertion checker for the Java Modeling Language (JML)*. March 2002.

[11] Cok, D. R. Specifying Java Iterators with JML and ESC/Java2. In *Proceedings of the 2006 conference on Specification and verification of component-based systems* (New York, New York, USA, 2006), ACM, pp. 71–74.

[12] Correa, A., Werner, C., and Barros, M. Refactoring to improve the understandability of specifications written in Object Constraint Language. *Software, IET 3*, 2 (2009), 69–90.

[13] Evans, B.; Verburg, M. G. R. Exploring Lambda Expressions for the Java Language and the JVM. *Java Magazine November/December 2012* (2012), 34–38.

[14] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. *ACM SIGPLAN Notices 37*, 5 (2002), 234–245.

[15] Hoare, C. a. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[16] Huisman, M. Verification of Java's AbstractCollection class: A case study. In *Mathematics of Program Construction* (2002), Springer, pp. 175–194.

[17] Hunt, A., Thomas, D., Hargett, M., and Programmers, P. *Pragmatic Unit Testing in C# with NUnit*, vol. 2. Pragmatic Bookshelf, 2004.

[18] Kassios, I. T. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Proceedings of the 14th international symposium of Formal Methods FM06* (2006), J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085 of *Lecture Notes in Computer Science*, Springer-Verlag Berlin, Heidelberg, pp. 268–283.

[19] Kiniry, J. R., and Cok, D. R. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system.

[20] Laitinen, K. Estimating understandability of software documents. *SIGSOFT Software Engineering Notes 21*, 4 (July 1996), 81–92.

[21] Leavens, G. T. JMLUnit: JML's Unit Testing Tool.

[22] Leavens, G. T. JML's rich, inherited specifications for behavioral subtypes. In *Formal Methods and Software Engineering*. Springer, 2006, pp. 2–34.

[23] Leavens, G. T., Baker, A. L., and Ruby, C. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes 31*, 3 (2006), 1–38.

[24] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Peter, M., Kiniry, J., Chalin, P., Zimmerman, D. M., and Dietl, W. *JML Reference Manual*, vol. 0400. 2011.

[25] Lehner, H. *A formal definition of JML in Coq and its application to runtime assertion checking*. PhD thesis, ETH Zurich, 2011.

[26] Lions, J. Ariane 5: Flight 501 Failure. Tech. rep., Independent Inquiry Board, Paris, 1996.

[27] Marché, C., Paulin-Mohring, C., and Urbain, X. The Krakatoa tool for certificationof Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming 58*, 1-2 (2004), 89–106.

[28] Meyer, B. *Object-Oriented Software Construction*, second ed. Prentice Hall, 1997.

[29] Meyer, J., and Poetzsch-Heffter, A. An architecture for interactive program provers. *Tools and Algorithms for the Construction and Analysis of Systems* (2000).

[30] Misra, S., and Akman, I. Weighted Class Complexity: A Measure of Complexity for Object Oriented System. *Journal of Information Science and Engineering 24* (2008), 1689–1708.

[31] Mostowski, W. Fully verified Java Card API reference implementation. *Verify 259* (2007), 136–151.

[32] Müller, P. *Modular specification and verification of object-oriented programs*. Springer, 2002.

[33] Peters, D. Specifying selected parts of the Java API using Dynamic Frames, 2010.

[34] Platzer, A. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010.

[35] Raghavan, A. D., and Leavens, G. T. Desugaring JML method specifications. *Department of Computer Science, Iowa State University TR# 00-03e* (2005).

[36] Reid, J., Freeman, S., Chris, R., Parker, A., and Denley, T. Java Hamcrest, 2012.

[37] Roth, A. Deduktiver Softwareentwurf am Beispiel des Java Collections Framework - Verfeinerungsbeziehungen in UML/OCL. Diploma thesis, University of Karlsruhe, Department of Computer Science, 2002.

[38] Rustan, K., and Leino, M. Data groups: specifying the modification of extended state. In *Proceedings, 13th ACM Conference on Object- Oriented Programming Systems, Languages and Applications (OOPSLA 1998)* (1998), ACM Press, pp. 144–153.

[39] Tillmann, N., and de Halleux, J. Pex-White Box Test Generation for .NET. In *Tests and Proofs*, B. Beckhert and R. Hähnle, Eds., 4966 ed. Springer Berlin Heidelberg, Berlin, 2008, pp. 134–153.

