Verifying EVA

Formal verification of the software for deciding Norwegian governmental elections

Henrik Torland Klev

Thesis submitted for the degree of
Master in "Informatics: Programming and System Architecture"
("Software")
60 credits

"Department of Informatics"
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2020

Verifying EVA

Formal verification of the software for deciding Norwegian governmental elections

Henrik Torland Klev

Abstract

EVA is the main support system used by municipalities and counties in Norway to hold and implement elections. As such, it is critical to society that the system behaves as expected. By using formal methods it can be proven that the system behaves as expected, and that it cannot fail to behave as expected. This work will present EVA and its central aspects, specify the critical properties using the specification language *Java Modeling Language* [50], and prove those properties using a formal technique known as *automated theorem proving* [5] with the assistance of the KeY System [17]. In addition, the Java Modeling Language and the KeY System will be sufficiently presented such that the most prominent features and theories can be implemented into other verification efforts, whereas the limitations of the KeY System will be presented and discussed. The sequential, non-distributed nature of the implementation of EVA makes this system a prime target for using deductive verification in an environment were library methods and external frameworks are extensively used. En masse, this work may serve as a guide for others who seek to start a verification project of their own, or to continue and extend the presented work.

Sammendrag

EVA er hovedstøttesystemet for valggjennomføring for kommunene og fylkeskommunene i Norge. Som sådan er det kritisk for samfunnet at systemet oppfører seg som forventet. Ved å bruke formelle metoder kan det påvises at systemet oppfører seg som forventet, og også at det ikke kan unnlate å oppføre seg som forventet. Dette arbeidet vil presentere EVA og dets sentrale aspekter, spesifisere de kritiske egenskapene ved bruk av spesifikasjonsspråket *Java Modeling Language* [50], og bevise disse egenskapene ved å bruke en formell teknikk kjent som *automated theorem proving* [5] med assistanse fra KeY Systemet [17]. I tillegg vil Java Modelling Language og KeY-systemet bli tilstrekkelig presentert slik at de mest fremtredende funksjonene og teoriene kan implementeres i andre verifiseringsinnsatser, og begrensningene til KeY-systemet vil bli presentert og diskutert. Den sekvensielle, ikke-distribuerte karakteren av implementeringen av EVA gjør systemet til et utmerket mål for å vise deduktiv verifisering i et miljø der bibliotekmetoder og eksterne rammeverk blir mye brukt. I det store og hele kan dette arbeidet fungere som en veiledning for andre som er ute etter å starte sitt eget verifiseringsprosjekt, eller ute etter å fortsette og utvide det presenterte arbeidet.

The space above and below this message is intentionally left blank.

Acknowledgements

This thesis is the result of my five year long spell at the Department of Informatics at University of Oslo, and would not be possible without all the fantastic people that have helped, guided, taught and challenged me along the way. I would like to give special thanks to the great people of the Reliable Systems Group, especially my supervisors Christian Johansen and Martin Steffen — without you this work would have been impossible to create.

Furthermore, both my girlfriend and roommate deserves explicit praise. My girlfriend for being my greatest supporter and always believing in me, and my roommate for keeping me motivated and inspired through difficult times. You both mean more to me than you think.

I would also like to thank the people that have made my life as a student one of the best experienced I could have ever hoped for. In no specific order, these include OSI Dragons, FIFI, SV-IT, RF, SEF, Eliten2.0, the OSRS Community (especially Adam, Curtis, Kacy and Jase), all my teachers, TAs, colleagues and fellow students, and of course, all of my friends and entire family.

Finally, to my grandma and grandpa: you were the greatest two people I ever knew. This work is dedicated to you.

The space above and below this message is intentionally left blank.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Formal Methods for Software Verification

Formal verification is the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of correctness [6]. It does not suffice to show that a system meets its requirements; one has to *prove* that the system cannot fail to meet its requirements. In a software setting, formal verification has the ability to prove that certain properties hold for a program regardless of supplied input. Further, in the case where a property does not hold, most formal verification approaches provide rigorous counterexamples that can guide system designers towards the flaw in the system.

Currently in industry, measures to increase software quality are in use to different degrees, where prominent methods for establishing that the software fulfills its functional requirements include runtime testing, static code reviews, and, to some degree, formal methods. Testing is the most widely accepted form of quality assurance [37], and is a part of all software development projects to some degree [66]. However, as Dijkstra famously wrote [18]:

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

This is especially true with bugs caused by numerical calculations; bugs of this type have a tendency to be *silent*, meaning they often occur without the program throwing exceptions or terminating abruptly. For instance, in some programming languages such as C, the function in Listing 1.1 will return true for a specific input due to the obscure semantics of integer overflow. There are ways to reduce the possibility of errors caused by numerical calculations, such as utilizing compiler-specific options [25], adapting a more *defensive* programming approach [68], or, the approach taken in this work, proving formally that they cannot occur.

In domains such as aviation, medicine, or telecommunications, where software failure is linked with catastrophic costs, formal methods have become an indispensable part of the development process. However, in the domain of general software development, formal methods have not received a similar warm welcome. Still, with the advent of verification tools, the increased abundance of computational resources, the growth in application areas, the importance of software quality for marketing [45], and the general satisfaction reported by developers working with formal methods [24], the attitude towards formal methods are turning for the better [42].

```
// Should always return FALSE.
// Returns TRUE for x = 2 147 483 648.
int overflowSemantics(int x){

    int MAX_INT = 2147482647;
    int MIN_INT = -2147483648;
    int TRUE = 1;
    int FALSE = 0;

    if (x-1 > x && x == -x
                && x != 0
                && x * 2 == 0
                && x <= MAX_INT
                && x >= MIN_INT){
        return TRUE;
    } else {
        return FALSE;
    }
}
```

Listing 1.1: Function that should always return `FALSE`, but returns `TRUE` for specific inputs due to integer overflow semantics.

This thesis will demonstrate the capabilities of formal methods by deductively verifying parts of EVA[1] [53]. EVA is the main support system for electoral implementation for municipalities and counties in Norway. All the basic data needed to conduct elections in Norway are recorded in this system, including the parties' list proposals and the electoral figure. It is also in EVA that the municipalities and constituencies register voting on election day and report their election results.

EVA is continuously operated and developed by the Electoral Directorate. The version of the EVA that is presented in this thesis was started already before the Storting and Sami Parliament elections in 2017, and the Electoral Directorate is planned to start work on the newest version to be used for the elections in 2021 [19].

1.2 Research Goals

This work will concern itself with parts of the EVA-system [53] — the official software system used for administrating municipal-, county-, and parliamentary elections in Norway. Specifically, the primary verification target is the systems implementation of the central algorithm known as Sainte-Laguë's modified method [69].

The work presented in this thesis seeks to achieve three principal goals:

[1]EVA is an acronym for *Elektronisk Valg Administrasjon*, which translates to *Electronic Election Administration*

1. The primary goal is to increase the confidence in EVA by proving certain properties about its code. It is of great importance that the calculated result of the election is equivalent to the expected, correct result, as an error of a single representative may have severe consequences [74]. EVA is continuously operated and the newer version builds upon the previous version, and as such, any undetected errors present in previous versions may propagate into later versions. The longer an error goes undetected, the more resources will be required to resolve it. Also, the error's impact on the system may increase drastically as the system evolves. Furthermore, as electronic elections are becoming the new standard, it is vital that the electors trust the new processes and the software that implements them. By mathematically proving that the system cannot fail to meet its expectations, at least one possible source of scepticism can be eliminated.

2. Secondly, this thesis will attempt to be as transparent as possible when reaching the primary goal. The goal is that the steps taken, and more importantly, the reasoning behind them, will be understood throughout the verification effort. If accomplished, this work may assist in implementing formal verification into other similar projects. Furthermore, fundamental and relevant tools, techniques and paradigms will be presented and discussed. When put together, this work may help in reducing the initial effort required to start a formal verification effort. In turn, this might alleviate the concerns of complexity that one might encounter when starting out, and, as a consequence, less people are deterred from the field.

3. Finally, there is a lack of examples of KeY [17] being deployed for the verification of real-world systems; most of the cases presented in [17] are based on trivial programs, systems that were designed for verification, or for academic competitions such as VerifyThis [33]. This work will demonstrate how one can formalize and verify certain properties of a complicated system that is being actively used and developed, in order to eliminate any unwarranted mistrust one might have in the system's behavior. Furthermore, several of the verification artifacts presented throughout this thesis may be reused and implemented for other projects.

1.3 Research Methodology

According to the 2002 article *Scientific Methods in Computer Science* [21] the field of Computer Science is divided into three distinct categories, each with their own preferred methodologies. The three categories are can be summarized as follows:

- **Theoretical computer science** seeks largely to understand the limits on computation and the power of computational paradigms, and attempts to develop general approaches to problem solving [21].

- **Experimental computer science** is concerned with extracting results from real world implementations, in order to test the veracity of theories or obtaining new knowledge. Methodologies in this category are used for, among others, automated theorem proving [26].

- **Computer simulations** make it possible to investigate regimes that are beyond current experimental capabilities and to study phenomena that cannot be replicated in laboratories, such as the evolution of the universe [21].

In this thesis, the methodologies from experimental computer science are applied. The real-world implementation of EVA is specified and transformed into proof obligations in order to prove that the implementation conforms to the specification. The methodology of automated theorem proving with assistance of the proof assistant of the KeY System is used to complete the proofs. The verification effort is presented in Chapter 4, while the results are summarized in Chapter 5.

1.4 Main Contribution

Section 1.2 presented the three main goals of this thesis. The first is to increase confidence in that EVA behaves as expected according to Electoral Law [52], the second is to present the theoretical foundation necessary to verify other Java-based systems, and the third goal is to provide an example of a verification effort from start to finish and thereby displaying the capabilities of the KeY System.

1. The first goal is reached through proving properties of EVA Resultat in Chapter 4.
2. The second goal is reached in Chapters 2 and 3 by presenting the specification language JML, the logical language JDL, and the theory behind the KeY System.
3. The third goal is reached through the first goal and Section 4.5.5, as Chapter 4 discusses the verification process and problems encountered.

1.5 Related Work

- The work in [46] explains the process of formally verifying the seL4 microkernel utilizing interactive theorem proving and the Isabelle/HOL proof assistant [60]. The property proved was *refinement*: a refinement proof establishes a correspondence between a high-level (abstract) and a low-level (concrete, or refined) representation of a system. To show this property, they implemented both an abstract and executable specification in Isabelle/HOL code, and proved that the C code implementing the kernel conformed to those specifications. The verification effort in Chapter 4 will adopt a similar approach where the verification target is implemented in Java, not C, and the proof assistant is part of the KeY Project [17] as opposed to Isabelle/HOL.

- A case study displaying functional verification and information flow analysis of an electronic voting system is found in [34]. The verification target in this study is a simplified Java-implementation of the protocols discussed in [48]. The authors of [34] have themselves implemented the verification target, and their implementation utilizes no external libraries. The goal of the verification process was to prove that the system preserved privacy of votes, i.e., individual votes were not attributable to a particular voter. They were able to verify that their implementation upholds the specified functional and dependency properties, and thereby conclude that the goal was reached. Their work does well in illustrating how one can create a formal specification for verification of a system created by one-selves. However, it is often of interest to verify artifacts that one has not created one-selves, such as verifying properties of external libraries. In this respect, this thesis attempts to illustrate how one can apply the verification process to systems where one has no control over the development process.

1.6 Outline

The remainder of this thesis is structured in the following manner:

- Chapter 2 contains introductory sections on the fundamental terms, concepts, definitions, and notions used throughout this work. This includes a presentation of deductive verification, proof obligations, and the design-by-contract paradigm. Further, the specification language JML, and the corresponding dynamic logic JavaDL, will be introduced due to them being vital aspects of the deductive verification methodology employed in later chapters.

- Chapter 3 will present the KeY System — a tool used for translating JML annotations into JavaDL contracts, extracting proof obligations from contracts, and then assisting in discharging the obligations. Here the focus will be primarily on the formalism and processes that KeY employs, along with the proof assistant inherent in the system.

- Chapter 4 demonstrates the capabilities of the KeY System by formally verifying parts of the electoral software employed for Norwegian elections, EVA [53]. The software will be presented along with an introduction to the Norwegian electoral process. Further, the system's behavior will be formally specified, and the system's implementation will be proved to conform to the specification.

- Chapter 5 concludes the thesis by discussing the results of verification effort, and by giving an outlook on further ideas and future work related to the topic.

2 Fundamentals

In the following, a presentation of some fundamental concepts, upon which this work is based, will be given. Starting of, in Section 2.1, the reader will become familiar with the concepts of formal software verification, with emphasis on deductive verification. In addition, the concepts of proof obligations and a deductive system, Hoare logic, will be introduced. The section will conclude with a discussion on software correctness and the necessity of specification languages. In Section 2.3 and 2.4 the Java Modeling Language (JML) and Java Dynamic Logic (JavaDL) will be presented respectively. JavaDL is a dynamic logic language [38] suitable for writing proof obligations for Java code, while JML is a specification language intended to be used for annotating programs written in Java. JML annotations can be mechanically translated into JavaDL proof obligations, which can later be discharged automatically or interactively. Specifications written in JML are largely based on the *design-by-contract* paradigm, which will be introduced in Section 2.2.

2.1 Deductive Verification

2.1.1 Proof Obligations, Properties and Correctness

Deductive verification [3, 23] is a formal verification approach based on extracting mathematical statements from a component so that proving the statements also proves properties of the component. Such mathematical statements will, from now on, be refereed to as *proof obligations*:

Definition 2.1 - Proof Obligations

A proof obligation for a component `comp` is a mathematical formula to be proven/discharged, in order to ensure that `comp` upholds specified properties, and is thereby correct with regard to a given specification.

The term "component" may be used to refer to software, hardware, specifications, protocols, or other artifacts, that may itself run several concurrent processes while deployed in either a local or distributed environment. In addition, there are multiple types and categories for properties the may be beneficial to verify for a given component depending on the components type, function, and environment. Further, there are two main notions of correctness, *partial* and *total* correctness, that will be discussed shortly.

KeY and JML are suitable for proving two types of properties of components, where the components are methods and classes of sequential Java programs running in a local Java virtual machine (JVM) [57]. These two property-types are *functional* properties and *dependency* properties. Functional properties for methods verify that the result of invoking a method adheres to a given specification, while dependency properties verify that the result of a method depends at most on a given set of locations on the JVM heap. Properties for classes are very similar, but may in addition verify certain aspects of objects created from a class.

The process of mathematically proving properties of programs is far from new; one of the very first proofs was presented by Alan Turing in 1949 [11], and in 1969, Hoare suggested in [40] a deductive system for rigorously proving program properties. Hoare's system is interesting for historical reasons, but it is also presented to illustrate the core ideas behind deductive verification. As such, the following section is devoted to his system.

2.1.2 Hoare Logic

Hoare logic is a formal system that consists of a set of axioms and inference rules for reasoning rigorously about the correctness of computer programs. The system itself is based on *Hoare triples*, which are defined informally as:

Definition 2.2 - Hoare Triples for Partial Correctness

A Hoare triple for partial correctness is a triple $\{p\}\, S\, \{q\}$, where S is a program segment, and p and q are predicates called the *precondition* and *postcondition*, respectively. S is *partially* correct with respect to p and q, written $\models_{\texttt{PAR}} \{p\}\, S\, \{q\}$, iff:

> If S is started in a state where p is true and *if* the computation of S terminates, then it terminates in a state where q is true.

Hoare's original approach could prove partial correctness, i.e., the result is correct only when the program terminates. Programs may also not terminate, as we know from Turing [70], either intentionally, as in the case of a web server or a reactive system[2], or unintentionally, e.g., by a mistake made by the programmer which leads the execution in an infinite loop.

Hoare's approach was extended by Manna and Pnueli in 1974 [54] to support proving both correctness and termination by one unified formalism. This was done, in part, by extending Definition 2.2 into the following:

Definition 2.3 - Hoare Triples for Total Correctness

A Hoare triple for total correctness is a triple $\{p(x) \mid S \mid q(x, \hat{x})\}$, where S is a program segment, and p and q are predicates called the *precondition* and *postcondition*, respectively. S is *totally* correct with respect to p and q, written $\models_{\texttt{TOT}} \{p\}\, S\, \{q\}$, iff:

> If, for every x, S is started in a state where $p(x)$ is true, then the execution of S terminates and $q(x, \hat{x})$ holds between the initial values x and the resulting values $\hat{x}$.

For a system to be considered totally correct, termination has to be guaranteed; a program that runs indefinitely is by definition partially correct irregardless of its computation. Unsurprisingly, total correctness is preferred to, and often more difficult to prove than, partial correctness. When unconcerned with the type of correctness, or the type is clear from context, the notation $\models \{p\}\, S\, \{q\}$ is used without specifying the correctness type.

Hoare's system consists of several axioms describing the transforming effect that simple statements have on program variables, and a set of inference rules for combining theorems for smaller segments into one theorem for a larger segment. The system has rules for assignment, concatenation, conditionals and `while`-loops. For instance, the first rule, the axiom of assignment, reads

$$\models P_0\, \{x := f\}\, P \tag{1}$$

where, x is an identifier for a simple program variable, f is a side-effect free expression, possibly containing x, and P_0 is obtained from P by substituting all occurrences of f by x, i.e., $P_0 = P\,[f \leftarrow x]$.

The rule states that, in order for the assertion P on the right-hand side of the assignment to be true for the value of x *after* the assignment, it also has to be true for the value of f *before* the assignment. Thus, if $P(x)$ is to be true after the assignment, the $P(f)$ has to be true before the assignment.

Theoretically, there are no restrictions on the size of programs that Hoare's system is applicable to. However, as Hoare's proposed process involves proving validity of several conditions, the process is considered tedious, time-consuming and prone to error. As such, it is limited by the mathematical skill of the user. Fortunately, half a century has passed, and there are now a plethora of tools, such as automatic theorem provers (ATPs) or satisfiability-modulo-theories (SMT) solvers, providing assistance with some portion of the verification process. Examples of tools include Sledgehammer [55] for the Isabelle proof assistant [60], the Z3 [15] SMT-solver, and the KeY System [17] for verifying Java programs. KeY works with proof obligations written in a dynamic logic, JavaDL, which is based on a first-order dynamic logic closely related to Hoare logic; in JavaDL, the Hoare triple $\{p\}\, S\, \{q\}$ can be expressed as $p \rightarrow [S]\, q$, where $\rightarrow$ is logical implication, and $[.]$ is a modal operator (see Section 2.4 for details). In contrast to Hoare logic, where formulas are purely first-order, formulas in JavaDL are more expressive, as they can contain programs. JavaDL is presented further in Section 2.4, and the KeY System in Section 3.

2.1.3 The Importance of Specifications

Tools are only assistants for reaching a specific goal; in this case, the goal is proving program correctness. A program is defined to be correct only if it meets the requirements of its specification. Therefore, a program can only be correct in accordance to a given specification. As such, it is of equal importance that the specification is correct. To illustrate, consider the follow specification for a sorting algorithm:

Specification 2.1 - Sorting Algorithm

The sorting algorithm should comply with the following:

1. The length of the input and the output should be equal.
2. Every element in the output should be sorted, i.e., `output` $[i] \leq$ `output` $[i+1]$.
3. The sorting algorithm must assume that elements of the input can be equal.
4. The algorithm should provide stable sorting, i.e., if `input` $[i] =$ `input` $[j]$, $i < j$, `input` $[i]$ is sorted into `output` $[k]$ and `input` $[j]$ is sorted into `output` $[r]$, then $k < r$.

One attempt at implementing the specification is seen in Listing 2.1 below. The implementation is *"correct"* as it does fulfill all the requirements of its given specification. Or does it? Requirement 4 might seem formal, but it is ambiguous as it can be argued either way if the requirement is fulfilled by the method in Listing 2.1. One can argue that the requirement has not been fulfilled as the input contains two instances of the element `2`, while the output contains a single instance of the element. Contrary, the requirement can be considered fulfilled due to the *if*-condition, "`input` $[i]$ is sorted into `output` $[k]$ and `input` $[j]$ is sorted into `output` $[r]$", being falsified by `input` $[j]$ *not* being sorted into `output` $[r]$. Finally, one has to take user expectations into consideration when creating and/or fulfilling a specification; it is common to expect that a sorting algorithm fulfills the *retention criteria* such that the output is a permutation of the input, i.e., all elements of `input` must be present in `output`.

This example displays a key challenge in the process of formal verification: if an error exists, is the error rooted in the code or the specification? As the software industry evolves, it has become increasingly clear that specifications play an important part in the verification process. Events such as the disasters caused by the Boeing software scandal [63] further displays that even if a specification is correct according to some stakeholders, e.g., system designers, it might not be correct according to other stakeholders, e.g., end-users.

In conclusion, in order to formally verify a system according to a specification, even when proving simple properties, the specification needs to correctly and unambiguously capture the system's behavior as intended and as expected by the stakeholders of the system. To achieve such goals, expressive formal specification languages are a necessity. Roughly speaking, a specification language consists of a mathematical language in which proof obligations and components of the specification is written, and its integration with a programming language. The choice of specification language is tightly coupled with both the methodology used to generate the proof obligations and the techniques used to prove them. One such language central to this thesis is the Java Modeling Language (JML), introduced in Section 2.3.

```java
public class Sort {
    public static void main(String args[]) {
        int[] input = {5, 2, 98, 2, 48, 99, 23, 184};
        int[] output = sort(input);

        // Prints 0 1 2 3 4 5 6 7
        for (int i : output){
            System.out.print(i + " ");
        }
    }

    // Input:              5   2  98   2  48  99  23 184
    // Expected output:    2   2   5  23  48  98  99 184
    // Actual output:      0   1   2   3   4   5   6   7
    // Correct according to specification, not necessarily
    // correct according to user expectations.
    public static int[] sort(int[] input){
        int length = input.length;
        int[] output = new int[length];
        for (int i = 0; i < length; i++){
            output[i] = i;
        }
        return output;
    }
}
```

Listing 2.1: Sorting algorithm without retention criteria.

2.2 Design by Contract

The term ”design by contract” was coined by Bertrand Meyer in [56], where he describes an approach to software design based on the central idea that modules of a software system collaborate with each others on the basis of mutual *obligations* and *benefits*. At the core lies the notion of method contracts:

Definition 2.4 - Method Contracts

Contracts of methods are an agreement between the *caller* and the *callee*, describing what guarantees they provide to each other. They describe what is expected from the code that calls the method, and it provides guarantees about what the method will actually do. The expectation on the caller are called the *preconditions*, while guarantees provided by the callee are called the *postconditions*.

Any contract for a module can require that certain conditions be guaranteed by the client modules that calls it, guarantee certain properties on module termination, and guarantee that certain properties are maintained. In other words, the contract defines the preconditions, postconditions and invariants[2] of their respective modules. Preconditions and invariants are obviously a benefit to the callee and a obligation to the caller, and the converse is true for postconditions. Further, the obligations of contracts can be formalised in e.g. Hoare triples as they are semantically equivalent.

The principle of design by contract promotes modularization and abstraction, which are both key techniques for scaling up large software systems. In the case of formal verification, modularization and abstraction is achieved by method contracts; once a method contract has been separately verified there is no need to inspect the methods code again, but instead rely on the postconditions ensured by the contract. In addition, method contracts enable system developers to program more offensively; that is, they can assume that the supplied input to the method in question is always well-defined and in the required range. When validating the application, one checks that every call to the method conforms to the methods contract, and thus explicit checks inside the methods body are not necessary.

2.3 Java Modeling Language (JML)

The Java Modeling Language, JML, is an increasingly popular specification language for Java source code, that has been developed by the community since 1999 [17]. The language is still changing and growing, both in features and users, and as such there is no accepted ultimate reference. Fortunately, for the elements addressed during this thesis, there exist a gold standard created by Gary T. Leavens and his colleagues [50]. This references manual explains how the language is designed to have the required expressivity to document the behavior of special aspects of the Java programming language, e.g., object creation, abstraction, inheritance or throwable exceptions, while being readily understandable to Java programmers and amenable to tool support. The rest of Section 2.3 will present the principal elements of the JML syntax and semantics.