[40] Van Den Berg, J., and Jacobs, B. The LOOP Compiler for Java and JML. In *Proc Tools and Algorithms for the Construction and Analysis of Software TACAS Springer LNCS 2031 2001* (2001), vol. 2031 of *LNCS*, Springer, pp. 299–312.

[41] Warmer, J., and Kleppe, A. *The Object Constraint Language: Getting Your Models Ready for MDA*, 2 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[42] Weiss, Benjamin. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2010.

[43] Zimmerman, D. M., and Nagmoti, R. JMLUnit: the next generation. In *Formal Verification of Object-Oriented Software* (Paris, 2011), Springer-Verlag Berlin, Heidelberg, pp. 183–197.

Appendices

A. `repr` as `model` vs `ghost`

In Listing A.1 and Listing A.2, two minimalistic implementations are provided to understand the differences between abstract data representation with a `model` field versus a representation with a `ghost` field. Furthermore, a specification for the `copyof` method from `Arrays` – used by the implementations – is provided in Listing A.3.

```
1  public class GhostList {
2
3    /*@ public model instance \locset footprint;
4      @ public accessible \inv: footprint;
5      @ public accessible footprint: footprint;
6      @
7      @ public model instance int seqLength;
8      @ public instance invariant seqLength >= 0;
9      @ public accessible seqLength: footprint;
10     @
11     @ public ghost instance \seq repr;
12     @ public represents footprint = count, elems, elems[*], repr;
13     @ public represents seqLength = count;
14     @
15     @ public instance invariant (\forall int i; 0 <= i && i < repr.length;
16         repr[i] == elems[i]);
17     @ public instance invariant seqLength == repr.length;
18     @ public instance invariant seqLength <= elems.length;
19     @
20     @ public instance invariant elems != null;
21     @ public instance invariant \typeof(elems) ==
22         \type(java.lang.Object[]);
23     @*/
24
25   int count;
```

```
26    /*@ nullable @*/ Object[] elems;
27
28    /*@ public normal_behavior
29      @   ensures count == 0;
30      @   ensures repr == \seq_empty;
31      @   assignable footprint;
32      @*/
33    public GhostList() {
34      //@ set repr = \seq_empty;
35      count = 0;
36      elems = new Object[0];
37    }
38
39    /*@ public normal_behavior
40      @   ensures \result == seqLength;
41      @*/
42    /*@ strictly_pure @*/ public int size() {
43      return count;
44    }
45
46    /*@ public normal_behavior
47      @   ensures seqLength == \old(seqLength) + 1;
48      @   ensures (\forall int i; 0 <= i && i < \old(seqLength);
49            (Object)repr[i] == (Object)(\old(repr[i])));
50      @   ensures (Object)repr[\old(seqLength)] == o;
51      @   ensures \new_elems_fresh(footprint);
52      @   assignable footprint;
53      @*/
54    public void add(/*@ nullable @*/ Object o) {
55      //@ set repr = \seq_concat(repr, \seq_singleton(o));
56      elems = Arrays.copyOf(elems, count + 1);
57      elems[count++] = o;
58    }
59
60  }
```

Listing A.1: Data representation using a ghost field

```
1   public class ModelList {
2
3     /*@ public model instance \locset footprint;
4       @ public accessible \inv: footprint;
5       @ public accessible footprint: footprint;
```

```
6       @
7       @ public model instance int seqLength;
8       @ public instance invariant seqLength >= 0;
9       @ public accessible seqLength: footprint;
10      @
11      @ public model instance \seq repr;
12      @ public accessible repr: footprint;
13      @
14      @ public represents footprint = count, elems, elems[*];
15      @ public represents seqLength = count;
16      @
17      @ public instance invariant seqLength <= elems.length;
18      @ public instance invariant elems != null;
19      @
20      @ public represents repr \such_that
21          (\forall int i; 0 <= i && i < count;
22          (Object)repr[i] == (Object)elems[i]);
23      @
24      @ public instance invariant \typeof(elems) ==
25          \type(java.lang.Object[]);
26      @*/
27
28    int count;
29    /*@ nullable @*/ Object[] elems;
30
31    /*@ public normal_behavior
32      @   ensures count == 0;
33      @   assignable footprint;
34      @*/
35    public ModelList() {
36      count = 0;
37      elems = new Object[0];
38    }
39
40    /*@ public normal_behavior
41      @   ensures \result == seqLength;
42      @*/
43    /*@ strictly_pure @*/ public int size() {
44      return count;
45    }
46
47    /*@ public normal_behavior
```

```
48      @   requires \typeof(o) == \type(java.lang.Object);
49      @   ensures seqLength == \old(seqLength) + 1;
50      @   ensures (\forall int i; 0 <= i && i < \old(seqLength);
51            (Object)repr[i] == (Object)(\old(repr[i])));
52      @   ensures (Object)repr[\old(seqLength)] == o;
53      @   ensures \new_elems_fresh(footprint);
54      @   assignable footprint;
55      @*/
56    public void add(/*@ nullable @*/ Object o) {
57      elems = Arrays.copyOf(elems, count + 1); // OutOfMemoryError
58      elems[count++] = o;
59    }
60
61  }
```