An example JML method contract is displayed in Listing 2.2. The elements of the contract will be explained throughout sections 2.3.1-2.3.6.

```
class StringSet {

  /*@ spec_public @*/ private String[] strArr;
  /*@ spec_public @*/ private int limit;

  //@ public ghost \bigint size;
  //@ instance invariant size <= limit;

  //@ public instance model String first;
  //@ represents first = strArr[0];

```

[2]An invariant is a property that does not change after certain transformations. In other words, an invariant is a formula (or expression, predicate) that should always be true for a given module (e.g., class, method, program, loop).

```
  public StringSet(int limit){
    this.limit = limit;
    strArr = new String[limit];
    //@ set size = 0;
  }

  /*@ public normal_behavior
    @ requires size < limit && !contains(elem);
    @ ensures \result == true;
    @ ensures contains(elem);
    @ ensures (\forall int i;
    @                   0 <= i && i < size &&
    @                   strArr[i] != elem;
    @                   strArr[i] == \old(strArr[i]));
    @ ensures size == \old(size) + 1;
    @ assignable strArr[*];
    @
    @ also
    @
    @ public exceptional_behavior
    @ requires strArr.length >= limit;
    @ signals_only ArrayIndexOutOfBoundsException;
    @ signals (ArrayIndexOutOfBoundsException e) true;
    @ assignable \strictly_nothing;
    @*/
  public boolean add( /*@ non-null @*/ String elem) {
    /*...
    //Branch where elem was added
    //@ set size = size + 1;
    ...*/
  }

  /*@ public normal_behavior
    @ ensures
    @  \result == (\exists int i;
    @                       0 <= i && i < strArr.length;
    @                       strArr[i] == elem);
    @*/
  public /*@ pure @*/ boolean contains(String elem) {/*...
   */}
}
```

Listing 2.2: Example of a JML method contract.

2.3.1 Visibility

In Java, it is a common design philosophy to encapsulate the behavior of objects by declaring their fields private. However, this limits their use in specifications. JML follows access rules identical to Java, meaning that elements within specifications have to be visible to it and that a specification itself also has a visibility. The access modifiers `public`, `protected`, and `private` are explicitly used to define the visibility of specifications and fields. If no modifiers are present, JML defaults to package-private visibility. In addition, in order to not expose implementation details, it is not possible to use private variables directly within protected or public specifications. To change the visibility of a variable only for the specification layer the modifiers `spec_protected` or `spec_public` may be applied.

2.3.2 Purity

The last method in Listing 2.2 is annotated as a *pure*[3] method. This notion is defined next.

Definition 2.5 - Pure Methods

A method is *weakly* pure if it terminates unconditionally and has no visible side effects. A method is *strictly* pure if it terminates unconditionally and alters *no* memory locations whatsoever.

A weakly pure method may not alter the state that was allocated on the heap before the method call; they are only allowed to alter fields of objects they initialize. Similarly, constructors are weakly pure if they only operate on the fields of the objects that they initialize. The wording "no visible side effects" is not accidental; if the method temporarily alters memory locations, but reverts them back before termination, it will still be considered pure in a sequential setting. It will, however, not be considered *strictly pure*; a strictly pure method may not alter *any* heap locations. Specifying that a method is weakly pure is done through the clause `assignable \nothing`, or by annotating it as pure as in Listing 2.2. Annotating methods as strictly pure is done by appending the clause `assignable \strictly_pure` to the method.

Non-pure method specifications often have a restricted set of locations they are allowed to alter, as decided by the keyword `assignable`. The default clause for an `assignable`-expression is `\everything`, meaning that the method may alter any location. The default clause should be avoided when possible. Specifying that a method may only alter the value of a variable `var` is done through `assignable var`, and specifying that a method may only alter the value of elements of array `arr` is done through `assignable arr[*]` for all elements, `assignable arr[i]` for a specific element at index `i`, or through `assignable arr[i..u]` for the range of elements from index `i` to index `u`.

[3]The JML-annotation `pure` is used for *weakly* pure methods.

2.3.3 Normal Behavior

The notation `normal_behavior`[4] denotes a specification case modelling the expected behavior of a method. Normal method behavior indicates that the method terminates, but never never terminates unexpectedly by raising any errors or throwing any exceptions. The keyword `requires` specifies a precondition, while `ensures` specifies a postcondition. The keyword `also` is used to bind together specification cases.

2.3.4 Exceptional Behavior

In some cases, exceptions cannot be avoided. Therefore JML allows users to annotate exception-prone methods accordingly, by annotating the method with an `exceptional_behavior` specification case. Clauses starting with the keyword `signals` introduces *exceptional postconditions*, and has the form
`signals (E e) P`, where `E` is a subtype of `Throwable`. The intention behind such a clause is to specify that if the annotated method throws an exception of type `E`, then the expression `P` has to hold. Specification cases annotated with `normal_behavior` have an implicit `signals (Throwable e) false` exceptional postcondition implying they may never throw exceptions or errors, while specification cases annotated with `exceptional_behavior` have an implicit `ensures false` implying they may never terminate normally. If no behavior type is specified, the method is allowed to both throw exceptions or terminate normally.

2.3.5 Specification-Only Elements

Both model- and ghost fields can be utilized to, respectively, abstract or extend the state of an object, class or interface. Model fields, such as `first` in Listing 2.2, abstract some part of the state, which might be useful in many cases, such as when specifying interfaces or creating less verbose specifications. The behavior of an interface may be specified in terms of model variables, and the classes implementing the interface define `represents` clauses for these model variables, relating them to their own concrete implementation. Similarly, ghost fields, such as `size` in Listing 2.2, extend the state of an object. Ghost fields are able to provide useful information for specification purposes, even when that information is not provided directly by the source code. The value of a ghost field `var` is updated through the statement `set var = expr`, where `expr` is any side-effect free expression with a type conforming to `var`.

2.3.6 Special Constructs

In addition to the aforementioned elements of the JML syntax, the JML language offers a set of special constructs that are useful when creating clauses in specification cases. Some of the most common constructs have been applied in

[4]The non-American spelling *behaviour* may also be used.

Listing 2.2, including `old`, `result`, `invariant` and `forall`. The names of these constructs do well at disclosing their semantics; `old` refers to the field's state prior to the method call, `result` refers to the result returned by the method, `invariant` specifies an invariant, and `forall` is equivalent to the first-order logical quantifier ∀.

During Chapter 4 the reader might encounter elements of the JML syntax that have not been defined in Section 2.3. In such a case, the elements will be explained at a later time. However, the reader is encouraged to study the JML reference manual [50] for further specific details of the language.

2.3.7 Loop Invariants

A loop invariant for a loop is a formula that is valid prior to executing the loop and is maintained valid by the loop body. Loop invariants play a vital part in the verification of programs [22], as verification tools typically require guidance in the presence of loops due to the general impossibility to statically evaluate the loop body repeatedly until the loop condition evaluates to false [17]. If a formula is proven to be an invariant for some loop, then the formula may "replace" the loop during symbolic execution.

Finding suitable loop invariants is considered to be one of the most difficult tasks in formal program verification and it is arguably the one that is least amenable to automation [17]. There exists plenty of resources on generating loop invariants, such as [22, 51], and the strategies used to generate loop invariants for the verification effort in this work are presented in Section 16.3 of [17]. Understanding the processes and strategies of generating loop invariants is not required to enjoy the rest of this thesis, but it is recommended that anyone planning to carry out a verification effort familiarizes themselves with the theory and relevant litterature.

JML offers the ability to annotate Java programs with loop invariants directly through the keyword `loop invariant` or `maintaining`. In addition, in order to prove that a loop terminates, the loop can be annotated with an expression starting with the JML keyword `decreasing`. A formula evaluating to a decreasing value must follow the keyword, and the loop has to terminate when the value of the formula reaches 0. Finally, a loop must be annotated with an `assignable`-clause specifying the locations that may be altered by the loop body[5]. An example of method containing a loop annotated with loop invariants are presented in Listing 2.3 below:

```
/*@ public normal_behavior
  @ ensures
  @  (\forall int i;
  @    0 <= i && i < \result.length;
  @    \result[i] == i*i);
  @ assignable \nothing;
  @*/
```

[5]Local variables need not be explicitly added to the loop's assignable-clause; KeY adds them automatically.

```java
public int[] loopInvariantExample(){
  int[] square = new int[10];
  int i;
  /*@ maintaining 0 <= i && i <= square.length;
    @ maintaining
    @ (\forall int u;
    @   0 <= u && u < i;
    @   square[u] == u*u);
    @ decreasing square.length - i;
    @ assignable square[*];
    @*/
  for (i = 0; i < square.length; i++){
    square[i] = i*i;
  }
  return square;
}
```

Listing 2.3: Example of method with loop annotated with loop invariants.

JML also supports *enhanced for-loops* [75] — the index can then be referred to by the JML keyword `\index`. Finally, it should be noted that the set of valid loop invariants is closed under conjunction [17]: if A and B are loop invariants for the same loop, then $A \wedge B$ is also a loop invariant.

2.4 Java Dynamic Logic (JavaDL)

Java Dynamic Logic (JavaDL) [4] is an instance of first-order logic [7, 38, 39] specially suited for reasoning about sequence of states of Java programs. The application of JavaDL is largely invisible to the user when working with KeY. This is primarily due to KeY's ability to automatically convert JML specifications into JavaDL proof obligations, which the inherent proof assistant can discharge. As such, this section will only explain the elements necessary to enjoy the rest of this thesis; the interested reader is referred to chapter 2 and 3 of [17] for in-depth details.

2.4.1 First-Order Dynamic Logic

This thesis assumes that the reader is to some degree familiar with formal logic. As such, the syntax and intuitive meaning of first-order dynamic logic, the foundation of JavaDL, is primarily given for the sake of completeness. Additionally, in order to reason formally about Java-programs, there is a need to introduce some founding concepts. Most later definitions will build on the following fundamental definitions. First, the notion of type hierarchies is defined. Type hierarchies enables reasoning around types and subtypes.

Definition 2.6 - Type Hierarchies

A *type hierarchy* is a pair $\mathcal{T} = (\text{TSYM}, \sqsubseteq)$, where TSYM is a set of type symbols and $\sqsubseteq$ is the subtype relation. Each type is a subtype of itself (i.e., $\sqsubseteq$ is reflexive), and every subtype may have several other subtypes as long as the transitive property of $\sqsubseteq$ is kept.

Definition 2.7 - Global Type Symbols

There are two global type symbols present in all type hierarchies: the *empty* type $\bot \in \text{TSYM}$ and the *universal* type $\top \in \text{TSYM}$. $\bot$ is a subtype of all types, and all types are subtypes of $\top$, i.e., $\bot \sqsubseteq A \sqsubseteq \top$ for all $A \in \text{TSYM}$.

Further, the vocabulary for first-order dynamic logic is given by its *signature*. Signatures are defined for a given type hierarchy.

Definition 2.8 - Signatures

A signature $\Sigma = (\text{FSYM}, \text{PSYM}, \text{VSYM})$ for a given type hierarchy $\mathcal{T}$ consists of

1. a set of typed function symbols, FSYM. The *arity* of the function is defined by its number of input parameters, while the *domain*, and *co-domain* of the function specifies the types of the input and output, respectively. For instance, for $f \in \text{FSYM}$ the function $f : X \times Y \to A$ takes two input parameters. Therefore, its arity is 2, and the function is 2-ary/binary. Further, the functions domain is the Cartesian product $X \times Y$, with the corresponding co-domain being A.
2. a set of typed predicate symbols, PSYM. For $p \in \text{PSYM}$ the predicate $p(A_1, ..., A_n)$ takes n arguments of types $A_1, ..., A_n$ in that given order. Predicates evaluate to $true$ or $false$.
3. a set of typed logical variable symbols, VSYM. The declaration $v : A$ for $v \in \text{VSYM}$ states that variable v is of type A.

A symbol may not have several instances in $\text{FSYM} \cup \text{PSYM} \cup \text{VSYM}$ with different typing, and all types A_i are different from $\bot$.

Further, we inductively define syntactic categories of terms and formulas for a given type hierarchy and signature.

Definition 2.9 - Terms

Let $\mathcal{T}$ be a type hierarchy, and Σ a signature for $\mathcal{T}$. The set TRM_A contains the terms of type $A \in \mathcal{T}$, and $A \neq \bot$. TRM_A is inductively defined as follows:

- For every variable $v : A \in \mathrm{VSYM}$, we have $v \in \mathrm{TRM}_A$.
- For every function $f(t_1, ..., t_n)$ for $f : A_1 \times ... \times A_n \to A \in \mathrm{FSYM}$ and $t_i \in \mathrm{TRM}_{B_i}$ where $B_i \sqsubseteq A_i$ for all $1 \leq i \leq n$, we have $f(t_1, ..., t_n) \in \mathrm{TRM}_A$.

Definition 2.10 - Formulas

Let $\mathcal{T}$ be a type hierarchy, Σ a signature for $\mathcal{T}$, $v : A \in \mathrm{VSYM}$, and α be an *action*[6] or *event*. The set FML contains the formulas for $\mathcal{T}$ and Σ. Formulas are inductively defined as follows:

- $p(t_1, ..., t_n)$ for $p \in \textsc{psym}$ and $t \in \textsc{trm}$, is a formula.
- If ϕ, ψ are formulas, then $(\neg\phi)$ and $(\phi \wedge \psi)$ are both formulas.
- If ϕ is a formula, then $\forall v; \phi$ and $\exists v; \phi$ are both formulas.
- If ϕ is a formula, then $[\alpha]\,\phi$ and $\langle\alpha\rangle\,\phi$ are both formulas.

The syntax is neither minimal nor maximal. The operators $\neg$ and $\wedge$ constitutes the propositional part of the language, and operators such as $\vee$, $\to$ or $\leftrightarrow$ are syntactic sugar, as semantically equivalent formulas can be expressed using only $\neg$ and $\wedge$. The always false term $false$ can be expressed as $\phi \wedge \neg\phi$, and the always true term $true$ can be expressed as $\neg false$. The first-order quantifiers $\forall$ (read for-all) and $\exists$ (read exists) are used to quantify over variables. The formula $\forall A\ v; \phi$ intuitively states that for all variables v of type A, ϕ holds. Similarly, $\exists A\ v; \phi$ states that there exist a variable v of type A such that ϕ holds. Finally, the dynamic modalities $[\_]$ and $\langle\_\rangle$ will be used when actions are replaced by *legal program fragments* (see Section 2.4.2), and may then be seen as variations of *if-then* statements. If p is a legal program fragment, the formula $[p]\,\phi$ expresses that *if* p terminates, then ϕ holds in the final state. Conversely, $\langle p\rangle\,\phi$ expresses that the program p terminates in a state where ϕ holds.

2.4.2 Extending First-Order Dynamic Logic to JavaDL

Definition 2.8 can be extended to cover Java program variables by defining signatures specifically for the Java type hierarchy displayed in Figure 2.1.

Definition 2.11 - JavaDL Signatures

A JavaDL signature $\Sigma_{\text{JDL}} = (\textsc{fsym}, \textsc{psym}, \textsc{vsym}, \textsc{progsym})$ for the JavaDL type hierarchy $\mathcal{T}_{\text{JDL}}$ for a Java program $prog$ consists of

- a logical signature $(\textsc{fsym}, \textsc{psym}, \textsc{vsym})$ as defined in Definition 2.8,

[6]The formal definition of actions have been omitted, as they will later on be replaced by Java program fragments.

- and a set of nullary nonrigid function symbols, PROGSYM, called *program variables*. The set contains all local variables a declared in $prog$, where the type of $a : A \in$ PROGSYM is given by the declared Java type T as follows:
  - $A = T$ if T is a reference type,
  - $A = boolean$ if $T =$`boolean`,
  - $A = int$ if $T \in \{$`byte, short, int, long, char`$\}$.

In addition, PROGSYM contains the special program variable

$$\texttt{heap} : Heap \in \text{PROGSYM}$$

used to refer to the program heap.

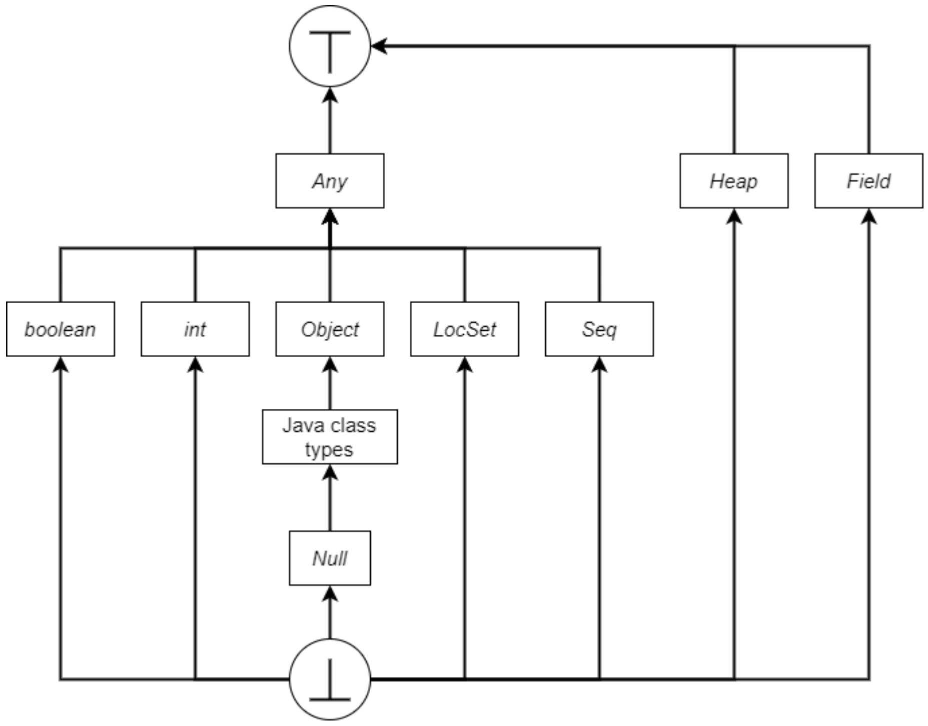


Figure 2.1: Mandatory type hierarchy $\mathcal{T}_{\text{JDL}}$ for JavaDL.

The reader should be aware of the distinction between logical variables in VSYM and the program variables present in PROGSYM. Logical variables must be quantified, but may never occur in programs. Program variables cannot be quantified, but may occur in programs. Terms and formulas are called rigid if they do not contain any occurrences of program variables. Conversely, they are nonrigid if they do contain occurrences of program variables.

The definition of *legal*, or *well-defined*, programs are left out. The reasoning behind this decision is that the Java compiler will be able to sort out program fragments that do not conform to the Java syntax. As such, a Java program that is ill-defined will not compile, and therefore it cannot be executed. Consequently, this thesis will only consider executable programs, as there is little merit to verifying programs that cannot be executed.

The formal definition of legal program fragments is given in Definition 3.2 in section 3.2.3 of [17], while the formal definition of terms and formulas are extended to conform to JavaDL in Definition 3.3 in Section 3.2.4 of the same book. Intuitively, terms and formulas of JavaDL are similar to those for first-order dynamic logic, with the only difference being in Definition 2.10 where *action* α are replaced by legal program fragments (often denoted p).

Before continuing, a final syntactical category is introduced, namely *updates*. Updates are side-effect free expressions denoting state changes. The notation $\{u\}t$, where u is an update and t is a term, restricts the term t to be evaluated in the state produced by update u. Updates can occur in parallel, denoted $(u_1||u_2)$. If updates occurring in parallel clash, i.e., they attempt to assign different values to the same variable, then the value written by u_2 prevails. Updates always terminate.

2.4.3 Sequent Calculus for JavaDL

The sequent calculus [28, 29, 30, 31] are based on the algorithmic manipulation of *sequents*.

Definition 2.12 - Sequents

A sequent has the form $\Gamma \implies \Delta$ in which Γ and Δ are finite (possibly empty) multisets of formulas. The left side of the sequent is the *antecedent*, while the right side is the *succedent*. $\Gamma \cup \{A\}$ or $\Delta \cup \{B\}$ are written as Γ, A and Δ, B respectively.

Intuitively, a sequent represents "provable from", where Γ are assumptions for the set of formulas Δ to be proven. If *all* of the formulas in Γ are true, then *some* of the formulas in Δ are true. Sequents are manipulated by applying rules of a calculus, and sequents with JavaDL formulas are manipulated by rules of the JavaDL calculus. Rule applications to sequents have the form

$$\frac{\texttt{premisse}_1, \ldots, \texttt{premisse}_n}{\texttt{conclusion}}\ \texttt{ruleName}$$

For an example of a rule from the JavaDL sequent calculus, consider the basic `assignment`-rule in the calculus:

$$\frac{\implies \{loc := value\}\,\langle \pi\ \omega \rangle\, \phi}{\implies \langle \pi\ loc\ =\ value;\ \omega \rangle\, \phi}\ \texttt{assignment}$$

Here, ($loc\ =\ value;$) is the *active* statement, while the symbols π and ω are program constructs. π refers to nonactive prefix of Java code, while ω refers to the remainder of the Java code after the active statement. The rule expresses that evaluating $\{loc := value\}\,\phi$ in a state is equivalent to valuating only ϕ in a modified state where *loc* has the value *value*, i.e., it turns the assignment into an update.

Some rules of the JavaDL calculus have restrictions. For instance, the `assignment` rule is subject to the restrictions placed on rules of equality. The restrictions placed on rules of equality ensures that an equality $t_1 = t_2$ may only be used for rewriting if both t_1 and t_2 are rigid, i.e., they do not contain program variables, or the term being replaced is (1) not in the scope of two different program modalities, (2) not in the scope of two different updates.

In KeY, rules of the JavaDL calculus are implemented as *taclets*, which are presented in Section 3.3.

3 The KeY System

KeY is well-documented in [17], and the reader is advised to study this book for a thorough understanding of the tool. The purpose of this section is to present the most central concepts that the KeY System is built upon.

3.1 Introduction

As mentioned in Section 2.1, deductive verification is based on discharging proof obligations extracted from software. The KeY System creates proof obligations from JML annotated Java source code by converting JML contract annotations into JavaDL contracts before turning JavaDL contracts into proof obligations. This process is the focus of Section 3.2.

When proof obligations have been extracted through the aforementioned process, they can be automatically or interactively discharged with the help of KeY's implemented *proof assistant*. This proof assistant is based largely on a formalism known as *taclets*. Therefore, an introduction to taclets and their role in the verification process in given in Section 3.3.

3.2 From JML Annotations to Proof Obligations

The first step towards converting JML annotations to proof obligations is to translate JML contract annotations into JavaDL contracts. These contracts can be further encoded into proof obligations in the shape of JavaDL formulas.

JML is a feature-rich specification language designed with redundancy, i.e., keywords can be syntactically different while being semantically equal. This often improves the readability of specifications, while toughening the task of the verifier. Therefore, to make things simpler for the verifier, the initial JML contract annotations go through a normalization process before being translated into JavaDL contracts. The normalization process consists of the following:

1. expand nested specifications by creating specification cases where each sub-clause are extended with the shared clauses,
2. make implicit non-null, object invariants, behavior and `signals_only` specifications explicit,
3. expand purity modifiers with the clauses `assignable \nothing` and `diverges false`,
4. add default clauses,
5. contract multiple clauses, and
6. separate verification aspects by separating functional and dependency contract, and splitting possibly diverging contracts.

The normalization process may return one or more separate JavaDL contracts for a given JML contract in which all needs to be satisfied. For instance, the normalization process for the JML contract specification

```
/*@ behavior
  @ requires expr;
  @ ensures expr;
  @ assignable loc;
  @ accessible loc;
  @*/
```

will return a *behavior operation contract* and an *accessible clause*. The JML keyword `accessible loc` defines the set of locations `loc` that the observable result of the method can be reliant on. In order to prove that some method conforms to the JML contract specification, the method has to conform to both the behavior operation contract and the accessible clause.

After the normalization process concludes, the special cases of contracts for constructors, model methods and model fields are dealt with. Eventually, the end-product will be one or more JavaDL contracts. As with JML contracts, there are separate JavaDL contracts for the behavioral effects of a method and the dependency of a query method. These are categorized as functional method contracts and dependency method contracts, respectively. Their definitions are given as:

Definition 3.1 - Functional Method Contracts

A functional JavaDL method contract for a method or constructor `ReturnType method(Type1 param1, ..., TypeN paramN)` declared in some class C is a quadruple

$$(pre, post, mod, term)$$

where

- pre is a formula in dynamic logic representing a precondition,
- $post$ is a formula in dynamic logic representing a postcondition,
- mod is a modifier set that is either a term describing the set of heap locations that may be changed by the method, or the string StrictlyNothing,
- and $term$ is a termination witness providing an argument for the method's termination.

All contract components may refer to the special program variables `self` (unless `method` is static), `heap` and to the program variables `param1, ..., paramN`.

Definition 3.2 - Dependency Contracts

A dependency JavaDL method contract for a method `ReturnType method(Type1 param1, ..., TypeN paramN)` is a triple

$$(pre, term, dep)$$

where

- pre is a formula in dynamic logic representing a precondition,
- $term$ is a termination witness providing an argument for the method's termination,
- and dep is a term describing the set of heap locations in which the method dependens on.

All contract components may refer to the special program variables `self` (unless `method` is static), `heap` and to the program variables `param1, ..., paramN`.

To verify JML and JavaDL method contracts, they are encoded into proof obligations, i.e., formulas of JavaDL. The corresponding proof obligations are valid iff the method implementation is correct with respect to the method contract. In fact, KeY defines a method implementation as correct when all its proof obligations are valid. Otherwise, if the formula is falsified, the counterexample is a proof that the contract is not correct. Proof obligations thus define a semantics for JavaDL contracts: A method implementation fulfills its formal contract if and only if the corresponding JavaDL proof obligation is universally valid. The general idea of JavaDL proof obligations is to show that the precondition implies that the postcondition holds after the execution of the method, as introduced in Section 2.1.2. A simplified formula illustrating the idea is given in 3.3 below. The definitions employed by KeY are found in Section 8.3 of [17].