Listing A.2: Data representation using a `model` field

```
1   class Arrays {
2
3     /*@ public normal_behavior
4       @   requires original != null;
5       @   requires newLength >= 0;
6       @   requires \typeof(original) == \type(java.lang.Object[]);
7       @   ensures \typeof(\result) == \type(java.lang.Object[]);
8       @   ensures newLength < original.length ==>
9             (\forall int i; 0 <= i && i < newLength;
10            \result[i] == original[i]);
11      @   ensures newLength >= original.length ==>
12            (\forall int i; 0 <= i && i < original.length;
13            \result[i] == original[i]);
14      @   ensures newLength > original.length ==>
15            (\forall int i; original.length <= i && i < newLength;
16            \result[i] == null);
17      @   ensures \result.length == newLength;
18      @   ensures \fresh(\result);
19      @   ensures \result != null;
20      @   assignable \nothing;
21      @ also
22      @ public exceptional_behavior
23      @   requires (newLength < 0) || (original == null);
24      @   signals (NegativeArraySizeException) newLength < 0;
25      @   signals (NullPointerException) original == null;
26      @   signals_only NegativeArraySizeException, NullPointerException;
```

```
27      @   assignable \nothing;
28      @*/
29    native public static /*@ nullable @*/ Object[]
30      copyOf(/*@ nullable @*/ Object[] original, int newLength);
31
32  }
```

Listing A.3: Specification of the method `copyof`

B. Questions for specification styles questionnaire

The questionnaire for specifications styles was divided into three pages, i.e., one for *understandability*, *extensibility* and *verifiability*. For each question the participant was asked to provide a motivation.

Understandability questions;

- 'Ghost fields' make understandability of specifications ..
- 'Model fields' make understandability of specifications ..
- 'Pure methods' make understandability of specifications ..
- 'Abstract data types' (e.g., sequences) make understandability of specifications ..
- 'Abstract predicates' (model methods) make understandability of specifications ..

Participants could answer the questions with the following multiple choice answers;

- *far worse*, *worse*, *no difference*, *better*, *far better* or *(not applicable)*.

Extensibility questions;

- 'Ghost fields' make extensibility of specifications ..
- 'Model fields' make extensibility of specifications ..
- 'Pure methods' make extensibility of specifications ..
- 'Abstract data types' (e.g., sequences) make extensibility of specifications ..
- 'Abstract predicates' (model methods) make extensibility of specifications ..

Participants could answer the questions with the following multiple choice answers;

- *far worse*, *worse*, *no difference*, *better*, *far better* or *(not applicable)*.

Verifiability questions;

- 'Ghost fields' make verification ..
- 'Model fields' make verification ..
- 'Pure methods' make verification ..
- 'Abstract data types' (e.g., sequences) make verification ..
- 'Abstract predicates' (model methods) make verification ..

Participants could answer the questions with the following multiple choice answers;

- *much harder*, *harder*, *no difference*, *easier*, *much easier* or *(not applicable)*.