Definition 3.3 -
Simplified Proof Obligations for Functional Correctness

Given the functional method contract

$$(pre, post, mod, term)$$

for functional correctness of method `m`, the simplified proof obligation for the method contract is expressed as the following formula in JavaDL:

$$pre \rightarrow \langle \mathsf{res} = \mathsf{self.m}(\mathsf{p}_1, \ldots, \mathsf{p}_n);\rangle\, post \wedge frame$$

The formula $frame$, called the *framing condition*, ensures that the method does not change any locations that are not in the modifier set mod.

3.3 Taclets

In order to reason in any given logic, a class of tools known as *calculi* is employed. A calculus allows us to determine the validity of a logical formula utilizing purely syntactic operations. As mentioned in Section 2.1, Hoare proposed in [40] a deductive system, i.e., a calculus consisting of a set of inference rules and axioms for reasoning about the correctness of computer programs. Having such a calculus at hand enabled, in theory, to prove the validity of complex software. The first proofs were completed by applying inference rules to annotated programs using pen and paper. However, the field of formal verification has matured immensely [35] over the decades. Parts of this growth can be attributed to the advent of automated tools, such as SMT-solvers [16], model checkers [36], and various proof assistants [8, 17, 60, 61].

The KeY Project comes with a proof assistant, the KeY System, that is able to automatically apply inference rules of the JavaDL sequent calculus 2.4.3 to formulas of JavaDL. Furthermore, the proof assistant is designed to comply with the following requirements:

1. First, as there is a a need for a considerable number of rules in a full-scale verification effort, the proof assistant must understand a language where new rules can easily be written or derived, as opposed to hard-coded.
2. Second, when supplementing the verifier with a new rule (or lemma), there must not be any uncertainty about whether the rule is unsound. Therefore, the proof assistant needs to be able to determine whether a new rule is sound.
3. Third, to allow for interactive theorem proving, the complexity of any given rule has to be restricted in such a manner that the user interaction is kept clear and concise.
4. Finally, the formalization has to enable the automation of as many proof tasks as possible. This includes simplification, symbolic execution and automated proofs or decision procedures for simpler fragments of the logic and theories.

In order to meet these requirements, KeY implements calculus rules in a formalism known as *taclets*. In the coming section, this formalism will be introduced through example.

3.3.1 A Taclet Tutorial

Taclets are a domain specific language for programming proof modification steps, developed as part of the KeY Project. A taclet schematically describes a set of sequent calculus rules, and by instantiating the schema variables with concrete syntactical elements, the taclet itself becomes a concrete applicable rule. Once a complete taclet instantiation has been found applicable, it can be used to perform a step in the sequent calculus.

Taclets are used in the KeY System for

1. defining first-order calculus rules,
2. defining rules of the JavaDL calculus,
3. introducing data types and decision procedures, and
4. to give users the possibility to define and reason about new logical theories.

Theories are introduced by giving a vocabulary of types and functions, a set of axioms defining the semantics, and a set of derived rules that are suitable for the construction of actual proofs. Both axioms and derived rules are formulated as taclets in KeY.

In order to provide a more practical introduction to taclets, the following presents an example by implementing a taclet for a basic theory of lists. A list is defined through a type *List*, and the two constructor symbols

$$nil \; : \; List$$

$$cons \; : \; int \times List \rightarrow List.$$

The properties of types are expressed through axioms, which eliminate all interpretations of the constructor symbols that are inconsistent with said properties. The desired properties for a type is specified on a case-by-case basis. In this example, the two desired properties of type *List* are

1. every list can be represented as a term consisting of *nil* and *cons*, and
2. the representation of a list is *unique* assuming a unique representation of the integers.

The axioms needed can be formalized as such:

Axiom 3.1 - Induction on type *List*

$$\Big(\phi\,[List \leftarrow nil] \land \forall List\, l,\ \forall int\; a;\ \big(\phi \rightarrow \phi\,[l \leftarrow cons(a,l)]\,\big)\Big) \rightarrow \forall List\, l; \phi$$

Axiom 3.2 - *List* containing an element is not *nil*

$$\forall List\, l; \forall int\; a; \big(nil \neq cons(a,l)\big)$$

Axiom 3.3 - *List* is injective

$$\forall List\, l_1, l_2; \forall int\, a_1, a_2; \big(cons(a_1, l_1) = cons(a_2, l_2) \rightarrow a_1 = a_2 \land l_1 = l_2\big)$$

Axiom 3.2 expresses that the ranges of *cons* and *nil* do not overlap, i.e., a list containing an element cannot be *nil*, and Axiom 3.3 states that *cons* is injective. Together, the axioms imply that two lists are equal if and only if they have the same number of elements, and the elements coincide. Axiom 3.1 is the induction axiom reflecting that any element of the list data type can be constructed using *nil* and *cons*. The notation $\phi\,[List \leftarrow nil]$ represents the result of replacing every occurrence of type *List* in formula ϕ with *nil*, i.e., it represents the empty list. In full, the axiom states that if a formula ϕ holds for the empty list and holds for any a, then ϕ holds for all lists. Furthermore, the axiom represents an *axiom schema*, as it is formulated with schematic variable ϕ representing an arbitrary formula.

The taclet for the simplified theory of lists in Listing 3.1 contains the principal elements of taclets. The block `\sorts` declares the theory type, whereas `\functions` declares the available function symbols and their signature. The keyword `\unique` declares a function to be unique; a unique function is defined to be injective, meaning the values of two distinct `\unique` functions are never equal. Further, `\find` defines a pattern that must occur in the sequent to which the taclet is supposed to be applied. The expression matched by `\find` is called the *focus* of a taclet application. The keyword `\replacewith` creates a new proof goal by replacing the expression matched in `\find` with the expression in `\replacewith`. The operator `{\subst x; t}` expresses substitution of a variable `x` with a term `t`. Finally, `\schemaVar` states that the variable is a schema variable, and `\varcond` specifies conditions that have to hold for admissible instantiations of the schema variable of a taclet.

```
\sorts {
    List;
}

\functions {
    \unique List nil;
    \unique List cons(any, List);
}

\axioms {
    list_induction {
        \schemaVar \formula phi;
        \schemaVar \variable List l;
        \schemaVar \variable int a;
        \find( ==> \forall l; phi )
        \varcond(\notFreeIn(a, phi))

        \replacewith( ==> {\subst l; nil} phi
                        & \forall l; \forall a;
                            (phi -> {\subst l;
                                    cons(a, l)}
                                    phi) )
    };
}
```

Listing 3.1: Taclet for simplified theory of lists.

Appending more functionality and lemmas to a taclet is simple. Listing 3.2 displays the result of adding new rules and axioms to support the addition of the new function `int length(List)`. Keep in mind that the taclet in Listing 3.2 should be seen as an extension to the taclet in Listing 3.1, and does not alone represent a valid taclet.

```
\functions {
    int length(List);
}

\axioms {
    length_nil {
        length(nil) = 0
    };

    length_cons {
        \forall List l; \forall any a;
                length(cons(a, l)) = 1 + length(l)
    };
}

\rules {
    length_nil_rw {
        \find ( length(nil) )
        \replacewith( 0 )
    };

    length_cons_rw {
        \schemaVar \term any a;
        \schemaVar \term List l;
        \find( length(cons(a, l)) )
        \replacewith( 1 + length(l) )
    };
}
```

Listing 3.2: Taclet for simplified theory of lists with length.

3.3.2 Soundness of Taclets

For the proofs generated by KeY's proof assistant to be of value, the rules applied by the proof assistant must be sound. As the rules applied by the proof assistant are represented as taclets, there is an inherent need to be able to reason about and prove soundness of taclets.

In order to reason about the soundness of taclets, KeY's proof assistant implements a two-step translation for encoding taclets into *first-order soundness proof obligations*. The translation consists of

1. translation of taclets into *meaning* formulas, such that a taclet is sound iff its meaning formula is valid, and

2. a second transformation that handles the elimination of schema variables in meaning formulas.

Formulas in point (1) above are named *meaning formulas* as they capture the semantic meaning of the different clauses composing a given taclet. These formulas still contain schema variables, meaning that to prove validity, higher-order proof methods are needed. Therefore, the second transformation removes the schema variables such that the end result is a purely first-order formula which is valid if the original formula is valid. However, one cannot simply remove such variables; schema variables are placeholders for syntactic construct like terms, programs or formulas.

In order to prove soundness of a given taclet, one must prove that the taclet's meaning formula is valid for all possible instantiations. Albeit somewhat complicated, well-known techniques such as induction over terms [14] are applicable. However, more preferable methods such as replacing quantification through the introduction of Skolem symbols [65] are possible to some degree: variables can be replaced by a fresh logical variable, terms can replaced by Skolem functions [65], and formulas can be replaced flexible predicate symbol.

Unfortunately, soundness proof obligations cannot be generated for all taclets in KeY. Specifically, taclets containing (1) program modalities in any clause, (2) variable conditions other than `\new` and `\notFreeIn`, (3) meta-functions, (4) generic sorts, or (5) schema variables other than `\term`, `\formula` and `\variable`, are not guaranteed to be verifiable with respect to soundness. As such, these artifacts should be avoided when creating custom taclets.

3.4 Limitations

KeY's proof assistant imposes some limitations on the target code, as there are some common elements of Java programs that are not supported. These limitations are listed in Table 3.1 below. For a thorough review of elements supported by KeY, the reader is refereed to [17].

Library Methods	KeY will throw an error when libraries outside of KeY's classpath are used. The recommended workaround is to create *stubs* for constructors, fields and methods of deployed library code.
Generics	Generics are unsupported by KeY. The purpose of generics in Java is to enable types to be parameters when defining classes, interfaces and methods [13]. The benefits are (1) stronger compiler-time type-checking, (2) eliminating casts, and (3) generics algorithms and methods. However, when a system has been created, any generic classes, interfaces and methods that have been utilized, can be statically substituted for the concrete types used.
Floating Point Types	Floats are unsupported by KeY. The creators have not given any specific reason, but it can be speculated that complexity is one of the primary reasons; floating points are generally considered difficult to formally verify [58].
Multi-threading	KeY does not handle multiple threads in programs. However, there exist other tools that assist in verifying concurrent properties of Java programs. One prominent tool is Java PathFinder [72].
`try-with-resources` and `multi-catch`	Both unsupported. Reason unknown.
Java 8 Features	Not supported. Reason unknown.

Table 3.1: Unsupported features of the KeY System.

4 Verifying EVA Resultat

4.1 Introducing EVA Resultat

The target of verification, EVA Resultat [53], is presented in this section. EVA is the fundamental IT-system used by municipalities and counties to prepare for and implement elections in Norway. The system was first deployed and utilized in 10 municipalities during the 2011 election, and has since 2013 been used for all government elections in Norway. In short, all municipalities, counties and the Sami parliament collect all the information regarding practical implementation aspects of the election process using EVA, including the location and opening hours of polling stations and details of electoral lists submitted by disparate political parties/groups. Collectively, this information constitutes the basis for ballots and voting cards, and is a deciding factor in how EVA will further assist the municipalities and counties in tallying the election. Finally, EVA provides election forecasts and calculates how many representatives are to be returned from each electoral list. EVA is split into three separate software applications:

EVA Admin	The core support system for administrating the electoral process in municipalities and counties. All core data taking part in executing the election is registered in this system, as well as the registration of votes and tally. EVA Resultat retrieves the data used for calculating the seat allocation from EVA Admin.
EVA Scanning	The software managing the tally of the votes given by electors.
EVA Resultat	Responsible for providing forecast of election results, as well as calculating the election results based on the data received from EVA Admin. This application will be the principal verification target of this thesis.

Considering that EVA constitutes the primary machinery responsible for calculating the result of an election, a profound understanding of the electoral process would be required when designing the system. As this thesis is mainly focused on verifying distinct system properties, only a superficial review will be presented with the focal point being Sainte-Laguës's modified method. For more specific details and more in-dept explanations, the reader is referred to [1] and [69].

The system itself considers the electoral process as part of its specification, and the behavior of the system is expected to conform to the description written in the Election Act [52]. The official implementation, EVA-Resultat, has been implemented in Java where external libraries such as Spring [73], Hibernate [62] and JavaX [32] has been utilized extensively. As this system is the primary verification target of this thesis, a presentation of the architecture and implementation will be the primary focus of Section 4.3.

4.2 The Norwegian Electoral Process

The electoral term in Norway is four years both for members of parliament, and for members of municipal and county councils. Elections to municipal and county councils are conducted at the same time, and are held midway in the parliamentary electoral term. Parliamentary elections occur every four years, and local elections, i.e., elections for municipal councils and county councils, occur two years after each parliamentary election. As a consequence, Norwegian governmental elections occur every two years.

To keep consistent with the wording used in the Electoral Law [52], statements of the form ”county seat X is awarded to electoral list Y” will be used to express that ”the political party that submitted electoral list Y will decide who becomes representative number X for the county council”. At times, the term ”constituency representative” may be used instead of ”county seat”.

The electoral process itself is based on two central principles:

1. Direct election; electors vote directly for their preferred representatives by casting their votes for an electoral list. The various political parties or groups submit their own electoral lists specifying their favored candidates, which, when applicable, are arranged in prioritized order. Such an arrangement is in contrast to other electoral processes, such as the U.S. presidential election [20, 44], where voting is performed indirectly through the Electoral College.

2. Proportional representation; representatives are assigned to the electoral lists in relation to the number of votes received by the respective political parties and groups. Proportional representation is an attempt to guarantee that each electoral list gets its ”fair share” of representatives; if an electoral list receives half of all the votes cast in an election, then half of the resulting governing body will consist of representatives belonging to the corresponding political party. The result of the election is therefore independent of the localization of voters within the region, thereby rendering gerrymandering irrelevant. Again, this is in contrast to the U.S. election where the election is dependent on the localization of voters within a region [10].

The Norwegian parliamentary election is held to decide representatives for the 169 seats of *Stortinget*, i.e., the legislative branch of Norway. During the parliamentary election Norway is divided into 19 separate constituencies, one for each county, and each constituency is responsible for assigning a given number of representatives to Stortinget. This number of members may differ between constituencies, as it is dependent on the population and area of the constituency. Of the 169 representatives elected, 150 are elected as constituency representatives while the remaining 19, one seat from each constituency, are elected as representatives *at-large*. Constituency representatives and representatives at-large hold the exact same power. The constituency representatives are partitioned in accordance to Sainte-Laguës's modified method, and the representatives at-large are decided by an electoral committee, the National Electoral Committee, appointed by the King in the same year as the parliamentary election [43, 52]. However, the allocation of representatives at-large does have strict procedural guidelines in place, and must follow the procedure represented by the pseudocode in Listing 4.3. Further, each municipal and county council is responsible for appointing Municipal Electoral Committees and County Electoral Committees.

Sainte-Laguës's modified method implies that the total vote polled by each electoral list is divided by the odd numbers starting with 1.4, i.e., 1.4, 3, 5, 7 and so forth. Each total vote polled shall be divided as many times as necessary to find the number of seats the electoral list shall have. The first seat is awarded to the electoral list that has the largest quotient. The second seat is awarded to the electoral list that has the second largest quotient and so forth. If two or more electoral lists have the same quotient, the seat is awarded to the electoral list that has polled the largest number of votes. If they have polled the same number of votes, it is determined by lot to which electoral list the seat shall be awarded.

Pseudocode for Sainte-Laguës's modified method is found in Listing 4.1 and Listing 4.2.

```
SainteLaguesModifiedMethod ($\mathcal{R}$, $\mathcal{C}$)
  input:  $\mathcal{R}$, the number of representatives to give/the
                number of available seats.
          $\mathcal{C}$, constituency to return representatives from.
  output: $\mathcal{M}$, map with key-value pairs
          <Electoral List, Number of representatives given>
  begin
    $\mathcal{M} \leftarrow \emptyset$;
    $\mathcal{Q}$ : Stack of pairs <Electoral List, Quotient> $\leftarrow \emptyset$;
    $\mathcal{Q} \leftarrow$ CalculateQuotients($\mathcal{C}$, $\mathcal{R}$);
    $\mathcal{Q} \leftarrow$ $\mathcal{Q}$.sortDesc();

    while $\mathcal{Q}$.size() < $\mathcal{R}$ do
      pair $\leftarrow$ $\mathcal{Q}$.pop();
      $\mathcal{M}$.insertOrIncrement(pair.getKey());
    end

    return $\mathcal{M}$;
  end
```

Listing 4.1: Pseudocode for Sainte-Laguë's modified method.

```
CalculateQuotients ($\mathcal{C}$)
  input:  $\mathcal{C}$, constituency to return representatives from.
          $\mathcal{R}$, the number of representatives to be awarded.
  output: $\mathcal{Q}$, stack of pairs <Electoral List, Quotient>.
          $\mathcal{R}$ number of quotients are calculated for every
          electoral list in constituency $\mathcal{C}$ in case an
          electoral list has received enough votes to win
          all representatives.
  begin
    $\mathcal{Q} \leftarrow \emptyset$;
    $\mathcal{L}$ : Electoral lists for the
        given constituency $\leftarrow$ $\mathcal{C}$.getElectoralLists();

    quotientsToCalculate $\leftarrow$ $\mathcal{L}$.length $\times$ $\mathcal{R}$;
    i : iteration counter $\leftarrow$ 0;

    while i <= quotientsToCalculate do
      divisor $\leftarrow$ (i == 0) ? 1.4 : (2*i + 1);
      foreach el $\in \mathcal{L}$ do
        quotient $\leftarrow$ el.getVotes()/divisor;
        pair $\leftarrow$ <el,quotient>;
        $\mathcal{Q}$.push(pair);
      end
      i $\leftarrow$ i + 1;
    end

    return $\mathcal{Q}$;
  end
```

Listing 4.2: Pseudocode for calculating quotients.

There is one element in the pseudocode that is not self-explanatory, mainly `insertOrIncrement`. The function `insertOrIncrement` either inserts a new key (i.e., electoral list) or, if the key is already present in the map, increments the key's value to represent that the electoral list is awarded a new representative. As such, $\mathcal{M}$ functions as a counter of the number of representatives awarded for each electoral list. Further, it is assumed that the function `sort` sorts based on number of votes received in the case where two quotients are equal.

It is important to calculate *enough* quotients. To illustrate why the term *enough* is rather dynamic, consider the example in Table 4.7. Here, one could stop at iteration 4 and award the 16th seat to the Yellow Party. However, this would be a mistake, as during the 5th iteration it becomes clear that the correct decision is to award the 16th seat to the Green Party. One possible solution is to terminate only when enough quotients to fill all seats are calculated, and the highest quotient in the current iteration is lower than the lowest quotient in the previous iteration. However, the solution implemented in EVA is to calculate as many quotients as there are seats available for every electoral list. That is, following the example in Table 4.7, 16 quotients would be calculated for every electoral list, totalling in a total of 64 quotients.

When concerned with parliamentary elections, 19 of the 169 representatives are elected for seats at-large. Recall that according to electoral law, the representatives will be decided by the National Electoral Committee appointed by the King based on transcripts that are sent in from the County Electoral Committees. These transcripts contain the number of constituency representatives that has been assigned each electoral list in the County Electoral Committees' given counties. Further, The National Electoral Committee determines how many seats at-large are allocated to each electoral list as follows:

1. Allocate, using Sainte-Laguës's modified method, all 169 representatives among the electoral lists when the entire nation is considered as a single constituency. The number of representatives `rep` awarded for an electoral list `EL` using this method is denoted `EL.rep`.

2. For each electoral list, compare `EL.rep` to the number of constituency seats that has been given electoral list `EL`, denoted `EL.con`.

   (a) If `EL.rep` is lower than `EL.con`, then the electoral list has been overly represented. Redo the allocation in Point 1 for the remaining representatives, and disregard the overly represented electoral list `EL`.

   (b) If `EL.rep` is higher than or equal to `EL.con`, then the electoral list will be awarded a number of representatives at-large corresponding to the difference between `EL.rep` and `EL.con`.

Note the distinction between allocating constituency representatives and representatives at-large: constituency representatives are allocated based on votes tallied in each distinct constituency, while representatives at-large are allocated based on *all*[7] tallied votes.

After it has been decided how many representatives at-large each electoral list shall have, then it has to be decided in which constituencies they shall be awarded from. This is decided by the following process:

1. First, calculate the *county factor* for each constituency by dividing the total tallied votes cast in the constituency by the number of constituency representatives returned from the constituency.

2. Secondly, a quotient shall be calculated for each electoral list in each constituency.

   (a) If an electoral list *has not* received any constituency representatives from a given constituency, then the quotient for the electoral list in the given constituency shall be based on the number of votes tallied for the electoral list in the constituency. The quotient for the given electoral list in a given constituency is then

$$\frac{\text{Votes tallied for the electoral list in the constituency}}{\text{The constituency's county factor}}.$$

[7]All votes *except* protest votes.

(b) If a electoral list *has* received $\mathcal{X}$ number of constituency representatives from a given constituency, then the quotient for the electoral list in the given constituency shall be based on the number of votes tallied for the electoral list in the constituency divided by $(\mathcal{X} \times 2) + 1$. The quotient for the given electoral list in a given constituency is then

$$\frac{\text{Votes tallied for the electoral list in the constituency}}{((\mathcal{X} \times 2) + 1) \times \text{The constituency's county factor}}.$$

3. Collect all quotients — each constituency will supply one quotient for each electoral list. The first representative at-large is awarded the electoral list and constituency that has the highest quotient. The second representative at-large is awarded the electoral list and constituency that has the second highest quotient. Every constituency will receive only one representative at-large, and an electoral list cannot receive any representatives at-large if it has tallied less than 4% of the total votes tallied nationwide.

Pseudocode for the at-large calculation is presented in the following Listing 4.3.

```
SeatAtLargeAllocation ($\mathcal{C}$)
  input:
    $\mathcal{C}$ : Transcript from constituencies.
    The statement $\mathcal{C}$.getCons(EL) returns number of total
    constituency representatives awarded to electoral list
    EL.
    The statement $\mathcal{C}$.getAllCons() returns a list of all
    constituencies that has awarded constituency
    representatives.
    Further, each constituency con has information
    regarding total number of votes cast in constituency for
     the election and number of constituency representatives
     awarded. This information is retrieved through con.
    getVotes() and con.getReps() respectively.
    The statements con.getVotes() and con.getReps() can
    take an electoral list as a parameter, and will then
    return the number of votes cast for that electoral list
    in the constituency, and the number of constituency
    representatives awarded that electoral list in the
    constituency, respectively.

  output:
    Void method; no return value. Method allocates
    representatives at-large from each constituency among
    the electoral lists.

  begin
    /* Determine how many representatives at-large each
     * electoral list shall have. */
    tot : Number of total representatives ← 169;
    REP : Map with key-value pairs
          <Electoral List, Number of representatives> ← ∅;
    REP ← SainteLaguesModifiedMethod(tot, NORWAY);

    foreach el ∈ REP do
      diff ← REP.get(el) - 𝒞.getCons(el);
      if diff < 0 do
        tot ← tot - 𝒞.getCons(el);
        /* The expression NORWAY\el means "consider Norway
         * as a single constituency, but disregard
         * electoral list el from calculations". */
        REP ← SainteLaguesModifiedMethod(tot, NORWAY\el);
        redo loop;
      else
        REP.put(el, diff);
        /* REP now contains how many at-large
         * representatives (diff) each
         * electoral list (el) shall have. */
      end if
    end foreach

    // Calculate county factors
    CONS  : List with all constituencies ← 𝒞.getAllCons();
    C_FAK : Map with key-value pairs
            <Constituency, County Factor> ← ∅;

    foreach con ∈ CONS do
      totalVotesCast ← con.getVotes();
      consRepsGiven ← con.getReps();
      factor ← totalVotesCast / consRepsGiven;
      C_FAK.put(con, factor);
    end foreach;

    // Calculate quotients

    /* The notation 𝒬[con][el] denotes the quotient for
     * electoral list el in constituency con */
    𝒬 : Map for quotients ← ∅;
    foreach con ∈ CONS do
      foreach el in REP do

        if (REP.get(el) = 0) do
          continue;
        end if;


        if con.getReps(el) = 0 do
          𝒬[con][el] ← con.getVotes(el) / C_FAK.get(con);

        else
          denom ← $((\texttt{con.getReps(el)}\times 2)+1)\times$C_FAK.get(con);
          $\mathcal{Q}$[con][el] ← con.getVotes(el) / denom;

        end if

      end foreach;
    end foreach;

    // Award representatives at-large
    $\mathcal{Q}$ ← $\mathcal{Q}$.sortDesc();
    foreach quotient ∈ $\mathcal{Q}$ do
      con ← quotient.getCon();
      el  ← quotient.getEl();
      if hasReceivedAtLargeRepresentative(con) do
        continue;
      else if hasLessThanFourPercent(el) do
        continue;
      else
        awaredSeatAtLarge(con, el);
      end if;
    end foreach;

  end
```

Listing 4.3: Pseudocode for seat at-large allocation

4.2.1 Seat At-Large Calculation Example

The following example illustrates how the at-large allocation would occur for a parliamentary election for a nation with 5 counties/constituencies and 3 political parties. The process is extendable to as many constituencies and political parties as necessary. Pseudocode for the at-large allocation process is found in Listing 4.3 starting on Page 36.