C. Accompanying file for questionnaire

The understandability questionnaire used an accompanying file, provided in Listing C.1.

```
1  class SelfArrays {
2
3    /*@ public normal_behavior
4      @   requires original != null;
5      @   requires newLength >= 0;
6      @   requires \typeof(original) == \type(java.lang.Object[]);
7      @   ensures \typeof(\result) == \type(java.lang.Object[]);
8      @   ensures newLength < original.length ==>
9            (\forall int i; 0 <= i && i < newLength;
10           \result[i] == original[i]);
11     @   ensures newLength >= original.length ==>
12           (\forall int i; 0 <= i && i < original.length;
13           \result[i] == original[i]);
14     @   ensures newLength > original.length ==>
15           (\forall int i; original.length <= i && i < newLength;
16           \result[i] == null);
17     @   ensures \result.length == newLength;
18     @   ensures \fresh(\result);
19     @   ensures \result != null;
20     @   assignable \nothing;
21     @ also
22     @ public exceptional_behavior
23     @   requires (newLength < 0) || (original == null);
24     @   signals (NegativeArraySizeException) newLength < 0;
25     @   signals (NullPointerException) original == null;
26     @   signals_only NegativeArraySizeException, NullPointerException;
27     @   assignable \nothing;
28     @*/
```

```
29    native public static /*@ nullable @*/ Object[]
30      copyOf(/*@ nullable @*/ Object[] original, int newLength);
31  }
32
33  public class ArrayList {
34
35    /*@ public model instance \locset footprint;
36      @ public accessible \inv: footprint;
37      @ public accessible footprint: footprint;
38      @
39      @ public nullable ghost instance \seq repr;
40      @ public model instance int seqLength;
41      @ public accessible seqLength: footprint;
42      @
43      @ represents footprint = elementData, elementData[*], size, modCount,
            repr;
44      @
45      @ public instance invariant (\forall int i; 0 <= i && i < repr.length;
46            repr[i] == elementData[i]);
47      @ public represents seqLength = size;
48      @
49      @ public instance invariant size == repr.length;
50      @ public instance invariant seqLength <= elementData.length;
51      @ public instance invariant \typeof(elementData) ==
52          \type(java.lang.Object[]);
53      @ public instance invariant modCount >= 0;
54      @ public instance invariant size >= 0;
55      @*/
56
57    private Object[] /*@ spec_public nullable @*/ elementData;
58    //@ public instance invariant elementData != null;
59
60    protected int /*@ spec_public @*/ modCount = 0;
61    /*@ spec_public @*/ protected int size;
62
63    //////////////
64    /* Method 1 */
65    //////////////
66
67    /*@ public normal_behavior
68      @   requires initialCapacity >= 0;
69      @   ensures elementData.length == initialCapacity;
```

```
70      @   ensures (\forall int i; 0 <= i && i < seqLength;
71            elementData[i] == null);
72      @   ensures repr == \seq_empty;
73      @   ensures \fresh(footprint);
74      @   assignable footprint;
75      @*/
76    public ArrayList(int initialCapacity) {
77      if (initialCapacity < 0)
78        throw new IllegalArgumentException();
79
80      this.elementData = new Object[initialCapacity];
81      //@ set repr = \seq_empty;
82      {}
83    }
84
85    //////////////
86    /* Method 2 */
87    //////////////
88
89    /*@ public normal_behavior
90      @   requires index >= 0 && index < seqLength;
91      @   ensures \result == repr[index];
92      @   assignable \strictly_nothing;
93      @ also
94      @ public exceptional_behavior
95      @   requires index < 0 || index >= seqLength;
96      @   signals (IndexOutOfBoundsException) true;
97      @   assignable \nothing;
98      @*/
99    /*@ nullable @*/ public Object get(int index) {
100     if (index >= size || index < 0)
101       throw new IndexOutOfBoundsException();
102
103     return elementData[index];
104   }
105
106   //////////////
107   /* Method 3 */
108   //////////////
109
110   /*@ public normal_behavior
111     @   requires size < elementData.length;
```

```
112     @   ensures modCount == \old(modCount) + 1;
113     @   ensures elementData.length == size;
114     @   ensures repr == \old(repr);
115     @   assignable elementData, modCount;
116     @ also
117     @ public normal_behavior
118     @   requires size >= elementData.length;
119     @   ensures modCount == \old(modCount) + 1;