Consider a scenario where 3 political parties — Red Party, Blue Party and Yellow Party — are running for the 155 parliamentary seats, of which 5 are seats at-large. During the election, the nation is divided into 5 constituencies — North, East, Middle, South and West.

The tallied votes are displayed in Table 4.1, and according to the transcripts sent from the County Electoral Committees the constituency representatives have been allocated as displayed in Table 4.2. The result of applying Sainte-Laguës's modified method for all 155 representatives with the votes tallied in Table 4.1 is displayed in Table 4.3.

	North	East	Middle	South	West	Total
Red Party	603	27	288	217	861	1996
Blue Party	769	98	67	202	790	1926
Yellow Party	33	73	27	52	210	395
Total	1405	198	382	471	1861	4317

Table 4.1: Tallied votes from parliamentary election.

	North	East	Middle	South	West	Total
Red Party	20	2	4	6	34	66
Blue Party	25	7	1	5	31	69
Yellow Party	1	5	0	1	8	15
Total	46	14	5	12	73	150

Table 4.2: Number of constituency representatives awarded to the political parties from each constituency.

Party	**Representatives**
Red Party	71
Blue Party	69
Yellow Party	15
Total	155

Table 4.3: Result of applying Sainte-Laguës's modified method for 155 representatives based on data in Table 4.1.

Constituency	**County Factor**
North	$\frac{1405}{46} = 30.54$
East	$\frac{198}{14} = 14.14$
Middle	$\frac{382}{5} = 76.4$
South	$\frac{471}{12} = 39.25$
West	$\frac{1861}{73} = 25.49$

Table 4.4: County factors for each constituency.

When comparing the result displayed in Table 4.3 with the allocation of constituency representatives in Table 4.2, it becomes clear that the Red Party will be awarded all 5 representatives at-large. As the Blue Party and Yellow

Party shall receive no representatives at-large, they should be disregarded during further calculations. However, to better demonstrate the at-large allocation, their quotients will still be calculated.

The next step in the process is to calculate all county factors and all quotients. The resulting county factors are displayed in Table 4.4, while the resulting quotients are displayed in Table 4.5.

	North	East
Red Party	$\frac{603}{((20\times2)+1)\times30.54} = 0.4815$	$\frac{27}{((2\times2)+1)\times14.14} = 0.3818$
Blue Party	$\frac{706}{((25\times2)+1)\times30.54} = 0.4532$	$\frac{98}{((7\times2)+1)\times14.14} = 0.4620$
Yellow Party	$\frac{33}{((1\times2)+1)\times30.54} = 0.3601$	$\frac{73}{((5\times2)+1)\times14.14} = 0.4693$
	Middle	South
Red Party	$\frac{288}{((4\times2)+1)\times76.4} = 0.4188$	$\frac{217}{((6\times2)+1)\times39.25} = 0.4252$
Blue Party	$\frac{67}{((1\times2)+1)\times76.4} = 0.2923$	$\frac{202}{((5\times2)+1)\times39.25} = 0.4678$
Yellow Party	$\frac{27}{76.4} = 0.3534$	$\frac{52}{((1\times2)+1)\times39.25} = 0.4416$
	West	
Red Party	$\frac{861}{((34\times2)+1)\times25.49} = 0.4895$	
Blue Party	$\frac{790}{((31\times2)+1)\times25.49} = 0.4919$	
Yellow Party	$\frac{210}{((8\times2)+1)\times25.49} = 0.4846$	

Table 4.5: Quotients from all constituencies for each electoral list.

Finally, the quotients calculated in Table 4.5 can be sorted in descending order, and the representatives at-large can be decided. In the following list, the quotients are listed together with the corresponding constituency, political party and a comment.

1. 0.4919 — West — Blue Party — Disregarded.
2. 0.4895 — West — Red Party — First representative.
3. 0.4846 — West — Yellow Party — Disregarded.
4. 0.4815 — North — Red Party — Second representative.
5. 0.4693 — East — Yellow Party — Disregarded.
6. 0.4678 — South — Blue Party — Disregarded.

7. 0.4620 — East — Blue Party — Disregarded.
8. 0.4532 — North — Blue Party — Disregarded.
9. 0.4416 — South — Yellow Party — Disregarded.
10. 0.4252 — South — Red Party — Third representative.
11. 0.4188 — Middle — Red Party — Fourth representative.
12. 0.3818 — East — Red Party — Fifth representative.
13. 0.3601 — North — Yellow Party — All seats allocated.
14. 0.3524 — Middle — Yellow Party — All seats allocated.
15. 0.2923 — Middle — Blue Party — All seats allocated.

It can then be concluded that the Red Party is allocated one representative at-large from each constituency, for a total of five. The quotients partaking in this decision are quotients 2, 4, 10, 11 and 12 above.

4.2.2 Local Government Election Example

Local government elections for municipal councils and county councils are similar to the parliamentary rendition. The preeminent dissimilarity arise from how each constituency themselves decide the number of seats. However, this decision is restricted by statutory guidelines. Further, the representatives for the municipal counties are decided by the numbers of votes received by the electoral list in that municipality, and are calculated in accordance to Sainte-Laguës's modified method. When it has been decided how many representatives each electoral list shall have, the Municipal Electoral Committee allocates the seats to the candidates on the electoral list.

Consider a scenario where there is a local election in an arbitrary municipality where 16 seats are up for election. The tallied results are displayed in Table 4.6, the application of Sainte-Laguës's modified method is displayed in Table 4.7, and finally, the summary of the election result is displayed in Table 4.8.

Each party/group can give an increased share of the poll to a limited number of candidates; these candidates are written at the top of the list in boldface. When it has been decided how many representatives each party/group is allocated, the returning of the members is based on the personal votes[8] of each candidate. Candidates whose names appear in boldface get an increase in their personal poll corresponding to 25 per cent of total tally of the party/group they represent. After the increased share of the poll has been calculated, the personal

[8]When casting a vote for an electoral list, the elector can choose to cast a personal vote for a candidate they believe should be the representatives for the electoral list. Personal votes can be cast both for candidates that the electoral list has listed for itself and for candidates that other electoral lists have listed for themselves.

votes given to each candidate are added. Personal votes received from the electoral list's own voters and personal votes received from other lists' voters count equally. The members are returned in order of who has the highest personal poll. To illustrate, the scenario from above is continued; Table 4.9 displays the returning of council members from the Red Party. All factors considered, the municipal council will consist of four members from the Red Party: Alice, Eve, Bob and Boris.

Party	**Votes**
Red Party	200
Blue Party	198
Yellow Party	140
Green Party	217

Table 4.6: Tallied votes from local election.

	Red Party	Blue Party	Yellow Party	Green Party
Total Votes	200	198	140	217
1.4	142 (2)	141 (3)	100 (4)	155 (1)
3	66 (6)	66 (7)	46 (8)	72 (5)
5	40 (10)	39 (11)	28 (15)	43 (9)
7	28 (13)	28 (14)	20	31 (12)
9	22	22	15	24 (16)
11	18	18	12	19

Table 4.7: Result from local election after applying Sainte-Laguës's modified method.

Party	**Seats No.**
Red Party	2, 6, 10, 13
Blue Party	3, 7, 11, 14
Yellow Party	3, 8, 15
Green Party	1, 5, 9, 12, 16

Table 4.8: Tallied votes from local election.

Name of candidate	**Increased party share**	**Personal votes**	**SUM personal poll**	**Returning of members**
1. **Alice**	50	13	63	1. Alice
2. **Bob**	50	2	52	2. Eve
3. Eve	0	56	56	3. Bob
4. Vladimir	0	37	37	4. Boris
5. Boris	0	48	48	5. Vladimir
6. Donald	0	2	2	6. Donald

Table 4.9: Returning of members for the Red Party.

4.3 System Description

The architecture of EVA Resultat is separated into a five core modules composing the system. These modules and a summary of their functionality is given in Table 4.10. The back-end calculates the election result. As such, the functional correctness of the module is crucial; the other modules are primarily auxiliary, e.g., utility-, database-, or web-modules. Although these modules are also prone to error, the type of errors experienced here are generally more *noisy*, e.g., throws exceptions or causes abrupt termination. However, errors in numerical calculations can be more demanding to identify as they are often more silent.

Considering the importance of the back-end module, and the fact that it implements both integer- and floating-point calculations to calculate the quotients of each electoral list, the primary focus will be on this component. The class diagrams for the classes composing the seat-allocation functionality is displayed in Figure 4.1.

Back-end	The central machinery of the application. It handles parsing of JSON-documents from the database, computes the seat allocation according to Sainte-Laguës's modified method, and creates reports for the media and the public. The module is shielded from exterior processes, and interacts only with the database.
Common	Some common utilities, e.g., JSON helpers and custom exceptions.
Database	There are primarily two types of data stored in the database: static data and dynamic data. Static data is comprised of core data from EVA Admin, historical data, and data categorized as *other*. The dynamic data can be more or less mapped to the three functions of the back-end; (1) the parser reads from the database table modified by the receptor, (2) the seat distribution process writes its result to the database, and (3) the reports generated are also written to the database.

Front-end	The front-end is a server responsible for (1) retrieving batches of reports created by the back-end, and (2) divides the batches into singular reports while caching and making them more readable.
Receptor	A web-application/servlet that retrieves information from EVA Admin, and stores the data in a database table. This module is new in the 2019 version; previously EVA Admin stored the data directly into the database itself. This was changed due to performance- and security concerns.

Table 4.10: Summary of core modules of EVA Resultat.

4.3.1 Implementation Details

The system itself is implemented in Java with the help of the Spring Boot framework [73]. This framework contains libraries that do not have any formal specification, and contains features that are not supported by the KeY System. In addition, the implementation utilizes external packages such as Lombok [76], and the usage of Java 8 [32] features are common . The former causes few issues as the features used from the Lombok package, i.e., the annotations `@Getter` and `@Setter`, are purely syntactic sugar used to create less verbose source code. The annotations can be rewritten to standard Java. As for the latter, Java 8 features are fully unsupported. This is not to be confused with Java 8 libraries, as it is possible to create stubs with default contracts for unfamiliar library methods. However, features such as Lambda-functions have to be dealt with by either adding support for such features to KeY, or removing them by downgrading the system to conform to Java 7.

One of the Java libraries that is used extensively in the seat-allocation calculation, is `java.math.BigDecimal` [12]. The two principal reasons for using the BigDecimal-class are: (1) eliminating the risk of errors arising due to integer calculations (e.g. overflows), and (2) to provide a scale of type 32-bit integer. According to the `BigDecimal`-documentation, the scale, if zero or positive, is the number of digits to the right of the decimal point. This is a massive increase in the number of available decimals compared to using floats or doubles.

However useful the `BigDecimal`-class might seem, one can argue that the usage is superficial. Advents of integer calculation overflows are unlikely in this particular closed system; the number of votes in Norwegian election rarely exceeds the value of `Integer.MAX_VALUE`. Also, according to specifications, the implementation is set to have a maximal precision of 20. A maximal precision of 20 ensures that quotients are only considered equal if they are equal up to and including 20 digits right of the decimal point. The idea here is to ensure that two quotients are treated as equals only if they are based on the same divisor and number of votes. However, this seems superfluous as the Electoral Law [69] states that two quotients may be similar. In such a scenario, the seat is awarded

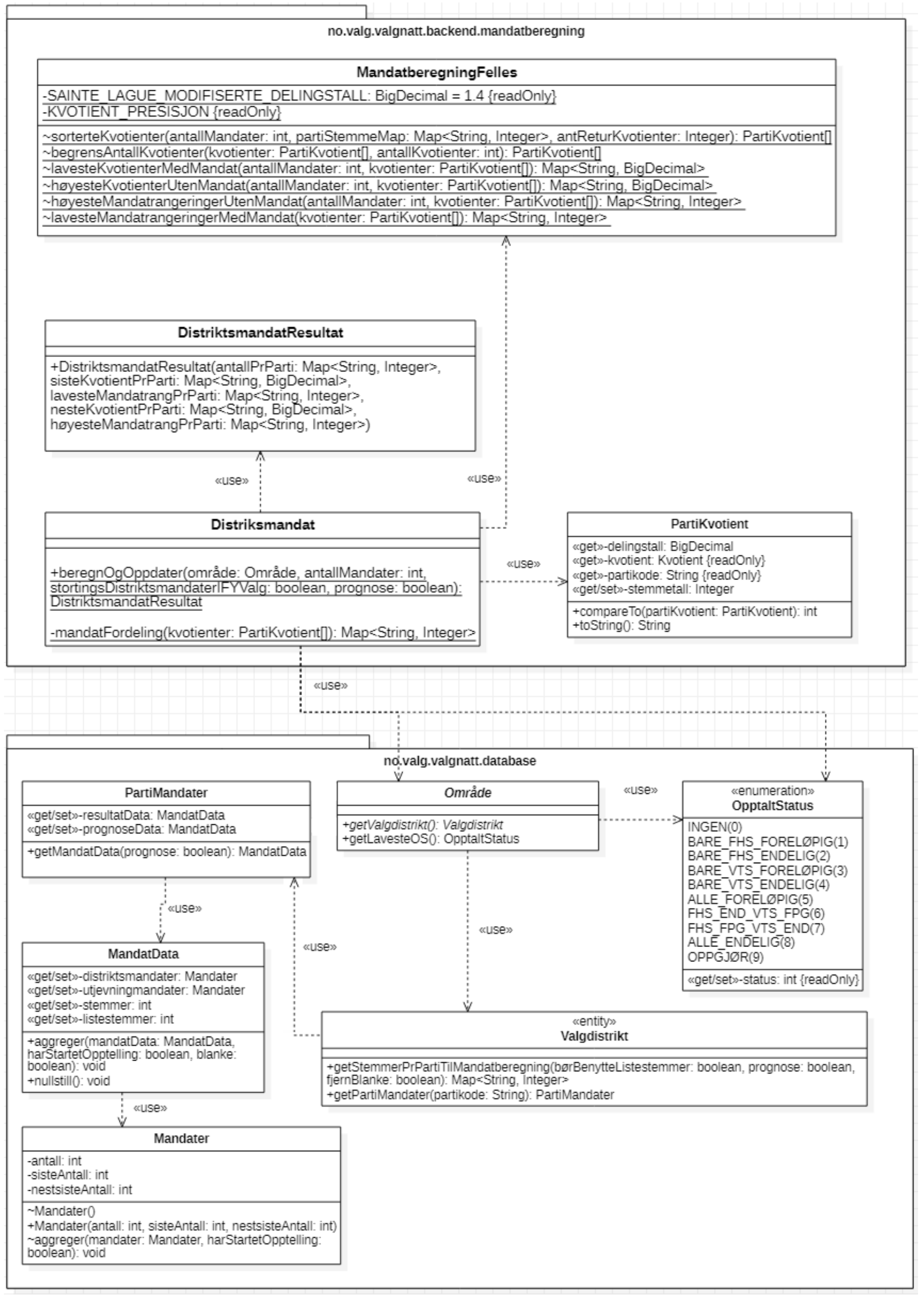

no.valg.valgnatt.backend.mandatberegning
MandatberegningFelles
-SAINTE_LAGUE_MODIFISERTE_DELINGSTALL: BigDecimal = 1.4 {readOnly}
-KVOTIENT_PRESISJON {readOnly}
~sorterteKvotienter(antallMandater: int, partiStemmeMap: Map<String, Integer>, antReturKvotienter: Integer): PartiKvotient[]
~begrensAntallKvotienter(kvotienter: PartiKvotient[], antallKvotienter: int): PartiKvotient[]
~lavesteKvotienterMedMandat(antallMandater: int, kvotienter: PartiKvotient[]): Map<String, BigDecimal>
~høyesteKvotienterUtenMandat(antallMandater: int, kvotienter: PartiKvotient[]): Map<String, BigDecimal>
~høyesteMandatrangeringerUtenMandat(antallMandater: int, kvotienter: PartiKvotient[]): Map<String, Integer>
~lavesteMandatrangeringerMedMandat(kvotienter: PartiKvotient[]): Map<String, Integer>
DistriktsmandatResultat
+DistriktsmandatResultat(antallPrParti: Map<String, Integer>, sisteKvotientPrParti: Map<String, BigDecimal>, lavesteMandatrangPrParti: Map<String, Integer>, nesteKvotientPrParti: Map<String, BigDecimal>, høyesteMandatrangPrParti: Map<String, Integer>)
«use»
«use»
Distriksmandat
+beregnOgOppdater(område: Område, antallMandater: int, stortingsDistriktsmandaterIFYValg: boolean, prognose: boolean): DistriktsmandatResultat
-mandatFordeling(kvotienter: PartiKvotient[]): Map<String, Integer>
«use»
PartiKvotient
«get»-delingstall: BigDecimal
«get»-kvotient: Kvotient {readOnly}
«get»-partikode: String {readOnly}
«get/set»-stemmetall: Integer
+compareTo(partiKvotient: PartiKvotient): int
+toString(): String
«use»
no.valg.valgnatt.database
PartiMandater
«get/set»-resultatData: MandatData
«get/set»-prognoseData: MandatData
+getMandatData(prognose: boolean): MandatData
Område
+getValgdistrikt(): Valgdistrikt
+getLavesteOS(): OpptaltStatus
«use»
«enumeration»
OpptaltStatus
INGEN(0)
BARE_FHS_FORELØPIG(1)
BARE_FHS_ENDELIG(2)
BARE_VTS_FORELØPIG(3)
BARE_VTS_ENDELIG(4)
ALLE_FORELØPIG(5)
FHS_END_VTS_FPG(6)
FHS_FPG_VTS_END(7)
ALLE_ENDELIG(8)
OPPGJØR(9)
«get/set»-status: int {readOnly}
«use»
«use»
«use»
MandatData
«get/set»-distriktsmandater: Mandater
«get/set»-utjevningmandater: Mandater
«get/set»-stemmer: int
«get/set»-listestemmer: int
+aggreger(mandatData: MandatData, harStartetOpptelling: boolean, blanke: boolean): void
+nullstill(): void
«entity»
Valgdistrikt
+getStemmerPrPartiTilMandatberegning(børBenytteListestemmer: boolean, prognose: boolean, fjernBlanke: boolean): Map<String, Integer>
+getPartiMandater(partikode: String): PartiMandater
«use»
Mandater
-antall: int
-sisteAntall: int
-nestsisteAntall: int
~Mandater()
+Mandater(antall: int, sisteAntall: int, nestsisteAntall: int)
~aggreger(mandater: Mandater, harStartetOpptelling: boolean): void

Figure 4.1: UML class diagram of seat-allocating functionality.

to the electoral list that polled the most votes. Further, a scenario where 20 decimal places are actually necessary seems unlikely. When reviewing the result of prior elections, this becomes increasingly clear. For instance, in the result for the 2019 municipal election for Oslo, there was only a single case where quotients were not able to be decided by the first decimal point for the first 21 divisors (i.e., the odd numbers from 1.4 to 41). The quotients in question were the 175th and 176th highest quotients, meaning they had no effect on deciding the 58 municipal seats to be awarded.

Another vital Java library is `java.util.Arrays`, as the method `Arrays.sort(Object[] a)` is used to sort the quotients. The elements of array `a` implements the interface `java.lang.Comparable<T>`, meaning that all elements of array `a` has implemented a `int compareTo(T o)`-method. This method is utilized by `Arrays.sort(Object[] a)` to sort the elements of the array. In EVA, the quotients are realized by the class `PartiKvotient` which implements `Comparable<PartiKvotient>`. Further, the method that creates and sorts quotients is `MandatberegningFelles.sorterteKvotienter(...)`. The latter method returns an array `PartiKvotient[]`. It is vital that the quotients in this array is sorted in descending order, otherwise representatives might be awarded to the wrong electoral lists. The sorted order is, as mentioned above, controlled by the method `PartiKvotient.compareTo(PartiKvotient pk)`, which again is based in the method `BigDecimal.compareTo(BigDecimal val)`. In fact, the latter method is the only method from the `BigDecimal`-class that is used to extract information. All other utilized methods from the `BigDecimal`-class are used to manipulate the result of the `compareTo(...)`-method.

4.3.2 Verification Goals and Limitations

This chapter will attempt to reach the research goals listed in Section 1.2 by creating specifications for all methods and classes of the EVA Resultat back-end that concerns seat allocation, and verifying methods and classes that concern the allocation of constituency representatives in an incremental and transparent fashion. Ideally, every desired property of the system should be specified and verified. Unfortunately, when constrained by available time and resources, a subset of methods and classes has be to selectively chosen as verification targets. This work has opted to limit its focus on what is considered the most fundamental aspects of the system's seat allocation functionality. Specifically, the parts of the system that are required for calculating constituency representatives[9] will be subject to verification. The specifications are written to be as prone to automatic theorem proving as possible; some of the proof trees that are deduced from the specifications will be exceptionally large meaning they become increasingly difficult to prove interactively. Further, during the following verification effort, the most central techniques and tools for behavioral verification will be presented, with two exceptions:

[9]The same part of the system is used to calculate results of local elections.

1. Exceptional behavior method specifications are not utilized as part of the verification effort. Instead, stubs and method specifications are annotated with preconditions ensuring that the method may not throw exceptions. As a consequence, it can be ensured that if all preconditions of method specifications are upheld, then the system will not terminate due to an exception. This is not to say that exceptions are always unwanted, as exceptions can be a powerful tool in the arsenal of programmers, but the verification effort in this work will focus on eliminating *silent* errors. Silent errors are errors that do not leave any obvious trace. This is in contrast to *noisy* errors, such as exceptions or errors, which are designed to leave informative traces as to what caused them. If the EVA system is used live to calculate the result of an election an exception or error would be unfortunate, but it would alert officials of some flaw in the system or user. It could be far more devastating if the system terminated successfully, while providing the wrong result. It should also be noted that none of the methods specified during the verification effort were implemented to throw exceptions; the Java keywords `throws`, `throw` or `try-catch` are nowhere to be found.

2. Dependency contracts for proving properties related to information flow is disregarded entirely. Information flow is specified through expressions starting with the JML keywords `accessible` and `determines by`, and can be used to prove that a program does not introduce information flows between resources in a way that is in violation of a security or privacy policy. There are no such requirements for the verification targets in this work. However, it could be beneficial to prove that the result of certain methods rely entirely on a specific set of locations. This was omitted mainly due to concerns regarding time and available computational resources, as proving dependency contracts are computationally heavy and time consuming. The reader is referred to Chapter 13 of [17] for further details on information flow analysis.

The part of the system that concerns calculating representatives at-large will be presented, and the core methods of the calculation will be specified formally; the source code will not be prepared for verification, and stubs and domain elements that are exclusive to the at-large calculation will not be presented. Still, the presented specifications of the at-large calculations displays how one could go about specifying similar methods and classes, and will create a foundation to build upon. It is believed that the verification of the at-large calculation could be completed by the tools and techniques presented and utilized during Section 4.5. That is, given enough computational resources and time to deal with the size and complexity of the classes and methods involved in the representatives at-large calculation.

4.4 Formal Specification

4.4.1 Representing the Quotients

There are two classes that are used to represent the quotients — `Kvotient` and `PartiKvotient`. The former are used for doing mathematical calculations on the quotients, while the latter are mainly used to store information. The summaries for the relevant constructors and methods in the two classes are displayed in Tables 4.11 - 4.14. Getter- and setter methods are only mentioned if their functionality is not immediately apparent from the name, or if they are of special importance.

Constructor	**Description**
`Kvotient(int k)`	Creates a quotient that has the same value as `k`.
`Kvotient(BigDecimal kvotient)`	Creates a quotient that is equal to `BigDecimal kvotient`.
`Kvotient(int teller, int nevner)`	Creates a quotient from a numerator (`teller`) and a denominator (`nevner`)

Table 4.11: Constructor summary for class `Kvotient`.

Constructor	**Description**
`PartiKvotient(String partikode, Kvotient kvotient, BigDecimal delingstall)`	Creates a quotient for a political party. Inputs are, in order of appearance, the political party's "party-code" (i.e., the political party's identifier), the calculated quotient, and the divisor used to calculate the quotient (e.g. 1.4, 3, 7, ...).

Table 4.12: Constructor summary for class `PartiKvotient`.

Modifier and Type	Method and Description
`BigDecimal`	`getKvotient()` — Returns the quotient of the object.
`Kvotient`	`add(Kvotient k)` — Returns a new `Kvotient` whose value is `(this + k)`.
`Kvotient`	`subtract(Kvotient k)` — Returns a new `Kvotient` whose value is `(this - k)`.
`Kvotient`	`multiply(BigDecimal bd)` — Returns a new `Kvotient` whose value is `(this * bd)`.
`Kvotient`	`divide(BigDecimal bd)` — Returns a new `Kvotient` whose value is `(this * bd)`.

Table 4.13: Method summary for class `Kvotient`.

Modifier and Type	Method and Description
`int`	`compareTo(PartiKvotient pk)` — Compares one party-quotient to another, by calculating the difference `(this - pk)`. If the difference is positive, then the method returns -1. If the difference is negative, it returns 1. If the difference is 0, then the difference is recalculated to `(pk.getStemmetall ()-this.getStemmetall())`. Note that this logic is opposite of the expected behavior of a class implementing `Comparable <T>`. This is done to have `Arrays.sort(Object[] a)` sort in descending order.
`Integer`	`getStemmetall()` — Returns the number of votes received by the electoral list.

Table 4.14: Method summary for class `PartiKvotient`.

The specification for class `Kvotient` in Listings 4.4-4.6 bases the correctness of the class predominately on the correctness of methods in `BigDecimal`, in that the specification utilizes the abstraction provided by the specification of `BigDecimal`. Concretely, the arithmetic methods of `Kvotient` are correct if they produce the same result as their equivalent methods in `BigDecimal`.

The following specification of class `Kvotient` displays how one could use ghost fields in specification. Specifically, the ghost field **`rValue`** residing in class `BigDecimal` has been utilized. The advantage here is that is makes the specifications more concise and readable, the implementation of the original class can be ignored, and it helps reduce the size of the proof tree during verification. The disadvantage is that the specification relies on elements that do not necessarily exists in the class that the stub represents. This disadvantage is removed if using model fields instead of ghost fields, but then the implementation of the original class cannot necessarily be ignored as the model fields then have to be represent actual fields.

The type of the ghost field is `bigint` as opposed to the ideal type `real` — the JML-type `\real` is not (yet) supported by KeY [17]. During the verification effort in Section 4.5 a discussion will be held surrounding the correctness of the model resulting from the usage of the ghost field; for now, one can intuitively think of `rValue` as a variable holding the *mathematical* number that the `BigDecimal`-object represents.

When using contracts as the primary means of verification, the verification is modular in the sense that a proof remains valid even if the program is extended conservatively. As such, a contract in a non-final class is not valid as the implicit pre-condition `requires \invariant_for(this)` is not guaranteed to hold if the class is extended. Therefore, by declaring `\inv` to only base its value on `this .*`, the specification does not consider sub-classes. As class `Kvotient` is never extended, this has no effect in this case; however, if the class is to be extended, the specification is not necessarily valid for the sub-classes. Further, the KeY-JML[10] keyword `invariant_for(o)` is used extensively. In KeY, invariants are not explicitly added to the specifications; the specifier must make explicit which invariants are included, and which are not. The expression `invariant_for(o)` refers to the invariant of object `o`. By the clause `requires \invariant_for(o)`, a method can require that the invariant of a given object `o` holds before being passed as a parameter or utilized in the method. Similarly, the clause `ensures \invariant_for(o)` specifies that the invariant of object `o` holds at the termination of the method. Requiring and ensuring that an invariant holds might seem redundant, as invariants are supposed to always hold. However, methods are allowed to temporarily break invariants as long as they are established before method termination. This can cause problems if invariants are broken for objects that are passed as parameters, or if other methods utilizes the same objects. If a method does *not* have to ensure invariants, it can be annotated as a `helper` method similarly to how `pure` methods are annotated.

The elements in the specification for class `Kvotient` contains the most fundamental JML elements. Most are presented in Section 2.3, but they will be briefly presented once again in Table 4.15 due to their importance.

JML Keyword	**Description**
spec_public	The field can be accesses directly in JML expressions, but does not override the visibility of the field for other Java expressions.

[10]In KeY's JML, all objects for which the invariants hold must be stated explicitly using the operator **\invariant_for** [17]. This differs from standard JML [50], and as such the language is referred to as KeY-JML.

normal_behavior	The following method contract is a behavioral contract that has to terminate, and may not throw exceptions.
requires	Represents a precondition. The keyword is followed by an expression that has to be true for the postconditions to be guaranteed. To specify explicitly that a method contract does not have any preconditions, the clause `requires true` may be used. If the method contract does not have any `requires` clauses, then the clause `requires true` is implicit.
ensures	Represents a postcondition. The keyword is followed by an expression; if the preconditions are true prior to method execution, then the expression is guaranteed to be true after the method terminates.
\fresh(o)	The object denoted by o is fresh, i.e., it did not exists prior to method execution.
\result	Refers to the result of the method.
assignable	Specifies the fields that the method may modify. The keyword is followed by a field, multiple fields separated by a comma, or one of the JML keywords `nothing` or `strictly_nothing`. The JML keyword `nothing` specifies that the method *may* create new objects, but not modify any existing objects. The JML keyword `strictly_nothing` specified that the method *may not* create new objects, nor may it it modify any existing objects.
\old(o)	The expression refers to the object o prior to method execution. For instance, `ensures i == \old(i) + 1` is used to specify that the value of an integer `i` is incremented by the method.

Table 4.15: Fundamental JML elements.

```
/*@ spec_public @*/ private BigDecimal kvotient;

//@ public accessible \inv: this.*;

// Constructors

/*@ public normal_behavior
```

```
  @ requires true;
  @ ensures kvotient.rValue == n;
  @ ensures \fresh(kvotient);
  @ assignable kvotient;
  @*/
public Kvotient(int n);

/*@ public normal_behavior
  @ requires \invariant_for(bd);
  @ ensures kvotient == bd;
  @ ensures \invariant_for(bd);
  @ ensures \invariant_for(kvotient);
  @ assignable kvotient;
  @*/
public Kvotient(BigDecimal bd);

/*@ public normal_behavior
  @ requires nevner != 0;
  @ ensures \fresh(kvotient);
  @ ensures kvotient.rValue == teller / nevner;
  @ ensures \invariant_for(kvotient);
  @ assignable kvotient;
  @*/
public Kvotient(int teller, int nevner);
```

Listing 4.4: JML specification for constructors of class `Kvotient`.

```
/*@ public normal_behavior
  @ requires \invariant_for(k);
  @ ensures \fresh(\result);
  @ ensures
  @  \result.kvotient.rValue ==
  @  (this.kvotient.add(k.kvotient)).rValue;
  @ ensures \invariant_for(k);
  @ ensures \invariant_for(kvotient);
  @ assignable \nothing;
  @*/
public Kvotient add(Kvotient k);

/*@ public normal_behavior
  @ requires \invariant_for(k);
  @ ensures \invariant_for(k);
  @ ensures \invariant_for(k.kvotient);
  @ ensures \invariant_for(kvotient);
  @ ensures \fresh(\result);
  @ ensures
  @  \result.kvotient.rValue ==
  @  (this.kvotient.subtract(k.kvotient)).rValue;
  @ assignable \nothing;
  @*/
```

```
public Kvotient subtract(Kvotient k);

/*@ public normal_behavior
  @ requires \invariant_for(bd);
  @ requires bd.rValue != 0;
  @ ensures \fresh(\result);
  @ ensures
  @  \result.kvotient.rValue ==
  @  (this.kvotient.divide(bd, 20, 4)).rValue;
  @ ensures \invariant_for(bd);
  @ ensures \invariant_for(kvotient);
  @ ensures \invariant_for(\result.kvotient);
  @ assignable \nothing;
  @*/
public Kvotient divide(BigDecimal bd);

/*@ public normal_behavior
  @ requires \invariant_for(bd);
  @ ensures \fresh(\result);
  @ ensures
  @  \result.kvotient.rValue ==
  @  (this.kvotient.multiply(bd)).rValue;
  @ ensures \invariant_for(bd);
  @ ensures \invariant_for(kvotient);
  @ assignable \nothing;
  @*/
public Kvotient multiply(BigDecimal bd);
```

Listing 4.5: JML specification for arithmetic methods of class `Kvotient`.

```
/*@ public normal_behavior
  @ requires true;
  @ ensures \result == kvotient;
  @ assignable \strictly_nothing;
  @*/
public BigDecimal getKvotient();

public /*@ pure @*/ String toString();
```

Listing 4.6: JML specification for helper methods of class `Kvotient`.

The specifications for class `PartiKvotient` follows. The class-level specification in Listing 4.7 introduces a new specification artifact, namely the *instance invariant*. An instance invariant contains an expression that *must* be true in all states in which a method is called or terminates [17]. The instance invariant in Listing 4.7 states that the number of votes that a party-quotient is based on cannot be negative.

In the original source-code, the method `PartiKvotient.compareTo` utilizes the constant `BigDecimal.ZERO` to check if an object of `BigDecimal` was greater than 0; a proposed model that does not utilize this constant will be presented in Section 4.5.1. Further, the type of field `stemmetall` was changed to the primitive `int`. These changes will be discussed further in Section 4.5.1.

Displayed in Listing 4.8 is the method contract for the method that `Array.sort` in Listing 4.35 utilizes to sort the quotients. For the quotients to be sorted in descending order, it is vital that the method returns a negative number if `this` is larger than `partiKvotient` and returns a positive number for the converse. The method contract assumes that `BigDecimal.compareTo(BigDecimal val)` is correct according to its specification [12].

```
import java.math.BigDecimal;
import Kvotient;

public class PartiKvotient{

  /*@ spec_public @*/
  private final static BigDecimal ZERO = new BigDecimal(0);

  /*@ spec_public @*/ private final BigDecimal delingstall;
  /*@ spec_public @*/ private final String partikode;
  /*@ spec_public @*/ private final Kvotient kvotient;
  /*@ spec_public @*/ private int stemmetall;

  //@ public accessible \inv: this.*;
  //@ public instance invariant stemmetall >= 0;

  /*@ public normal_behaviour
    @ ensures this.delingstall == delingstall;
    @ ensures this.partikode == partikode;
    @ ensures this.kvotient == kvotient;
    @ ensures
    @  \old(\invariant_for(delingstall))
    @   ==> \invariant_for(delingstall);
    @ ensures
    @  \old(\invariant_for(partikode))
    @   ==> \invariant_for(partikode);
    @ ensures
    @  \old(\invariant_for(kvotient))
    @   ==> \invariant_for(kvotient);
    @ assignable delingstall, partikode, kvotient;
    @*/
    public PartiKvotient(BigDecimal delingstall, String
   partikode, Kvotient kvotient);
```

Listing 4.7: JML specification for constructor of class `PartiKvotient`.

```
/*@ public normal_behavior
  @ requires \invariant_for(partiKvotient);
  @ requires \invariant_for(partiKvotient.getKvotient());
  @ requires \invariant_for(this.kvotient);
  @ ensures \invariant_for(partiKvotient);
  @ ensures \invariant_for(partiKvotient.getKvotient());
  @ ensures \invariant_for(this.kvotient);
  @ ensures
  @  getKvotient().getKvotient().compareTo(
  @    partiKvotient.getKvotient().getKvotient()) < 0
  @  ==> \result == 1;
  @
  @ ensures
  @  getKvotient().getKvotient().compareTo(
  @    partiKvotient.getKvotient().getKvotient()) > 0
  @  ==> \result == -1;
  @
  @ ensures
  @  getKvotient().getKvotient().compareTo(
  @    partiKvotient.getKvotient().getKvotient()) == 0
  @  ==> \result == partiKvotient.getStemmetall() -
  @                           this.getStemmetall();
  @
  @ assignable \nothing;
  @*/
public int compareTo(PartiKvotient partiKvotient);

/*@ public normal_behaviour
  @ requires \invariant_for(partiKvotient);
  @ requires \invariant_for(partiKvotient.getKvotient());
  @ ensures \result ==
  @          partiKvotient.getStemmetall() - stemmetall;
  @ ensures \invariant_for(partiKvotient);
  @ ensures \invariant_for(partiKvotient.getKvotient());
  @ assignable \strictly_nothing;
  @*/
private int sammenlignHvisKvotienterErLike(PartiKvotient
    partiKvotient);
```

Listing 4.8: JML specification for `compareTo()`-method of class `PartiKvotient`.

```
// Getters

/*@ public normal_behaviour
  @ requires true;
  @ ensures \result == delingstall;
  @ assignable \strictly_nothing;
  @*/
public BigDecimal getDelingstall();

/*@ public normal_behavior
  @ requires true;
  @ ensures \result == kvotient;
  @ assignable \strictly_nothing;
  @*/
public Kvotient getKvotient();

/*@ public normal_behaviour
  @ requires true;
  @ ensures \result == partikode;
  @ assignable \strictly_nothing;
  @*/
public String getPartikode();

/*@ public normal_behavior
  @ requires true;
  @ ensures \result == stemmetall;
  @ assignable \strictly_nothing;
  @*/
public /*@ nullable @*/ int getStemmetall();


// Setters
/*@ public normal_behavior
  @ requires stemmetall >= 0;
  @ ensures this.stemmetall == stemmetall;
  @ assignable this.stemmetall;
  @*/
public void setStemmetall(int stemmetall);
```

Listing 4.9: JML specification for helper methods of class `PartiKvotient`.

4.4.2 Representing the Representatives

The representatives themselves are represented by class `Mandater`, which is present in package `no.valg.valgnatt.database`. The reader is advised to familiarize themselves with Figure 4.1 for a better understanding on the separated architecture of the seat allocation. In short, all seat allocating calculations are done in package `no.valg.valgnatt.backend.mandatberegning`, while the results of said calculations are stored in instances of classes in package `no.valgnatt.database`. For instance, in method `beregnOgOppdater(...)`[11], the calculated result is stored as `Mandater`-objects that represents representatives.

It should be noted that classes `MandatData` and `Mandater` only concerns how many representatives each electoral list shall have, and not who should be representatives. The electoral lists themselves decide which candidates should become representatives. Candidates are represented by class `Kandidat` in package `no.valg.valgnatt.database.hibernate`, and each electoral list has a map of all its candidates in class `PartiMandater`. The map with electoral lists and votes casts that the seat allocation is based on, is retrieved from class `Valgdistrikt` . The class represents a constituency, and the returned map is generated by

[11]Method displayed in Listing 4.14 in Section 4.4.4.

iterating over all instances of `PartiMandater` in the constituency.
Classes `Mandater`, `MandatData` and `PartiMandater` have all been specified and verified to conform to their specifications, but are not presented in this document; the specifications and proofs can be found in the Git repository *EVA-KeY* [47]. Class `Valgdistrikt`, however, is presented in Listing 4.10.
The specification of class `Valgdistrikt` contains the first occurrences of quantified expressions. The quantified expressions `forall` and `exists` have the same syntax, and represent the first-order quantifies ∀ and ∃, respectively. Both expressions have the same syntax:

$$(\mathcal{Q}\, T\, v\ ;\ \textsc{range}(v)\ ;\ \textsc{exp}(v)) \tag{2}$$

where $\mathcal{Q}$ is either `\forall` or `\exists`, v is the quantified variable, T is the type of v (e.g., int, Object), $\textsc{range}(v)$ is the range of the variable (e.g., $0 \leq v \wedge v \leq 10$), and $\textsc{exp}(v)$ is an expression containing v. Intuitively, expressions starting with `\forall` are true if $\textsc{exp}(v)$ is true for *all* v in range, and expressions starting with `\exists` are true if $\textsc{exp}(v)$ is true for *some* v in range. The former is vacuously valid, while the latter is vacuously unsatisfiable.

```
package no.valg.valgnatt.database.hibernate;

import no.valg.valgnatt.database.geografi.MandatData;
import no.valg.valgnatt.database.geografi.PartiMandater;

import java.util.Map;
import java.util.TreeMap;

public class Valgdistrikt {

  private final transient Map<String, PartiMandater>
   partiMap = new TreeMap();
  private final static String BLANKE = "BLANKE";

  /*@ public normal_behavior
    @ requires \invariant_for(pmArr);
    @ requires pmArr.length > 0;
    @ requires
    @  (\forall int i; 0 <= i &&
    @                  i < getPartiMandater().length;
    @   \invariant_for(getPartiMandater()[i]) &&
    @   \invariant_for(getPartiMandater()[i]
    @                  .resultatData) &&
    @   \invariant_for(getPartiMandater()[i]
    @                  .prognoseData));
    @
    @ ensures
    @  fjernBlanke ==> !(\result.containsKey(BLANKE));
    @
    @ ensures fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \result.size() &&
    @   !(getPartiMandater()[i].partikode.equals(BLANKE));
    @   \result
    @    .containsKey(getPartiMandater()[i].partikode));
    @
    @ ensures !fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \result.size();
    @    \result.containsKey(getPartiMandater()[i]
    @                        .partikode)
    @    &&
    @     ((getPartiMandater()[i].isStillerliste() &&
    @      borBenytteListestemmer &&
    @      listestemmerFinnes(getPartiMandater()[i]
    @                         .resultatData))
    @     ==>
    @     (\result.get(getPartiMandater()[i].partikode) ==
    @      getPartiMandater()[i]
    @      .resultatData.getListestemmer()))
    @    &&
    @     (!(getPartiMandater()[i].isStillerliste() &&
    @        borBenytteListestemmer &&
    @        listestemmerFinnes(getPartiMandater()[i]
    @                           .resultatData))
    @     ==>
    @     (\result.get(getPartiMandater()[i].partikode) ==
    @      getPartiMandater()[i]
    @      .resultatData.getStemmer())));
    @
    @ assignable \nothing;
    @*/
    public Map<String, Integer>
   getStemmerPrPartiTilMandatberegning(boolean
   borBenytteListestemmer, boolean fjernBlanke, boolean
   prognose);

  /*@ public normal_behavior
    @ requires \invariant_for(partiMap);
    @ ensures \result == partiMap.values();
    @ ensures \invariant_for(partiMap);
    @ assignable \strictly_nothing;
    @*/
  public PartiMandater[] getPartiMandater();

  /*@ normal_behavior
    @ requires \invariant_for(md);
    @ ensures \result == md.getListestemmer() > 0;
    @ ensures \invariant_for(md);
    @ assignable \strictly_nothing;
    @*/
  private boolean listestemmerFinnes(MandatData md);

  /*@ public normal_behavior
    @ requires \invariant_for(partiMap);
    @ requires partiMap.containsKey(partikode);
    @ ensures \result == partiMap.get(partikode);
    @ assignable \strictly_nothing;
    @
    @ also
    @
    @ public normal_behavior
    @ requires \invariant_for(partiMap);
    @ requires !(partiMap.containsKey(partikode));
    @ ensures \result == null;
    @ assignable \strictly_nothing;
    @*/
  public PartiMandater getPartiMandater(String partikode);
}
```

Listing 4.10: Signatures and specifications for relevant methods and fields of class `Valgdistrikt`.

The method `getStemmerPrPartiTilMandatberegning` is the most interesting one, as this is the method that is called to retrieve maps for calculating the seat allocation. The parameter `fjernBlanke` specifies whether or not the electoral list with the party-code specified by field `BLANKE` should be part of further calculations. Parties with the party-code "BLANKE" represent protest parties; a protest party is a party that receives the protest votes cast during the election. A protest vote (also called a blank or "none of the above" vote) is a vote cast in an election to demonstrate dissatisfaction with the choice of candidates or the current political system [67]. The postconditions to the methods are as follows:

1. If the parameter `fjernBlanke` is true, then the resulting map will not contain the electoral list for the protest party.
2. In addition, if the parameter `fjernBlanke` is true, then the resulting map will contain all electoral lists in the constituency except those for the protest party.
3. If the parameter `fjernBlanke` is false, then the resulting map will contain all electoral lists, even for the protest party. Further, the type of votes for the electoral list will be the personal votes if they are requested and they exists; otherwise the type of votes will be regular. The specification lacks a similar postcondition for the case where `fjernBlanke` is true, as this is considered trivial.

4.4.3 Calculating the Sorted Quotients

The calculation of the sorted quotients is one of the most central aspects of Sainte-Laguës's modified method. As was briefly mentioned in Section 4.3.1, the logic realizing the calculation is implemented in the method `sorterteKvotienter (...)` in class `MandatberegningFelles`. The JML specification for the method can be found in Listing 4.11. The preconditions for the method are

1. all invariants for the supplied map must hold initially,
2. the supplied map must be initialized,
3. the invariants for the map's key-set and the invariants for all keys must hold initially,
4. all values are non-null and larger than 0,
5. the map has at least one element, and
6. the number of representatives to assign is greater than 0.

If the preconditions are met, the method will ensure that

1. neither the returning array nor any of its elements are `null`,
2. the length of the result, i.e., the number of returned quotients, are decided by the parameter `antReturKvotienter`; if the parameter is larger than the parameter `antallMandater` then all representatives calculated are returned, otherwise they are limited by `antReturKvotienter`,
3. for every `PartiKvotient` in the returning array
   (a) the object's `partikode` is a key in the map,
   (b) the object's `stemmetall` is the value of that key,
   (c) the object's `delingstall` is a Sainte-Laguë divisor, and
   (d) the value of the object's `kvotient` is calculated by taking dividing `stemmetall` with that divisor.
4. every key/value-pair in the map is present in the result. When combining this postcondition with the prior postcondition it is guaranteed that the result consists *only* of `PartiKvotient`-objects that are created based on a key/value-pair of the map, and `PartiKvotient`-objects are created for *all* the map's key/value-pairs,
5. all elements in the result are sorted according to `PartiKvotient.compareTo`,
6. the invariants hold for the map after termination.

```
/*@ public normal_behavior
  @ requires \invariant_for(partiStemmeMap);
  @ requires partiStemmeMap != null;
  @
  @ requires
  @  (\forall int i; 0 <= i && i < partiStemmeMap.size();
  @    partiStemmeMap.keySet() != null &&
  @    \invariant_for(partiStemmeMap.keySet()) &&
  @    \invariant_for(partiStemmeMap.keySet()[i]));
  @
  @ requires
  @  (\forall int i; 0 <= i && i < partiStemmeMap.size();
  @    partiStemmeMap.values() != null &&
  @    partiStemmeMap.values()[i] >= 0);
  @
  @ requires partiStemmeMap.size() > 0;
  @ requires antallMandater > 0;
  @
  @ ensures
  @   ((antallMandater < antReturKvotienter &&
  @     antReturKvotienter > 0 &&
  @     antReturKvotienter > partiStemmeMap.size() *
  @                          antallMandater)
  @    ==> \result.length == antReturKvotienter)
  @   ||
  @   \result.length == partiStemmeMap.size() *
  @                     antallMandater;
  @
  @ ensures (\forall int i; 0 <= i && i < \result.length;
  @  partiStemmeMap.containsKey(\result[i].getPartikode())
  @  && partiStemmeMap.get(\result[i].getPartikode()) ==
  @    \result[i].getStemmetall() &&
  @  (\exists int n; 0 <= n && n <= antallMandater;
  @     \result[i].getDelingstall()
  @               .compareTo(getSainteLagueDelingstall(n))
  @               == 0 &&
  @     \result[i].kvotient.kvotient.rValue ==
  @       \result[i].getStemmetall() /
  @       getSainteLagueDelingstall(n).rValue));
  @
  @ ensures
  @  (\forall int i; 0 <= i && i < partiStemmeMap.size();
  @    (\exists int j; 0 <= j && j < \result.length;
  @     partiStemmeMap.keySet()[i]
  @       .equals(\result[j].getPartikode()) &&
  @     partiStemmeMap.get(partiStemmeMap.keySet()[i]) ==
  @       \result[j].getStemmetall()));
  @
  @ ensures
  @  (\forall int i; 0 <= i && i < \result.length;
  @    (\forall int j; 0 <= j && j < i;
  @      \result[i].compareTo(\result[j]) >= 0));
  @
  @
  @ ensures \invariant_for(partiStemmeMap);
  @
  @ ensures \result != null;
  @ assignable \nothing;
  @*/
static PartiKvotient[] sorterteKvotienter(int
    antallMandater, Map<String, Integer> partiStemmeMap,
    Integer antReturKvotienter) {/*...*/}
```

Listing 4.11: Signature and JML specification for method responsible for calculating the sorted quotients.

```
static BigDecimal getSainteLagueDelingstall(int n) {
  return n == 0 ? new BigDecimal(1.4) :
                  new BigDecimal(2 * n + 1);
}
```

Listing 4.12: Method for calculating the Sainte-Laguë divisor for the iteration specified by parameter `int n`.

In addition to calculating quotients, class `MandatberegningFelles` offers the ability to limit the number of quotients to correspond to the number of representatives to award thorough method `begrensAntallKvotienter` seen in Listing 4.13. If the parameter `antallKvotienter` exceeds the number of available quotients in `kvotienter`, the parameter `kvotienter` will simply be returned. Otherwise, in the normal use-case, the result will be a new array in which the first elements, i.e., the elements from 0 to `antallKvotienter`, will equal the first elements of `kvotienter`.

```
/*@ public normal_behaviour
  @ requires \invariant_for(kvotienter);
  @ requires antallKvotienter < kvotienter.length
  @       && antallKvotienter > 0;
  @ ensures \result.length == antallKvotienter;
  @ ensures
  @   (\dl_seqPerm(
  @      \dl_array2seq(\old(kvotienter))
  @                 [0..antallKvotienter],
  @      \dl_array2seq(\result)));
  @ assignable \nothing;
  @
  @ also
  @
  @ public normal_behavior
  @ requires \invariant_for(kvotienter);
  @ requires antallKvotienter >= kvotienter.length
  @       || antallKvotienter <= 0;
  @ ensures \result == kvotienter;
  @ assignable \strictly_nothing;
  @*/
static PartiKvotient[] begrensAntallKvotienter(
    PartiKvotient[] kvotienter, int antallKvotienter);
```

Listing 4.13: Signature and JML specification of method `begrensAntallKvotienter` from class `MandatberegningFelles`.