120     @   ensures repr == \old(repr);
121     @   assignable modCount;
122     @*/
123   public void trimToSize() {
124     modCount++;
125     int oldCapacity = elementData.length;
126     if (size < oldCapacity) {
127       elementData = SelfArrays.copyOf(elementData, size);
128     }
129   }
130
131   //////////////
132   /* Method 4 */
133   //////////////
134
135   /*@ public normal_behavior
136     @   ensures modCount == \old(modCount) + 1;
137     @   ensures size == 0;
138     @   ensures repr == \seq_empty;
139     @   ensures (\forall int i; 0 <= i && i < elementData.length;
140           elementData[i] == null);
141     @   assignable elementData[*], repr, size, modCount;
142     @*/
143   public void clear() {
144     modCount++;
145
146     int i = 0;
147     /*@ loop_invariant 0 <= i && i <= size;
148       @ loop_invariant (\forall int j; 0 < j && j < i;
149           elementData[j] == null);
150       @ assignable elementData[*];
151       @ decreasing size - i;
152       @*/
153     while(i < size) {
```

```
154       elementData[i] = null;
155       i++;
156     }
157
158     //@ set repr = \seq_empty;
159     size = 0;
160   }
161
162   //////////////
163   /* Method 5 */
164   //////////////
165
166   /*@ public normal_behavior
167     @   requires \typeof(element) == \type(Object);
168     @   requires index >= 0 && index < seqLength;
169     @   ensures repr[index] == element;
170     @   ensures \result == \old(repr[index]);
171     @   ensures \new_elems_fresh(footprint);
172     @   assignable footprint;
173     @ also
174     @ public exceptional_behavior
175     @   requires index < 0 || index >= seqLength;
176     @   signals (IndexOutOfBoundsException)
177           index < 0 || index >= seqLength;
178     @   signals_only IndexOutOfBoundsException;
179     @   assignable \nothing;
180     @*/
181   public /*@ nullable @*/ Object set(int index,
182       /*@ nullable @*/ Object element) {
183     if (index >= size || index < 0)
184       throw new IndexOutOfBoundsException();
185
186     Object oldValue = elementData[index];
187     elementData[index] = element;
188
189     /*@ set repr = \seq_concat(
190           \seq_concat(\seq_sub(repr, 0, index), \seq_singleton(element)),
191           \seq_sub(repr, index + 1, seqLength)
192         );
193       @*/
194
195     return oldValue;
```

```
196   }
197
198   //////////////
199   /* Method 6 */
200   //////////////
201
202   /*@ public normal_behavior
203     @   requires \typeof(e) == \type(java.lang.Object) && e.\inv;
204     @   ensures seqLength == \old(seqLength) + 1;
205     @   ensures (\forall int i; 0 <= i && i < seqLength-1;
206           repr[i] == \old(repr[i]));
207     @   ensures repr[seqLength-1] == e;
208     @   ensures \result;
209     @   assignable footprint;
210     @*/
211   boolean add( /*@ nullable @*/ Object e) {
212     elementData = SelfArrays.copyOf(elementData, size + 1);
213     elementData[size++] = e;
214     //@ set repr = \seq_concat(repr, \seq_singleton(e));
215     {}
216     return true;
217   }
218
219   //////////////
220   /* Method 7 */
221   //////////////
222
223   /*@ public normal_behavior
224     @   requires \typeof(o) == \type(Object) && o.\inv;
225     @   ensures (\forall int k; 0 <= k && k <= seqLength-1; repr[k] != o)
226           ==> \result == -1;
227     @   ensures !(\exists int k; 0 <= k && k <= seqLength-1; repr[k] == o)
228           ==> (\result >= 0 && \result < seqLength && repr[\result] == o);
229     @   ensures !(\exists int k; 0 <= k && k < \result; repr[k] == o);
230     @   assignable \nothing;
231     @*/
232   public int indexOf( /*@ nullable @*/ Object o) {
233
234     if (o == null) {
235
236       /*@ loop_invariant 0 <= i && i <= size;
237         @ loop_invariant (\forall int j; 0 <= j && j < i; repr[j] != o);
```

```
238         @ assignable \strictly_nothing;
239         @ decreasing size - i;
240         @*/
241       for(int i = 0; i < size; i++)
242         if(elementData[i] == null)
243           return i;
244     } else {
245       /*@ loop_invariant 0 <= i && i <= size;
246         @ loop_invariant (\forall int j; 0 <= j && j < i; repr[j] != o);
247         @ assignable \nothing;
248         @ decreasing size - i;
249         @*/
250       for (int i = 0; i < size; i++)
251         if(o == elementData[i])
252          return i;
253     }
254     return -1;
255   }
256
257 }
```

Listing C.1: Accompanying file for the understandability questionnaire