4.4.4 Calculating the Constituency Representatives

The system offers calculations of constituency- and representatives at-large in classes `Distriktsmandat` and `Utjevningsmandat`, respectively. As constituency representatives and representatives for local councils are elected in equal fashion, there is no logical need to distinguish them. However, representatives at-large are calculated by a separate algorithm, and therefore they are represented by their own class. Class `Distriktsmandat` will be presented first, as it is inherently less complex, and the presentation of class `Utjevningsmandat` will be the focus of Section 4.4.5.

The only public method of `Distriksmandat` is presented in Listing 4.14. The method returns an object of `DistriktmandatResultat`, which contains the result along with forecasts. As this verification effort is not concerned with forecasts, the class is can be considered to simply containing a map with number of constituency representatives awarded to any given electoral list.

The method retrieves a map from parameter `omrade`[12] by calling the method `getStemmerPrPartiTilMandatberegning` on the parameter's `Valgdistrikt`-object. The entries of the map represents the number of votes cast (value) for a given electoral list (key). Next, the method calculates the sorted quotients through the methods in Listing 4.11 and 4.13, summarizes the number of resulting quotients for each electoral list by the method in Listing 4.15, and stores the sum in `Mandater`-objects nested inside parameter `omrade`.

Due to the dependency on internal methods and the nested complexity of class `Omrade`, creating a formal specification of non-trivial properties for the method `beregnOgOppdater` was exceptionally difficult. As such, this work will focus on formalizing and verifying the methods that the method bases its result on, i.e., the methods in Listings 4.10, 4.11, 4.13, and 4.15. Even though the method itself does not have any explicitly specified properties, all methods that the method uses to calculate quotients have explicitly specified properties. This will not completely verify the method, but it will increase confidence that the method behaves as expected.

```
public static DistriksmandatResultat beregnOgOppdater(
    Omrade omrade, int antallMandater, boolean
    stortingsDistriktsmandaterIFYValg, boolean prognose);
```

Listing 4.14: The method `beregnOgOppdater` in class `Distriksmandat` calculates the number of constituency representatives awarded to each electoral list.

[12]Class `Omrade` was originally called "Område", the Norwegian word for "area".

As mentioned above, the method in Listing 4.15 below is responsible for summarizing the number of resulting quotients for each electoral list. The method returns a map where the keys represent an electoral list and the values represent the number of quotients in parameter `kvotienter` that belongs to the electoral list. The two postconditions ensure that there are no duplicate keys, and that the value for any given key is the number of quotients belonging to the key in the parameter `kvotienter`.

To summarize the number of resulting quotients for each electoral list, the specification utilizes a new JML keyword, namely `\num_of`. The keyword is an alias for a specific application of another JML keyword `\sum`. Their relationship and syntax is seen in Equation 3.

$$(\backslash\text{num\_of T x; R(x); P(x)}) == (\backslash\text{sum T x; R(x) \&\& P(x); 1L}) \tag{3}$$

Here, T is the type of variable x, R(x) is the range of x that the expression will consider, and P(x) is a predicate containing x. The expression `num_of` returns the number of values for its variables for which the range and the predicate are true. That is, the value returned by `num_of` is the amount of values for x that satisfy both the range and the predicate. For example, the expression

```
(\num_of int i; 0 <= i && i <= 10; i < 5)
```

will return 5, i.e., the amount of numbers from 0 to 10, including 0 and 10, that are less than 5. As defined by Equation 3, the expression

```
(\sum int i; 0 <= i && i <= 10 && i < 5; 1L)
```

would be equivalent. The expression adds 1 to the current sum for every value of `i` that satisfies all three conditions in the range.

```
/*@ public normal_behavior
  @ requires \invariant_for(kvotienter);
  @ requires kvotienter.length > 0;
  @
  @
  @ ensures
  @  (\forall int i; 0 <= i && i < \result.size();
  @    !(\exists int j; i < j && j < \result.size();
  @       \result.keySet()[i]
  @       .equals(\result.keySet()[j])));
  @
  @ ensures
  @  (\forall int i; 0 <= i && i < \result.size();
  @     \result.values()[i] ==
  @     (\num_of int j; 0 <= j && j < kvotienter.length &&
  @       kvotienter[j].getPartikode()
  @       .equals(\result.keySet()[i])));
  @
  @ ensures \invariant_for(kvotienter);
  @ assignable \nothing;
  @*/
private static Map<String, Integer> mandatFordeling(
    PartiKvotient[] kvotienter);
```

Listing 4.15: Method `mandatFordeling` returns a map where the keys represent an electoral list and the values represent the number of quotients in parameter `kvotienter` that belongs to the electoral list

4.4.5 Calculating the At-Large Representatives

Class `Utgjevningsmandat` is approximately four times as large as `Distriksmandat` when comparing lines of code, but the public interfaces are nearly identical. The only public method in class `Utgjevningsmandat` is the static method `beregnOgOppdater(...)` in Listing 4.16.

```
public static UtjevningsmandatResultat beregnOgOppdater(
    Land land, Map<String,Integer> distriktsmandaterPrFylke,
     boolean prognose) {/*...*/}
```

Listing 4.16: The method `beregnOgOppdater` in class `Utgjevningsmandat` calculates the number of representatives at-large awarded to each electoral list.

The method attempts to capture the at-large allocation process as described and exemplified in Section 4.2. The algorithm implemented in the method computes the at-large allocation in the following manner:

1. Create a new map. Retrieve total votes cast for all electoral lists throughout the nation. Store the result in the newly created map.
2. Create a new map where the keys are constituencies and the values are maps with number of constituency representatives allocated for each electoral list.
3. Create a similar map, except constituency representatives are substituted for number of votes cast for each electoral list.
4. Create a set with party codes for local electoral lists.
5. Prepare data for calculations; for each constituency in parameter `land`,
    (a) create a new map, populate it with party codes, and for each party code, add the number of constituency representatives the party received from the constituency. Add the constituency number and newly created map to the map in Step 2;

(b) create a new map, populate it with party codes, and for each party code, add the number of votes cast for the party in the constituency. Add the constituency number and newly created map to the map in Step 3;

(c) add the party code to the set of party codes from Step 4.

6. Calculate the seat at-large allocation.

   (a) Create a map of electoral lists above the electoral threshold[13] with the total numbers of votes cast for the corresponding electoral list.

   (b) With the map from the previous sub-step, calculate the quotients when considering the nation as one constituency. The number of quotients calculated are limited to (number of electoral lists $\times$ number of representatives to allocate).

   (c) Summarize the total number constituency representatives received for each electoral list.

   (d) For every constituency representative, remove the highest quotient from the first sub-step for the electoral list which the constituency representative was allocated.

   (e) Create a new map. For the 19 highest remaining quotients, allocate the representatives at-large to the electoral lists the quotient is for. Store the result in the newly created map.

   (f) Create a new list containing quotients for all electoral lists that received representatives at-large during the previous step. This list of quotients will decide the constituencies that will receive the representatives at-large. The formula for the quotients is

$$\frac{\frac{\text{Votes cast for electoral list in a constituency}}{(2\times\text{Constituency representatives won for electoral list})+1}}{\text{Ratio of votes per representative in the constituency}}$$

   (g) Allocate the representatives at-large, one for each constituency.

      i. Allocate a representative to the highest quotient in (f).

      ii. Allocate the next representative to the second highest quotient in (f). If the quotient belongs to a constituency that has been allocated a representative at-large, disregard the quotient and move to the highest quotient thereafter. If the quotient belongs to an electoral list that has received all their representatives at-large, disregard the quotient and move to the highest quotient thereafter.

   After this step, all representatives at-large have been allocated; one for each constituency, and a correct number of representatives for each electoral list.

[13]The electoral threshold is the minimum percentage of total votes any electoral list must have in order to receive representatives; 4% for parliamentary elections, 8% for constituency elections [52, 69].

To assist in the computation, the method utilizes several private methods. In fact, most calculations are done inside these methods. Therefore, the correctness of the public method `beregnOgOppdater(...)` is based primarily on the correctness of the private methods. As such, a similar approach will be taken as for the method in Listing 4.14 where the method will not receive any explicit specification, but all the methods that method calls will receive explicit specifications. The private and package-private method signatures of class `Utgjevningsmandat` are seen in Listings 4.17-4.30 along with their JML specifications.

The method in Listing 4.17 has to fulfill the following requirements, listed in order of specification:

1. Both of the supplied maps, `distriktsmandaterPrPartiPrFylke` and `stemmerPrPartiPrFylke`, must have all counties in country `land` in their respective key-set, and all keys must have a corresponding county. A string that does not represent a county cannot be a key, and all strings that represent a county must be a key in both maps.

2. For all parties in all counties in `distriksmandatPrPartiPrFylke`, there exists a corresponding party in the corresponding county in country `land`, and the corresponding parties have the same number of constituency representatives.

3. For all parties in all counties in `stemmerPrPartiPrFylke`, there exists a corresponding party in the corresponding county in country `land`, and the corresponding parties have the same number of votes.

4. All strings in set `lokalePartikoder` represent the code of a local party. A party is local if the party's category is 3. All items of the set must represent a party in a constituency that is in category 3, and all strings that represent such a party must be in the set.

5. The invariants for all the parameters are upheld. This is, of course, given that they hold initially as specified by the precondition.

The assignable-clause specified that the method may only alter the *footprint* of the parameters that are maps and sets. When implementing the JML-interfaces or stubs for the maps and sets, the footprint will be the sequence that the keys/values or objects are stored in. For instance, in the JML implementation of `java.util.TreeMap` in Listing 4.38 on Page 100, the footprint of the class would be the set union of the set of locations of sequences `keys` and `values`. In other words, the assignable clause of the method specification ensures that the method may only alter the objects stores in the parameters that are sets and maps, and not the pointers themselves. Neither may the method alter the parameters `land` or `prognose`, or anything else for that matter, as these are not mentioned.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(land) &&
  @  \invariant_for(distriktsmandaterPrPartiPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke) &&
  @  \invariant_for(lokalePartikoder);
  @
  @ ensures
  @  (\forall Fylke f;
  @    land.getFylker().containsValue(f);
  @    distriktsmandatPrPartiPrFylke.containsKey(f.getNr())
  @    && stemmerPrPartiPrFylke.containsKey(f.getNr()));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    distriktsmandatPrPartiPrFylke.containsKey(fylkeNr);
  @    (\exists Fylke f;
  @    land.getFylker().containsValue(f);
  @    f.getNr().equals(fylkeNr)));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    stemmerPrPartiPrFylke.containsKey(fylkeNr);
  @    (\exists Fylke f;
  @    land.getFylker().containsValue(f);
  @    f.getNr().equals(fylkeNr)));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    distriktsmandatPrPartiPrFylke.containsKey(fylkeNr);
  @    (\forall String partiKode;
  @    ((Map) distriktsmandatPrPartiPrFylke.get(fylkeNr))
  @       .containsKey(partiKode);
  @    (\exists PartiMandater pm;
  @      (\exists Fylke f;
  @      land.getFylker().containsValue(f) &&
  @          f.getNr().equals(fylkeNr);
  @      f.getValgdistrikt().getPartiMandater()
  @                           .contains(pm));
  @      pm.getPartikode().equals(partiKode) &&
  @          pm.finnBesteMandatData(prognose)
  @            .getDistriktsmandater()
  @            .getAntall()
  @            .equals(
  @              ((Map) distriktsmandatPrPartiPrFylke
  @                     .get(fylkeNr))
  @              .get(partiKode)))));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    stemmerPrPartiPrFylke.containsKey(fylkeNr);
  @    (\forall String partiKode;
  @    ((Map) stemmerPrPartiPrFylke.get(fylkeNr))
  @       .containsKey(partiKode);
  @      (\exists PartiMandater pm;
  @      (\exists Fylke f;
  @      land.getFylker().containsValue(f) &&
  @          f.getNr().equals(fylkeNr);
  @      f.getValgdistrikt().getPartiMandater()
  @                              .contains(pm));
  @      pm.getPartikode().equals(partiKode) &&
  @      pm.finnBesteMandatData(prognose)
  @            .getStemmer()
  @            .equals( ((Map) stemmerPrPartiPrFylke
  @                             .get(fylkeNr))
  @                     .get(partiKode)))));
  @
  @ ensures
  @  (\forall PartiMandater pm;
  @    (\exists Fylke f;
  @      land.getFylker()
  @          .containsValue(f);
  @      f.getValgdistrikt()
  @       .getPartiMandater()
  @       .contains(pm));
  @     (pm.getPartikategori().intValue() == 3) ==>
  @     (lokalePartikoder.contains(pm.getPartikode())));
  @
  @ ensures
  @  \invariant_for(land) &&
  @  \invariant_for(distriktsmandaterPrPartiPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke) &&
  @  \invariant_for(lokalePartikoder);
  @
  @ assignable
  @  stemmerPrPartiPrFylke.footprint,
  @  distriktsmandaterPrPartiPrFylke.footprint,
  @  lokalePartikoder.footprint;
  @*/
private static void klargjorDataTilBeregning(Land land, Map
    <String, Map<String, Integer>>
    distriktsmandaterPrPartiPrFylke, Map<String, Map<String,
     Integer>> stemmerPrPartiPrFylke, Set<String>
    lokalePartikoder, boolean prognose);
```

Listing 4.17: Method signature and JML specification of method `klargjorDataTilBeregning` in class `Utgjevningsmandat`.

The method `beregn` in Listing 4.18 below is similar to method `beregnOgOppdater` in Listing 4.14 in that the result of the method is calculated based on a variety of other methods. As such, creating a formal specification for the method would be a cumbersome task, that would be of little significance. Instead, the effort is placed on creating specifications for the methods that are used throughout the calculation. As a consequence, if the methods utilized are verified to conform to their specifications, greater confidence can be held in the method's behavior to be as expected.

```
private static UtjevningsmandatResultat beregn(Map<String,
    Integer> stemmerPrParti, Set<String> lokalePartikoder,
    Map<String, Map<String, Integer>>
    distriktsmandaterVunnetPrPartiPrFylke, Map<String, Map<
    String, Integer>> stemmerPrPartiPrFylke,  Map<String,
    Integer> distriktsmandaterTilValgPrFylke);
```

Listing 4.18: Method signature and JML specification of method `beregn` in class `Utgjevningsmandat`.

The method in Listing 4.19 removes all entries from parameter `partiStemmeMap` where the key of the entry is "BLANKE". The code "BLANKE" is used to represent protest-parties, i.e., made-up parties that electors can cast their vote for if displeased with the other options[14]. As such, the result of this method must contain all entries from the supplied map, except for the entries with key "BLANKE". Also, the supplied map should be copied over, but not altered.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(partiStemmeMap);
  @
  @ ensures
  @  !(\result.containsKey("BLANKE"));
  @
  @ ensures
  @  (\forall String s;
  @    partiStemmeMap.containsKey(s) &&
  @    !(s.equals("BLANKE"));
  @    \result.containsKey(s) &&
  @    \result.get(s).equals(partiStemmeMap.get(s)));
  @
  @ ensures
  @  \invariant_for(partiStemmeMap);
  @
  @ assignable \nothing;
  @*/
private static Map<String, Integer> fjernBlankePartiet(Map<
    String, Integer> partiStemmeMap);
```

Listing 4.19: Method signature and JML specification of method `fjernBlankePartiet` in class `Utgjevningsmandat`.

[14]Protest votes do not count towards the total number of votes cast in the election.

The method in Listing 4.20 removes all parties from map `stemmerPrParti` that do not pass the electoral threshold, as well as all local parties. This corresponds to Point 6-(a) on page 66. The second term in the `\forall`-clause starting on line 13 ensures that only parties that are in map `stemmerPrParti` and that have more than 4% of all votes tallied, are considered further in the `ensures`-clause. The remaining `ensures`-clauses are for proving retention, proving that no *new* elements have been added to the result, and proving that invariants are upheld.

```
   /*@ public normal_behaviour
 2   @ requires
     @  \invariant_for(stemmerPrParti) &&
 4   @  \invariant_for(lokalePartikoder);
     @
 6   @ ensures
     @ (\forall
 8   @    String s;
     @    \result.containsKey(s);
10   @    !(lokalePartikoder.contains(s)));
     @
12   @ ensures
     @ (\forall
14   @    String s;
     @    stemmerPrParti.containsKey(s) &&
16   @    ( ((Integer) stemmerPrParti.get(s)).intValue() * 100
     @      / (\sum String s;
18   @          stemmerPrParti.containsKey(s);
     @          ((Integer) stemmerPrParti.get(s))
20   @                                   .intValue()) ) >=  4 ;
     @    \result.containsKey(s) &&
22   @    \result.get(s).equals(stemmerPrParti.get(s)));
     @
24   @ ensures
     @ (\forall
26   @    String s;
     @    \result.containsKey(s);
28   @    stemmerPrParti.containsKey(s) &&
     @    \result.get(s).equals(stemmerPrParti.get(s)));
30   @
     @ ensures
32   @  \invariant_for(stemmerPrParti) &&
     @  \invariant_for(lokalePartikoder);
34   @
     @ assignable \nothing;
36   @*/
   private static Map<String, Integer>
       beregnStemmerPrPartiOverSperregrensen(Map<String,
       Integer> stemmerPrParti, Set<String> lokalePartikoder);
```

Listing 4.20: Method signature and JML specification of method `beregnStemmerPrPartiOverSperregrensen` in class `Utgjevningsmandat`.

As the name suggests, the method in Listing 4.21 removes a given number of quotients from parameter `landsKvotienter` for the electoral list in `seatsParty`. Parameter `landsKvotienter` was calculated by method `sorterteKvotienter(...)` in Listing 4.11 to complete Point 6-(b) on page 66. The number of quotients removed are given from the value of `seatsParty`, while the electoral list that should have quotients removes are given in the key of `seatsParty`. This method is used for every electoral list to complete Point 6-(d) on page 66. The JML specification cases specifies, in order of the `ensure`-clauses, the following:

1. The number of quotients to be removed are given by the value of parameter `seatsParty`, and that number of quotients are removed.

2. If a quotient is removed, then there are no remaining quotients for the electoral list higher than the one removed.

3. All quotients for other electoral lists than the one specified by the key of parameter `seatsParty` remain.

4. Invariants for the parameters are upheld.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(seatsParty) &&
  @  \invariant_for(landsKvotienter);
  @
  @ ensures
  @ \result.length == (landsKvotienter.length -
  @                  ((Integer) seatsParty.getValue())
  @                                          .intValue());
  @
  @ ensures
  @ (\forall
  @    int i;
  @    i <= 0 && i < landsKvotienter.length;
  @    !(\exists
  @        int u;
  @        u <= 0 && u < \result.length;
  @        \result[u] == landsKvotienter[i])
  @        &&
  @        landsKvotienter[i]
  @         .getPartikode()
  @         .equals(seatsParty.getKey())
  @    ==>
  @    (\forall
  @        int y;
  @        y <= 0 && y < \result.length;
  @        landsKvotienter[i]
  @         .getKvotient()
  @         .compareTo(result[y].getKvotient()) == 1));
  @
  @ ensures
  @ (\forall
  @     int i;
  @     i <= 0 && i < landsKvotienter.length;
  @     !(landsKvotienter[i].getPartikode()
  @                          .equals(seatsParty.getKey()))
  @     ==>
  @     (\exists
  @         int u;
  @         u <= 0 && u < \result.length;
  @         \result[u] == landsKvotienter[i]));
  @
  @ ensures
  @  \invariant_for(seatsParty) &&
  @  \invariant_for(landsKvotienter);
  @
  @ assignable \nothing;
  @*/
private static PartiKvotient[]
    fjernHoyesteKvotienterTilsvarendeAntallDistriktsmandater
    (Entry<String, Integer> seatsParty, PartiKvotient[]
    landsKvotienter);
```

Listing 4.21: Method signature and JML specification of method `fjernHoyesteKvotienterTilsvarendeAntallDistriktsmandater` in class `Utgjevningsmandat`.

The method in Listing 4.22 is similar to some of the other methods, in that the correctness of the method is based on other methods. Specifically, the method in Listing 4.22 bases its correctness on the methods in Listings 4.23 and 4.24. Therefore, this thesis limits itself to creating specifications for the auxiliary methods in those listings.

```
private static Map<String, String>
    fordelUtjevningsmandatenePrFylkeOgParti(Map<String,
    Integer> antallUtjevningmandaterPrParti, List<
    PartiFylkeKvotient> partiFylkeKvotienter, Map<String,
    Map<String, Integer>> mandatnrPrPartiPrFylke);
```

Listing 4.22: Method signature and JML specification of method `fordelUtjevningsmandatenePrFylkeOgParti` in class `Utgjevningsmandat`.

The parameters to the method in Listing 4.23 are

1. a quotient that may decide a representative at-large,
2. a set of counties that have received all their representatives at-large, and

3. a set of electoral lists that have received all the representatives at-large they are entitled.

The method determines whether or not the county and electoral list behind quotient `partiFylkeKvotient` have received all their entitled representatives at-large. If so, returns false; else, returns true. This method is used once as part of method `fordelUtjevningsmandatenePrFylkeOgParti` in Listing 4.22. As the requirements for method `kanFordeles` is not rooted in electoral law, or elsewhere, the specification is based on what method `fordelUtjevningsmandatenePrFylkeOgParti` requires of method `kanFordeles` in order to fulfill its own requirements. As mentioned previously, the requirements of method `fordelUtjevningsmandatenePrFylkeOgParti` *is* rooted in electoral law.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(partiFylkeKvotient) &&
  @  \invariant_for(fjernedeFylker) &&
  @  \invariant_for(fjernedePartier);
  @
  @ ensures
  @  (\exists String s;
  @    fjernedeFylker.contains(s);
  @    s.equals(partiFylkeKvotient.getFylkeNr()))
  @  ==> \result == false;
  @
  @ ensures
  @  (\exists String s;
  @    fjernedePartier.contains(s);
  @    s.equals(partiFylkeKvotient.getPartikode()))
  @  ==> \result == false;
  @
  @ ensures
  @  !(\exists String s;
  @     fjernedeFylker.contains(s);
  @     s.equals(partiFylkeKvotient.getFylkeNr())) &&
  @  !(\exists String s;
  @     fjernedePartier.contains(s);
  @     s.equals(partiFylkeKvotient.getPartikode()))
  @  ==> \result == true;
  @
  @ ensures
  @  \invariant_for(partiFylkeKvotient) &&
  @  \invariant_for(fjernedeFylker) &&
  @  \invariant_for(fjernedePartier);
  @
  @ assignable \strictly_nothing;
  @*/
private static boolean kanFordeles(PartiFylkeKvotient
    partiFylkeKvotient, Set<String> fjernedeFylker, Set<
    String> fjernedePartier);
```

Listing 4.23: Method signature and JML specification of method `kanFordeles` in class `Utgjevningsmandat`.

The JML specification of the method in Listing 4.23 is divided into four `ensures` -clauses:

1. If the quotient belongs to a county that has received a representative at-large, then return false.
2. If the quotient belongs to an electoral list that has received all their entitled representatives at-large, then return false.
3. If the quotient does not belong to a county that has received a representative at-large, and the quotient does not belong to an electoral list that has received all their entitled representatives at-large, then return true.
4. The invariants of the parameters are upheld.

The third clause is a stronger default-clause. A weaker default would be to state that if neither the second nor the third clause is applicable, then true is returned. Instead, the third clause states that the result of the method must be true if the quotient does not belong to a county or an electoral list that have been allocated all their entitled representatives at-large.

The method `oppdaterFjernedePartier` in Listing 4.24 ensures that the electoral list specified by parameter `partikode` is added to the set `fjernedePartier` if the electoral list has received all its entitled representatives at-large. The specification is divided into separate `ensure`-clauses. The clauses respectively specify

1. the parameter `fjernedePartier` may not be altered if the electoral list specified by parameter `partikode` has not received all its entitled representatives at-large,
2. if the electoral list has received all its representatives at-large, then it must be added to parameter `fjernedePartier`; there are no requirements that `partikode` cannot be in set `fjernedePartier` prior to execution, and
3. the invariants of the parameters are upheld.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(utjevningmandaterPrParti) &&
  @  \invariant_for(partiPrFylke) &&
  @  \invariant_for(fjernedePartier) &&
  @  \invariant_for(partikode);
  @
  @ ensures
  @  !(utjevningmandaterPrParti.containsKey(partikode)) ||
  @  (((Integer) utjevningmandaterPrParti.get(partikode))
  @   .intValue() != (\num_of
  @                     String s;
  @                     partiPrFylke.containsValue(s);
  @                     s.equals(partikode)))
  @  ==>
  @  fjernedePartier == \old(fjernedePartier);
  @
  @ ensures
  @  utjevningmandaterPrParti.containsKey(partikode) ||
  @  (((Integer) utjevningmandaterPrParti.get(partikode))
  @   .intValue() == (\num_of
  @                     String s;
  @                     partiPrFylke.containsValue(s);
  @                     s.equals(partikode)))
  @  ==>
  @  fjernedePartier.contains(partikode);
  @
  @ ensures
  @  \invariant_for(utjevningmandaterPrParti) &&
  @  \invariant_for(partiPrFylke) &&
  @  \invariant_for(fjernedePartier) &&
  @  \invariant_for(partikode);
  @
  @ assignable
  @  fjernedePartier.footprint;
  @*/
private static void oppdaterFjernedePartier(Map<String,
    Integer> utjevningmandaterPrParti, Map<String, String>
    partiPrFylke, Set<String> fjernedePartier, String
    partikode);
```

Listing 4.24: Method signature and JML specification of method `oppdaterFjernedePartier` in class `Utgjevningsmandat`.

The method in Listing 4.25 creates a new map which it populates based on the first $\mathcal{N}$ = (`antallUtjevningsmandater`) quotients in parameter `landsKvotienter`. Each electoral list will be awarded the same number of representatives at-large as they have quotients in the first $\mathcal{N}$ elements of `landsKvotienter`. This corresponds to Point 6-(e) on page 66.

For the specification case for method in Listing 4.25 there is only need for two `ensure`-clauses; the first clause states that

1. the result contains the codes of all electoral lists that any of the first $\mathcal{N}$ quotients belongs to,

2. the total number of representatives at-large calculated equals parameter `antallUtjevningsmandater`, and

3. each electoral list is awarded the same number of representatives at-large as the number of quotients they have in the first $\mathcal{N}$ quotients of `landsKvotienter`,

while the second clause states that the invariants of the parameters are upheld.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(antallUtjevningsmandater) &&
  @  \invariant_for(landsKvotienter);
  @
  @ ensures
  @ (\forall
  @    int i;
  @    i <= 0 && i < antallUtjevningsmandater.intValue();
  @    \result.containsKey(landsKvotienter[i]
  @                         .getPartikode())
  @    &&
  @    (\sum String s;
  @          \result.containsKey(s);
  @          ((Integer) \result.get(s)).intValue())
  @    ==
  @    antallUtjevningsmandater.intValue()
  @    &&
  @    ((Integer) \result.get(landsKvotienter[i]
  @                            .getPartikode()))
  @    .intValue() == (\num_of
  @                      String s;
  @                      true;
  @                      landsKvotienter[i]
  @                      .getPartikode().equals(s)));
  @
  @ ensures
  @  \invariant_for(antallUtjevningsmandater) &&
  @  \invariant_for(landsKvotienter);
  @
  @ assignable \nothing;
  @*/
private static Map<String, Integer>
    beregnAntallUtjevningmandaterPrParti(Integer
    antallUtjevningsmandater, PartiKvotient[]
    landsKvotienter);
```

Listing 4.25: Method signature and JML specification of method `beregnAntallUtjevningmandaterPrParti` in class `Utgjevningsmandat`.

Method `getKvot` displayed in Listing 4.26 implements the formula in Point 6-(f) on page 66. The first `ensures`-clause checks the case where the ratio has not been calculated, or if the ratio is zero. The second clause ensures that the returned quotient equals the formula when calculated using the class `BigDecimal`. The third clause ensures that the invariants of the parameters are upheld.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(stemmerBakEtMandatPrFylke) &&
  @  \invariant_for(fylkeNr) &&
  @  \invariant_for(stemmer);
  @
  @ ensures
  @  stemmerBakEtMandatPrFylke.get(fylkeNr) == null
  @  ==>
  @  \result.getKvotient().intValue() == 0;
  @
  @ ensures
  @  stemmerBakEtMandatPrFylke.get(fylkeNr) != null
  @  ==>
  @  \result.getKvotient()
  @         .equals(
  @            createBigDecimal(stemmer)
  @            .divide(
  @               createBigDecimal(
  @                 (2*antallDistriktsmandaterVunnet)+1)
  @               .divide(
  @                 (java.math.BigDecimal)
  @                 stemmerBakEtMandatPrFylke
  @                 .get(fylkeNr),
  @                 20, 4),
  @               20, 4));
  @
  @ ensures
  @  \invariant_for(stemmerBakEtMandatPrFylke) &&
  @  \invariant_for(fylkeNr) &&
  @  \invariant_for(stemmer);
  @
  @ assignable \nothing;
  @ */
private static Kvotient getKvot(Map<String, BigDecimal>
    stemmerBakEtMandatPrFylke, String fylkeNr, Integer
    stemmer, int antallDistriktsmandaterVunnet);
```

Listing 4.26: Method signature and JML specification of method `getKvot` in class `Utgjevningsmandat`.

Listing 4.27 features two methods. The first method is a slightly modified version of original implementation; it has received an additional parameter, `stemmerPrFylke`, and the method body has had a single line of code altered — the original has been commented out in the listing. The second method has identical signature of the original method, but is set to call the first method.

Theses changes are to cope with KeY's lacking support of creating new objects in the specification cases.

The specification case itself states that the following holds for the method:

1. All counties represented in parameter `mandaterPrFylke` are also represented in the result.
2. For all counties in parameter `stemmerPrPartiPrFylke`, the map `stemmerPrFylke` contains, for every key (county), the total number of votes cast for all electoral lists in that county.
3. All counties in the result have had their votes per representative ratio calculated by method `stemmerBakEtMandat(...)` in Listing 4.29.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(mandaterPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke);
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    mandaterPrFylke.containsKey(fylkeNr);
  @    \result.containsKey(fylkeNr));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    stemmerPrPartiPrFylke.containsKey(fylkeNr);
  @    stemmerPrFylke.containsKey(fylkeNr) &&
  @    ((Integer) stemmerPrFylke.get(fylkeNr)).intValue()
  @    ==
  @    (\sum Integer stemmetall;
  @          ((Map) stemmerPrPartiPrFylke
  @                 .get(fylkeNr))
  @           .containsValue(stemmetall);
  @          stemmetall.intValue()));
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    \result.containsKey(fylkeNr);
  @    \result
  @     .get(fylkeNr)
  @     .equals(stemmerBakEtMandat(mandaterPrFylke,
  @                                stemmerPrFylke,
  @                                fylkeNr)));
  @
  @ ensures
  @  \invariant_for(mandaterPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke);
  @
  @ assignable
  @  stemmerPrFylke,
  @  stemmerPrFylke.footprint;
  @*/
private static Map<String, BigDecimal>
    beregnStemmerBakEtMandatPrFylke(Map<String, Integer>
    mandaterPrFylke, Map<String, Map<String, Integer>>
    stemmerPrPartiPrFylke, /*@ nullable @*/ Map<String,
    Integer> stemmerPrFylke) {

// Original
// Map<String, Integer> stemmerPrFylke = new TreeMap<>();

// Modification
stemmerPrFylke = new TreeMap<>();

/*...*/
}

private static Map<String, BigDecimal>
    beregnStemmerBakEtMandatPrFylke(Map<String, Integer>
    mandaterPrFylke, Map<String, Map<String, Integer>>
    stemmerPrPartiPrFylke) {
     beregnStemmerBakEtMandatPrFylke(mandaterPrFylke,
    stemmerPrPartiPrFylke, null);
}
```

Listing 4.27: Method signature and JML specification of method `beregnStemmerBakEtMandatPrFylke` in class `Utgjevningsmandat`.

The method in Listing 4.28 implements Point 6-(f) on page 66. The method itself contains nested map-iterations, which impacts the readability of its specification. More precisely, the specification case states that for all electoral lists in all counties, if the electoral list has been awarded representatives at-large then the following holds:

1. Parameter `kvotienterPrPartiPrFylke` will, for every electoral list in each county, be updated to have *correct* quotient; it is assumed that method `getKvot(...)` in Listing 4.26 always returns the correct quotient. Further, method `getKvot(...)` requires a map of the ratios of votes per representative in the county. This map is calculated by the assumed to be correct method `beregnStemmerBakEtMandatPrFylke` in Listing 4.27. The specification case checks if the electoral lists have received any constituency representatives. If it has, and alters the last parameter to method `getKvot(...)` thereafter. That is, the last parameter to `getKvot` is either 0 or the number of constituency representatives received.

2. The list returned contains a single element — a `PartiFylkeKvotient` with data that corresponds to the data in the parameters, and with the same quotient that was retrieved by method `getKvot(...)` in the previous item.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(antallUtjevningsmandaterPrParti) &&
  @  \invariant_for(distriktsmandatPrPartiPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke) &&
  @  \invariant_for(kvotienterPrPartiPrFylke) &&
  @   invariant_for(mandaterPrFylke);
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    (\forall Map stemmerPrParti;
  @      stemmerPrPartiPrFylke.containsKey(fylkeNr) &&
  @      stemmerPrPartiPrFylke.get(fylkeNr)
  @      == stemmerPrParti;
  @      (\forall String partikode;
  @        (\forall Integer stemmer;
  @          stemmerPrParti.containsKey(partikode) &&
  @          stemmerPrParti.get(partikode) == stemmer;
  @          antallUtjevningsmandaterPrParti
  @          .containsKey(partikode)
  @          ==>
  @          ((( distriktsmandatPrPartiPrFylke
  @              .get(fylkeNr) != null
  @              &&
  @              ((Map) distriktsmandatPrPartiPrFylke
  @              .get(fylkeNr)).get(partikode) != null)
  @             ==>
  @              ( ((Map) kvotienterPrPartiPrFylke
  @                .get(fylkeNr)).containsKey(partikode)
  @                &&
  @                ((Map) kvotienterPrPartiPrFylke
  @                       .get(fylkeNr))
  @                .get(partikode)
  @                ==
  @                getKvot(
  @                  beregnStemmerBakEtMandatPrFylke(
  @                    mandaterPrFylke,
  @                    stemmerPrPartiPrFylke),
  @                  fylkeNr, stemmer,
  @                  ((Integer)
  @                   ((Map) distriktsmandatPrPartiPrFylke
  @                          .get(fylkeNr))
  @                   .get(partikode))
  @                  .intValue())))
  @          &&
  @          ((( distriktsmandatPrPartiPrFylke
  @              .get(fylkeNr) == null
  @              ||
  @              ((Map) distriktsmandatPrPartiPrFylke
  @                       .get(fylkeNr))
  @                .get(partikode) == null)
  @                ==>
  @                ( ((Map) kvotienterPrPartiPrFylke
  @                  .get(fylkeNr)).containsKey(partikode)
  @                  &&
  @                  ((Map) kvotienterPrPartiPrFylke
  @                         .get(fylkeNr))
  @                  .get(partikode)
  @                  ==
  @                  getKvot(
  @                    beregnStemmerBakEtMandatPrFylke(
  @                      mandaterPrFylke,
  @                      stemmerPrPartiPrFylke),
  @                    fylkeNr, stemmer, 0)))
  @            &&
  @            \result.size() == 1
  @            &&
  @            ((PartiFylkeKvotient) \result.get(0))
  @            .getStemmetall() == stemmer
  @            &&
  @            ((PartiFylkeKvotient) \result.get(0))
  @            .getPartikode() == partikode
  @            &&
  @            ((PartiFylkeKvotient) \result.get(0))
  @            .getFylkeNr() == fylkeNr
  @            &&
  @            ((PartiFylkeKvotient) \result.get(0))
  @            .getKvotient() ==
  @            ((Map) kvotienterPrPartiPrFylke
  @                   .get(fylkeNr))
  @            .get(partikode)))));
  @
  @ ensures
  @  \invariant_for(antallUtjevningsmandaterPrParti) &&
  @  \invariant_for(distriktsmandatPrPartiPrFylke) &&
  @  \invariant_for(stemmerPrPartiPrFylke) &&
  @  \invariant_for(kvotienterPrPartiPrFylke) &&
  @   invariant_for(mandaterPrFylke);
  @
  @ assignable
  @  kvotienterPrPartiPrFylke.footprint;
  @*/
private static List<PartiFylkeKvotient>
    beregnPartiFylkeKvotienter(Map<String, Integer>
    antallUtjevningsmandaterPrParti, Map<String, Map<String,
     Integer>> distriktsmandatPrPartiPrFylke, Map<String,
    Map<String, Integer>> stemmerPrPartiPrFylke, Map<String,
     Map<String, Kvotient>> kvotienterPrPartiPrFylke, Map<
    String, Integer> mandaterPrFylke);
```

Listing 4.28: Method signature and JML specification of method `beregnPartiFylkeKvotienter` in class `Utgjevningsmandat`.

Method `stemmerBakEtMandat` in Listing 4.29 is used exclusively as part of method `beregnStemmerBakEtMandatPrFylke` in Listing 4.27 to, for every county, calculate the votes per representative ratio for the county. This is reflected in the specification case, where the returned `BigDecimal` represents the ratio

$$\frac{\textsc{Votes cast}}{\textsc{Representatives awarded}}$$

The specification case employs a helper method `createBigDecimal` representing the construction of a new `BigDecimal`-object. Such a helper method is necessary as KeY-JML does not support the JML-keyword `new`. The method `createBigDecimal` is not displayed, and can be substituted for any similar representation.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(mandaterPrFylke) &&
  @  \invariant_for(stemmerPrFylke) &&
  @  \invariant_for(fylkeNr) &&
  @  stemmerPrFylke.containsKey(fylkeNr) &&
  @  mandaterPrFylke.containsKey(fylkeNr);
  @
  @ ensures
  @  \result.equals(
  @     createBigDecimal((Integer) stemmerPrFylke
  @                               .get(fylkeNr))
  @     .divide(createBigDecimal(
  @               (Integer) mandaterPrFylke
  @                         .get(fylkeNr)),
  @               20, 4));
  @
  @ ensures
  @  \invariant_for(mandaterPrFylke) &&
  @  \invariant_for(stemmerPrFylke) &&
  @  \invariant_for(fylkeNr) &&
  @  \fresh(\result);
  @
  @ assignable \nothing;
  @*/
private static BigDecimal stemmerBakEtMandat(Map<String,
    Integer> mandaterPrFylke, Map<String, Integer>
    stemmerPrFylke, String fylkeNr);
```

Listing 4.29: Method signature and JML specification of method `stemmerBakEtMandat` in class `Utgjevningsmandat`.

The method `beregnDistriktsmandaterPrParti` displayed in Listing 4.30 above was implemented to fulfill Point 6-(c) on page 66. The method iterates through all counties and summarized the number of constituency representatives given to each electoral list. This is captured by the specification case, where the resulting map must contain keys representing electoral lists that appears in any of the counties, and the value of the key must be the sum of all numbers of constituency representatives for the represented electoral list throughout the counties. The first `ensure`-clause restricts the keys, while the second `ensure`-clause restricts the values of those keys.

```
/*@ public normal_behaviour
  @ requires
  @  \invariant_for(distriktsmandaterPrPartiPrFylke);
  @
  @ ensures
  @  (\forall String fylkeNr;
  @    distriktsmandaterPrPartiPrFylke
  @    .containsKey(fylkeNr);
  @    (\forall String partikode;
  @      ((Map) distriktsmandaterPrPartiPrFylke
  @             .get(fylkeNr))
  @      .containsKey(partikode);
  @      \result.containsKey(partikode)));
  @
  @ ensures
  @  (\forall String partikode;
  @    \result.containsKey(partikode);
  @    ((Integer) \result.get(partikode)).intValue() ==
  @    (\sum String fylkeNr;
  @      distriktsmandaterPrPartiPrFylke
  @      .containsKey(fylkeNr)
  @      &&
  @      ((Map) distriktsmandaterPrPartiPrFylke
  @             .get(fylkeNr))
  @      .containsKey(partikode);
  @      ((Integer)
  @       ((Map) distriktsmandaterPrPartiPrFylke
  @              .get(fylkeNr))
  @       .get(partikode))
  @      .intValue()));
  @
  @ ensures
  @  \invariant_for(distriktsmandaterPrPartiPrFylke) &&
  @  \fresh(\result);
  @
  @ assignable \nothing;
  @*/
private static Map<String, Integer>
    beregnDistriktsmandaterPrParti(Map<String, Map<String,
    Integer>> distriktsmandaterPrPartiPrFylke);
```

Listing 4.30: Method signature and JML specification of method `beregnDistriktsmandaterPrParti` in class `Utgjevningsmandat`.

4.5 Formal Verification

The first step in the verification process is to prepare the source code. As discussed in Section 3.4, there are some limitations to the KeY System that cannot be disregarded. In order to prepare the source code, measures were taken in the following order:

1. Partition the system based on dependency, in a bottom-up approach; the parts of the system that contain the least internal dependencies should be verified first.

2. Replace all non-ASCII characters (e.g. æ/Æ, ø/Ø, å/Å) from the source code. For instance, class Område had to be renamed to Omrade.

3. Create stubs for functionality imported from external libraries. There exists an Eclipse-extension for creating generic stubs, which can later be refined.

4. Remove generics, and cast the objects to type specified by the generic.

5. Re-write any unsupported Java features (e.g. Lambda-functions).

6. Create auxiliary verification artifacts, such as loop invariants and block contracts.

Then, if the implementation conforms to the specification, KeY will be able to automatically prove it. This is assumed that KeY has sufficient computational resources available, and assumed that KeY does not crash. For some contracts, the computational resources required becomes immense due to the number of branches in the resulting proof tree that has to be closed. Further, during the verification effort some issues with KeY regarding automatic theorem proving was discovered; these issues are discussed in Section 4.5.5.

For the verification effort in this thesis, the *proof search strategy* selected in KeY has been *Java verif. std.*, with the exceptions that *Stop at* is set to *Uncloseable*, and the *Max. Rule Applications*-slider is set to the highest setting.

KeY was run on a computer with the following hardware and specifications.

Processor	Intel® Core™ i5-4210M CPU @ 2.60GHz
Memory	8GB RAM
OS	Windows 10 Enterprise v1803
Graphics Card	Intel® HD Graphics 4600
Storage	Liteon Zeta 256GB SSD SATA3

Table 4.16: Computer Specifications

4.5.1 Verifying the Quotients

The classes with the least dependencies in package `no.valg.valgnatt.backend` are the classes that represent the quotients — `Kvotient` and `PartiKvotient`. As such, these will be the leading verification targets.

Both classes made use of the `BigDecimal`-class from package `java.math`. Stubs for `BigDecimal` was generated by the Eclipse-extension. However, as the method `BigDecimal.compareTo(...)` are only ever utilized as part of specifications, it was not automatically generated. As such, the following signature was appended at the end of the generated `BigDecimal`-stub:

```
public int compareTo(java.math.BigDecimal param0);
```

The generic stubs were modified to more precisely represent `BigDecimal` as implemented in `java.math`. The stubs used for the verification effort are displayed in Listing 4.31 below.

```
package java.math;

public final class BigDecimal {

  //@ public instance ghost \bigint rValue;

  /*@ public normal_behavior
    @ ensures rValue == n;
    @ assignable rValue;
    @*/
  public BigDecimal(int n);

  /*@ public normal_behavior
    @ ensures \result == rValue;
    @ assignable \strictly_nothing;
    @*/
  public int intValue();

  /*@ public normal_behavior
    @ ensures \fresh(\result);
    @ ensures \result.rValue == rValue * bd.rValue;
    @ ensures \invariant_for(bd);
    @ assignable \nothing;
    @*/
  public BigDecimal multiply(BigDecimal bd);

  /*@ public normal_behavior
    @ requires bd.rValue != 0;
    @ ensures \fresh(\result);
    @ ensures \result.rValue == rValue / bd.rValue;
    @ ensures \invariant_for(bd);
    @ assignable \nothing;
    @*/
  public BigDecimal divide(BigDecimal bd, int precision,
   int rounding);

  /*@ public normal_behavior
    @ ensures \fresh(\result);
    @ ensures \result.rValue == rValue + bd.rValue;
    @ ensures \invariant_for(bd);
    @ assignable \nothing;
    @*/
  public BigDecimal add(BigDecimal bd);

  /*@ public normal_behavior
    @ ensures \fresh(\result);
    @ ensures \result.rValue == rValue - bd.rValue;
    @ ensures \invariant_for(bd);
    @ assignable \nothing;
    @*/
  public BigDecimal subtract(BigDecimal bd);

  /*@ public normal_behavior
    @ requires \invariant_for(bd);
    @ ensures \invariant_for(bd);
    @ ensures bd.rValue > this.rValue ==> \result == -1;
    @ ensures bd.rValue < this.rValue ==> \result == 1;
    @ ensures bd.rValue == this.rValue ==> \result == 0;
    @ assignable \strictly_nothing;
    @*/
  public int compareTo(BigDecimal bd);
}
```

Listing 4.31: Stubs for class `BigDecimal`.

One important note is that the type of the ghost variable `rValue` is `bigint`. This variable is therefore unable to properly represent decimals in the similar manner as Java's `BigDecimal`-class can. However, the specific value of the variable `rValue` is not as important as how the modification of the value is specified. For instance, consider the following methods and their specifications:

```
/*@ ensures
  @  \result.kvotient.rValue ==
  @  this.kvotient.rValue + k.rValue;
  @ assignable \nothing;
  @*/
public Kvotient addDirectly(Kvotient k) {
  return new Kvotient(this.kvotient.add(k.getKvotient()));
}


```

```
/*@ ensures
  @  \result.kvotient.rValue ==
  @  (this.kvotient.add(k.kvotient)).rValue;
  @ assignable \nothing;
  @*/
public Kvotient addIndirectly(Kvotient k) {
  return new Kvotient(this.kvotient.add(k.getKvotient()));
}
```

Both methods have the exact same implementation, both conform to their specifications, and both specify that the result has a value that is the sum of the value of `this` and the parameter's value. The dissimilarity is that the first method specifies the behavior through direct arithmetic manipulation of the non-ideal variable `rValue`, while the second method specifies the behavior indirectly through the method `add` of `BigDecimal`. Therefore, the second method does not need to concern itself with direct manipulation of the value of the variable `rValue`, and instead rely on the library method to modify the value of the variable correctly. It is mostly this approach that is taken for the remainder of this work; the variable `rValue` is only manipulated directly when new `BigDecimal`-objects are created based on primitives instead of other `BigDecimal`- or `Kvotient`-objects, e.g., in the case of the constructor `Kvotient(int, int)`, it is not possible to indirectly specify the value of the created object's `kvotient` field. Ideally, the specifications would not need to access the variable `rValue` at all, but no suitable alternative, that would not immensely complicate the verification effort, was found.

Class `Kvotient` required no changes to its source code.

For class `PartiKvotient` we propose the following changes. The original class had implemented the `compareTo`-algorithm in a suspect manner. The original code was:

```
public int compareTo(PartiKvotient partiKvotient) {
  Kvotient diff = getKvotient()
                   .subtract(partiKvotient
                             .getKvotient());

  return diff.getKvotient()
              .compareTo(BigDecimal.ZERO) > 0 ? -1 :
               (diff.getKvotient()
                     .compareTo(BigDecimal.ZERO) < 0 ? 1 :
         sammenlignHvisKvotienterErLike(partiKvotient));
}
```

In other words, to test if the quotient of `this` is larger than the parameter, the difference between them was compared to zero. This does provide the intended result, but it is unnecessarily complex. The same behavior can be computed with a single call to `BigDecimal.compareTo`, while removing the reliance on class `Kvotient` and the constant `BigDecimal.ZERO`. The proposed solution displayed in Listing 4.32 conforms to the specification in Listing 4.8 from page 54.

```java
public int compareTo(PartiKvotient partiKvotient) {
  int comp = getKvotient()
             .getKvotient()
             .compareTo(partiKvotient
                        .getKvotient()
                        .getKvotient());

  return  (comp != 0) ? (-1 * comp) :
    sammenlignHvisKvotienterErLike(partiKvotient);
}
```

Listing 4.32: Proposed change of the `compareTo`-method in class `PartiKvotient`.

Furthermore, `PartiKvotient` utilized the wrapper class `java.lang.Integer` which complicated the verification effort to such a degree that all instances of `Integer` were changed to the primitive `int`. There are notable differences between the wrapper-class and the primitive [64]; however, for the properties proven throughout this verification effort, these differences are insignificant.

KeY was able to create method contracts (see Section 3.2) for the implemented methods based on the JML specifications; all method contracts for classes `Kvotient` and `PartiKvotient` were automatically verified. The proofs are found in [47].

4.5.2 Verifying the Representatives

The specifications of class `Mandater`, class `MandatData` and class `PartiMandater` were straight-forward to verify; all specified methods conform to their specification. The proofs are found in [47].

When it comes to the specifications of class `Valgdistrikt` presented in Listing 4.10, all methods but one were straight forward to verify. The method `getStemmerPrPartiTilMandatberegning` was the only one that required special attention as it contains a loop. In order to limit possible sources of errors early, a model of the aforementioned method was created. The model would loop over elements of an array `pmArr` received as a parameter, instead of looping over the elements of the array retrieved from the public method `getPartiMandater`. As such, two main changes were made for the specification in Listing 4.10: all occurrences of `getPartiMandater()` was changed to `pmArr`, and all occurenses of `\result.size()` were changed to `pmArr.length`.

Furthermore, the returning element of the model is an instance of class `TreeMap` instead of an instance of Interface `Map`; the class `TreeMap` and the challenges it imposes is presented in Section 4.5.3.

The model and the loop invariants utilized are found in Listing 4.33. As the required computational resources increase drastically when adding loop invariants with quantified expressions, the loop invariants and respective postconditions were initially verified separately, and only when all loop invariants were proven valid in isolation they were put together to complete the full proof. This was possible to do as loop invariants are closed under conjunction, as discussed in Section 2.3.7. The loop invariants were proven in the following order:

1. The three first auxiliary loop invariants were the first to be verified. Although these invariants are not needed for verifying any of the postconditions directly, they were used to prove the remaining loop invariants; the following loop invariants cannot be proven separately from the three first loop invariants.

2. The forth loop invariant states that if the parameter `fjernBlanke` is true then the map will not contain the key defined by the constant field `BLANKE`. The fifth loop invariant is a stronger version stating that if the parameter is true, then all elements of the parameter `pmArr` are represented in the map except those that have the party-code field equal to `BLANKE`. Both invariants were verified separately and in parts. Verifying invariants in parts can be useful when operating with limited computational resources, i.e.: verify a branch, save the proof (if possible[15]), restart the program to free memory, verify another branch, and repeat until all branches are verified.

3. the sixth loop invariant says that if parameter `fjernBlanke` is false, then *all* electoral lists, including `BLANKE`, will be present in the resulting map.

4. The seventh loop invariant states that if either of `isStillerList()`, `borBenytteListestemmer` or `listestemmerFinnes` is false, then the regular votes will be used. The eight loop invariant states the opposite: if all are true, personal votes will be used. Both loop invariants where verified separately and in parts.

5. Finally, all loop invariants and postconditions were verified combined. It was straight-forward to prove that the loop invariants held initially, and that they were strong enough to prove the postconditions. Further, due to limited computational resources, it was not possible to prove that they were all upheld by the method's body in one execution. Therefore, the proof is split into several executions; there are some overlap between executions, but it was ensured that every branch of the proof tree was verified by *at least* one execution. In [47], each execution is found in a file named similar to `getStemmerPrPartiTilMandatberegning-X.proof` along with a corresponding screenshot of the KeY Proof-Tree Display showing which branches were closed during the execution. Furthermore, every execution involved approximately 350k automatic rule applications, as any

[15]KeY is at times unable to save large proof trees. See Section 4.5.5 for further information.

more often resulted in the proof tree not being properly saved. The specification for the model is found in the file `Valgdistrikt.java` in [47], along with methods proving the loop invariants in isolation.

The case where the parameter `prognose` is true is ignored since this work is not concerned with verifying *forecasts-only* elements. Further, every *if/then*-statement induces a branch that may double the size of the emerging proof tree, meaning the resulting proof tree might become too big to verify given limited computational resources. As a consequence, the parameter `prognose` was removed from the model's signature, and the model's implementation represents the source code in the case where `prognose` is false.

```
private TreeMap getStemmerPrPartiTilMandatberegning(boolean
     borBenytteListestemmer, boolean fjernBlanke,
    PartiMandater[] pmArr) {

  TreeMap stemmerPrParti = new TreeMap();

  /*@ maintaining 0 <= \index && \index <= pmArr.length;
    @ maintaining \invariant_for(stemmerPrParti);
    @ maintaining
    @  (\forall int i; 0 <= i && i < pmArr.length;
    @   \invariant_for(pmArr[i]) &&
    @   \invariant_for(pmArr[i].resultatData) &&
    @    pmArr[i] != null &&
    @    pmArr[i].partikode != null &&
    @    pmArr[i].resultatData != null);
    @
    @ maintaining fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \index;
    @    !(stemmerPrParti.containsKey(BLANKE)));
    @
    @ maintaining fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \index &&
    @   !(pmArr[i].partikode.equals(BLANKE));
    @    stemmerPrParti.containsKey(pmArr[i].partikode));
    @
    @ maintaining !fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \index;
    @    stemmerPrParti.containsKey(pmArr[i].partikode));
    @
    @ maintaining !fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \index &&
    @    !(pmArr[i].isStillerliste() &&
    @      borBenytteListestemmer &&
    @      listestemmerFinnes(pmArr[i].resultatData));
    @    stemmerPrParti.get(pmArr[i].partikode) ==
    @    pmArr[i].resultatData.getStemmer());
    @
    @ maintaining !fjernBlanke ==>
    @  (\forall int i; 0 <= i && i < \index &&
    @    (pmArr[i].isStillerliste() &&
    @     borBenytteListestemmer &&
    @     listestemmerFinnes(pmArr[i].resultatData));
    @   stemmerPrParti.get(pmArr[i].partikode) ==
    @   pmArr[i].resultatData.getListestemmer());
    @
    @ decreasing pmArr.length - \index;
    @ assignable \nothing;
    @*/
  for (PartiMandater pm : pmArr) {

    if (pm.getPartikode().equals(BLANKE) && fjernBlanke) {
      continue;
    }
    MandatData md = prognose ? pm.getPrognoseData() : pm.
   getResultatData();
    if (pm.isStillerliste()) {
      stemmerPrParti.put(pm.getPartikode(),
   borBenytteListestemmer && listestemmerFinnes(md) ? md.
   getListestemmer() : md.getStemmer());
    }
    stemmerPrParti.put(pm.getPartikode(), md.getStemmer());
  }
  return stemmerPrParti;
}
```

Listing 4.33: Body and loop invariants of the proposed model of method `getStemmerPrPartiTilMandatberegning` of class `Valgdistrikt`.

4.5.3 Verifying the Sorted Quotients

In addition to the previously created stubs for `BigDecimal` (Listing 4.31), stubs had to be created for `java.lang.System.arraycopy` (Listing 4.34) and `java.lang.Arrays.sort` (Listing 4.35). As generics are unsupported by KeY, these stubs are modified to only accept arrays of type `PartiKvotient[]`. However, they can easily be modified to accept other interfaces or classes.

The stub for `arraycopy` in Listing 4.34 is obviously very limited; only the cases where `srcPos` is zero and `src.length` equals `length` are considered. However, these are the only cases relevant for the verification target, and the stub is easily extendable if required. The stub utilizes the KeY-JML predicate `\seqPerm(s1, s2)` which is true if `s2` is a permutation of `s1`. Be aware that [17] contains contradictory statements on the meaning of this predicate: Page 152 states that `seqPerm(s1, s2)` is "true if `s2` is a permutation of `s1`", while Page 639 states that `seqPerm(s1, s2)` holds "iff the first sequences is a permutation of the other". KeY considers the former to be true.

```
package java.lang;

import PartiKvotient;

public final class System {

  /*@ public normal_behavior
    @ requires dest != null;
    @ requires srcPos == destPos && destPos == 0;
    @ requires 0 <= length && dest.length == length &&
    @          src.length == dest.length;
    @
    @ ensures dest != null;
    @ ensures (\forall int i;
    @   0 <= i && i < dest.length; dest[i] != null);
    @
    @ ensures
    @   \dl_seqPerm(\dl_array2seq(src),
    @               \dl_array2seq(dest));
    @
    @ assignable dest[*];
    @
    @ also
    @
    @ public normal_behavior
    @ requires dest != null;
    @ requires srcPos == destPos && destPos == 0;
    @ requires 0 <= length && dest.length == length &&
    @                         src.length > dest.length;
    @
    @ ensures \invariant_for(src);
    @ ensures dest != null;
    @ ensures (\forall int i;
    @   0 <= i && i < dest.length; dest[i] != null);
    @
    @ ensures
    @   \dl_seqPerm(\dl_array2seq(src)[0..length],
    @               \dl_array2seq(dest));
    @
    @ assignable dest[*];
    @
    @ also
    @
    @ public normal_behavior
    @ requires dest != null;
    @ requires srcPos == 0 && destPos > 0;
    @ requires 0 <= length &&
    @          dest.length >= length + destPos &&
    @          src.length == length;
    @
    @ ensures dest != null;
    @
    @ ensures
    @   \dl_seqPerm(\dl_array2seq(src),
    @       \dl_array2seq(dest)[destPos..destPos+length]);
    @
    @ assignable dest[destPos..destPos+length];
    @*/
  public static void arraycopy(/*@ non_null @*/
   PartiKvotient[] src, int srcPos, /*@ nullable @*/
   PartiKvotient[] dest, int destPos, int length);
}
```

Listing 4.34: Stub for the method `java.lang.System.arraycopy`.

```
package java.util;

import PartiKvotient;

public class Arrays{
  /*@ public normal_behavior
    @ requires a != null;
    @ ensures a.length == \old(a).length;
    @
    @ ensures
    @  \dl_seqPerm(\dl_array2seq(\old(a)),
    @              \dl_array2seq(a));
    @
    @ ensures
    @  (\forall int i; 0 <= i && i < a.length-1;
    @    a[i].compareTo(a[i+1]) <= 0);
    @
    @ ensures \invariant_for(a);
    @ assignable a[*];
    @*/
  public static void sort(PartiKvotient[] a);
}
```

Listing 4.35: Stubs for the method `java.lang.Arrays.sort`.

The process of creating a stub for representing the data structure `java.util.Map` posed some challenges. One idea can be to have a simple list modelled by the built-in theory *sequent* (see Chapter 5.2 of [17]) that could hold instances of a class representing `Map.Entry`. However, there seems to exists no way of representing that a new object is created in JML without utilizing the keyword `new`[16], meaning that creating and appending new entries to the list is impossible in KeY-JML. A workaround could be to create the objects in Java instead

[16]The JML keyword `new` is unsupported by KeY.

of JML, but this would require quite drastic changes to the source-code; more specifically, all occurrences of `Map.put(key, value)` would have to be replaced by `Map.put(new Entry(key, value))`. Such a change could have unforeseen consequences for the encapsulating system.

A second attempt to create a KeY-JML `map`-structure can be to try to utilize KeY's theory of *map* (See Appendix B.1.9 of [17]). However, this theory lacks the functionality that was provided by sequents, e.g., `seqLen` or `values`, and functionality expected from `java.util.Map`, e.g., `keySet` or `entrySet`. Furthermore, several of the taclets in KeY that are applicable to the *map* theory are restricted to interactive theorem proving only, meaning that specifications using the theory could not be automatically verified.

We have settle for a third attempt, where we implemented a map-structure based on two distinct sequents — one for keys, and one for values. The idea is that `Map.get(key)` would be represented by `values[\dl_seqIndexOf(keys, key)]`. In other words, a key/value-pair would be represented by `keys[i]/values[i]` for any given index in range. The implementation is presented in Listing 4.38, and will be utilized where maps are necessary. The implementation was motivated by examples in [17] and the KeY-JML map in [33]. Further, the implementation is tested by the methods in Listing 4.39 which both conform to their given specification. However, KeY had trouble proving some trivial properties when quantifying over the sequents. This issue is demonstrated in Section 4.5.5. Still, the data structure proved sufficient for the verification effort at hand.

Before moving on, it should be noted that there exists a framework known as MultiJava [71] that has formalized several Java libraries in JML. However, the framework seems to be developed specifically for unit testing with JML and JUnit [49], and lacks specifications of important classes s.a. `BigDecimal`. The framework is not well suited for deductive verification using KeY because the MultiJava framework is about as nested as the Java framework; one could just as well import the Java code from libraries directly. The MultiJava documentation should still be considered a useful resource as it does display the full capabilities of JML, and can be a source of inspiration when looking to implement JML data structures.

Due to the problems with implementation of a KeY-JML data structure representing a map, a model (see Listing 4.37) of the original method in Listing 4.11 was created. The original algorithm for creating quotients would calculate a divisor, iterate over all entries in a supplied map of type `<String, Integer>`, create `PartiKvotient`-objects, and call `setStemmetall(value)` on the newly created objects. The objects themselves were created by invoking the constructor of `PartiKvotient` with parameters

- the key of the entry,
- a fresh object created by invoking the constructor of `Kvotient` with the key, then calling divide on the fresh object with the divisor, and
- the divisor.

The proposed model accepts a divisor and two distinct arrays — one for keys and one for values. The idea is the same as before; the model assumes that for every index `i`, key `keys[i]` is the key for value `values[i]`. The model returns a new array with the same number of elements as keys. The original algorithm can potentially compute quotients for several divisors, but the model only accepts a single divisor in order to limit complexity. A prior model was created that would create quotients for `n` number of divisors. However, this model became to complex to be able to verify with the limited resources available, or it ran into an issue where certain trivial conditions were unverifiable by KeY. This issue is also demonstrated in Section 4.5.5. Instead, the proposed solution involves calling the proposed model `n` times with `n` different divisor and appending the resulting arrays.

The original implementation of the method was

```java
int antallKvotienter =
    antallMandater * partiStemmeMap.size();
PartiKvotient[] kvotienter =
    new PartiKvotient[antallKvotienter];
int index = 0;

for (int n = 0; n < antallMandater; n++) {
  BigDecimal delingstall = getSainteLagueDelingstall(n);

  for (Map.Entry<String, Integer> partiStemme :
    partiStemmeMap.entrySet()) {

    PartiKvotient partiKvotient =
        new PartiKvotient(partiStemme.getKey(), new
        Kvotient(partiStemme.getValue())
        .divide(delingstall), delingstall);

    partiKvotient.setStemmetall(partiStemme.getValue());
    kvotienter[index++] = partiKvotient;
  }
}
```

while the *proposed model* is

```java
/*
 * Calculate String[] keys and int[] values from
 * partiStemmeMap, for instance by calling
 * keySet().toArray() and .values.toArray() and ensuring
 * that the order is correct.
 */
int antallKvotienter = antallMandater * keys.length;
PartiKvotient[] kvotienter = new PartiKvotient[
    antallKvotienter];

for (int n = 0; n < iterations; n++){
  BigDecimal delingstall = getSainteLagueDelingstall(n);
  PartiKvotient[] pkArr =
      createQuotients(keys, values, delingstall);
  System.arraycopy(pkArr, 0, res,
      keys.length*n, pkArr.length);
}
```

Listing 4.36: Proposed alternative implementation of `sorterteKvotienter` that is more suitable for verification.

The method `createQuotients` can be seen in Listing 4.37 below. The method is proven to conform to its specification; the proof is found alongside other proofs in the repository *EVA-KeY* [47].

```
/*@ public normal_behavior
  @ requires \invariant_for(delingstall) &&
  @          delingstall.rValue > 0;
  @ requires keys.length == values.length;
  @ requires keys.length > 0;
  @ requires
  @  (\forall int i;
  @    0 <= i && i < values.length;
  @    values[i] >= 0);
  @
  @ ensures \result.length == keys.length;
  @
  @ ensures
  @  (\forall int i;
  @   0 <= i && i < \result.length;
  @   \invariant_for(result[i]) &&
  @   \result[i].getPartikode() == keys[i] &&
  @   \result[i].getDelingstall() == delingstall &&
  @   \result[i].getStemmetall() == values[i] &&
  @   \result[i].kvotient.kvotient.rValue ==
  @   values[i] / delingstall.rValue);
  @
  @ assignable \nothing;
  @*/
public PartiKvotient[] createQuotients(String[] keys, int[]
     values, BigDecimal delingstall){
  PartiKvotient[] res = new PartiKvotient[keys.length];
  int index = 0;
  /*@ maintaining \index >= 0 && \index <= keys.length;
    @ maintaining \invariant_for(delingstall);
    @ maintaining delingstall.rValue > 0;
    @ maintaining
    @  (\forall int i; 0 <= i && i < \index;
    @    res[i] != null && \invariant_for(res[i]) &&
    @    res[i].getPartikode() == keys[i] &&
    @    res[i].getDelingstall() == delingstall &&
    @    res[i].getStemmetall() == values[i] &&
    @    res[i].kvotient.kvotient.rValue
    @    == values[i] / delingstall.rValue);
    @ maintaining index == \index;
    @ decreasing keys.length - \index;
    @ assignable res[*];
    @*/
  for (String s : keys){
    PartiKvotient pk = new PartiKvotient(s, new Kvotient(
   values[index]).divide(delingstall), delingstall);
    pk.setStemmetall(values[index]);
    res[index++] = pk;
  }
  return res;
}
```

Listing 4.37: Proposed model for creating new quotients.

The rest of the original method involves calling the library method `Arrays.sort`, and limiting the number of returned quotients by calling the method `begrensAntallKvotienter` from Listing 4.13. The former is a stub assumed to be correct, and the latter was proven to conform to its specification by KeY.

A full proof of the method listed in Listing 4.11 was not possible due to issues of complexity arising from the number of quantified loop invariants, pre- and postconditions, and issues surrounding implementing KeY-JML data structures. However, by going over the specified postconditions, one can discuss if they intuitively hold or not:

- The first postcondition is decided by method `begrensAntallKvotienter` which, as mentioned, conforms to its specification. Therefore, there is some certainty that the postcondition holds.
- The second postcondition is decided by the double loop at the beginning of the method's body, which was proposed changed to the model in Listing 4.36. The proposed model has the required properties and loop invariants proven for the inner loop, but it stands to prove that the properties hold for the outer loop. There was an issue with proving that the invariants for the map held after creating a new `BigDecimal`-object.
- The third postcondition is decided by method `Arrays.sort`, which is a stub that is assumed to be correct. If the actual library method `Arrays.sort` conforms to the specification in Listing 4.35, the postcondition holds.
- The forth postcondition is trivial as the parameter is not altered throughout the method.

There are still scenarios where all postconditions may hold without the expected behavior being displayed by the system, and the postconditons have not been explicitly proven. Still, when combining the proven properties and loop invariants with less formal methods such as different testing techniques [27] and static analysis [41], a higher degree of trust can be placed in the method. Also, during testing, debugging [9] becomes easier as every proven condition can eliminate one or more possible sources of error. In conclusion, we believe that given enough resources and a satisfactory KeY-JML theory of maps, all the postconditions of the method could be automatically proven valid.

4.5.4 Verifying the Constituency Representatives

The previous section explained the process of creating a map structure using KeY-JML for verification purposes. This section presents the result of that process, and the verification of the method `mandatFordeling` from Listing 4.15. The proposed map structure is presented in Listing 4.38, and the tests used to verify that the map structure exhibits the expected behavior is presented in Listing 4.39. The section concludes by presenting the verification effort and its results.

```
package java.util;

public final class TreeMap{

  //@ spec_public instance ghost \seq keys;
  //@ spec_public instance ghost \seq values;

  //@ instance invariant \dl_seqLen(keys) == \dl_seqLen(
    values);

  /*@ public normal_behavior
    @ ensures keys == \seq_empty;
    @ ensures values == \seq_empty;
    @ assignable this.keys, this.values;
    @*/
  public TreeMap();

  /*@ public normal_behavior
    @ // Case where key is not in map
    @ requires !(containsKey(key));
    @ ensures
    @  keys ==
    @  \seq_concat(\seq_singleton(key), \old(keys));
    @ ensures
    @  values ==
    @  \seq_concat(\seq_singleton(value), \old(values));
    @ assignable this.keys, this.values;
    @
    @ also
    @
    @ public normal_behavior
    @ requires containsKey(key);
    @
    @ // Case where key is in the first element of the map
    @ ensures
    @  \dl_seqIndexOf(keys, key) == 0 ==>
    @    values ==
    @      \seq_concat(\seq_singleton(value),
    @                  \old(values[1..keys.length]));
    @
    @ // Case where key is in the last element of the map
    @ ensures
    @  \dl_seqIndexOf(keys, key) > 0 &&
    @  \dl_seqIndexOf(keys, key) == keys.length - 1 ==>
    @    values ==
    @      \seq_concat(\old(values[0..keys.length-1]),
    @                  \seq_singleton(value));
    @
    @ // Case where key is anywhere else in the map
    @ ensures
    @  0 < \dl_seqIndexOf(keys, key) &&
    @  \dl_seqIndexOf(keys, key) < keys.length-1 ==>
    @    values ==
    @      \seq_concat
    @        (\old(values[0..\dl_seqIndexOf(keys, key)]),
    @         \seq_concat(\seq_singleton(value), \old(
    @  values[\dl_seqIndexOf(keys, key)+1..keys.length])));
    @
    @ ensures \invariant_for(key);
    @ assignable this.values;
    @*/
  public void put(String key, int value);

  /*@ public normal_behavior
    @ ensures
    @  \result ==
    @    (\exists int i; 0 <= i && i < keys.length;
    @     ((String) keys[i]).equals(key));
    @
    @ assignable \strictly_nothing;
    @*/
  public boolean containsKey(String key);

  /*@ public normal_behavior
    @ ensures
    @  \result ==
    @    (\exists int i; 0 <= i && i < values.length;
    @     ((int) values[i]) == value);
    @
    @ assignable \strictly_nothing;
    @*/
  public boolean containsValue(int value);

  /*@ public normal_behavior
    @ requires containsKey(key);
    @ ensures
    @  \result == (int) values[\dl_seqIndexOf(keys, key)];
    @ ensures \invariant_for(key);
    @ assignable \strictly_nothing;
    @*/
  public int get(String key);


  /*@ public normal_behavior
    @ ensures \result == (String[]) keys;
    @ assignable \strictly_nothing;
    @*/
  public String[] keySet();

  /*@ public normal_behavior
    @ ensures \result == (int[]) values;
    @ assignable \strictly_nothing;
    @*/
  public int[] values();

  /*@ public normal_behavior
    @ ensures \result == keys.length;
    @ assignable \strictly_nothing;
    @*/
  public int size();

}
```

Listing 4.38: JML specification of a map structure.

```
/*@ public normal_behavior
  @ requires !(key.equals(key2)) &&
  @          !(key.equals(key3)) &&
  @          !(key2.equals(key3));
  @ ensures \result.get(key) != value;
  @ ensures \result.get(key) == value+2;
  @ ensures \result.get(key2) != value+1;
  @ ensures \result.get(key2) == value+4;
  @ ensures \result.get(key3) != value+3;
  @ ensures \result.get(key3) == value+5;
  @ assignable \nothing;
  @*/
public TreeMap validMapTestOne(String key, String key2,
    String key3, int value){
  TreeMap map = new TreeMap();
  map.put(key, value);
  map.put(key2, value+1);
  map.put(key, map.get(key)+2);
  map.put(key3, value+3);
  map.put(key2, map.get(key2)+3);
  map.put(key3, map.get(key3)+2);
  return map;
}

/*@ public normal_behavior
  @ requires !(key.equals(key2)) &&
  @          !(key.equals(key3)) &&
  @          !(key2.equals(key3));
  @ ensures \result.get(key) != value;
  @ ensures \result.get(key) == value+2;
  @ ensures \result.get(key2) != value+1;
  @ ensures \result.get(key2) == value+4;
  @ ensures \result.get(key3) != value+3;
  @ ensures \result.get(key3) == value+5;
  @ assignable \nothing;
  @*/
public TreeMap validMapTestTwo(String key, String key2,
    String key3, int value){
  TreeMap map = new TreeMap();
  if (map.containsKey(key)){
    map.put(key, map.get(key)+2);
  } else {
    map.put(key, value);
  }
  if (map.containsKey(key2)){
    map.put(key2, map.get(key2)+3);
  } else {
    map.put(key2, value+1);
  }
  if (map.containsKey(key3)){
    map.put(key3, map.get(key3)+2);
  } else {
    map.put(key3, value+3);
  }
  return map;
}
```

Listing 4.39: Tester methods for method in Listing 4.38.

The implementation of the method in Listing 4.15 is presented in Listing 4.40 together with annotated loop invariants. The key change that has been made to the original source code is that the method now returns an explicit `TreeMap`-object; the original implementation returned an implicit `TreeMap`-object.

```
private static TreeMap validInitalAndUseCase(PartiKvotient
    [] kvotienter) {
  TreeMap partiMandatMap = new TreeMap();
```

```
/*@ maintaining
  @  0 <= \index && \index <= kvotienter.length;
  @
  @ maintaining
  @  \invariant_for(partiMandatMap);
  @
  @ maintaining
  @  (\forall int i; 0 <= i && i < kvotienter.length;
  @   \invariant_for(kvotienter[i]) &&
  @   \invariant_for(kvotienter[i].partikode) &&
  @    kvotienter[i] != null &&
  @    kvotienter[i].partikode != null);
  @
  @ maintaining
  @  (\forall int i;
  @    0 <= i && i < \index;
  @    partiMandatMap.keySet()[i] != null);
  @
  @ maintaining
  @  (\forall int i;
  @    0 <= i && i < partiMandatMap.size() &&
  @    partiMandatMap.keySet()[i] != null;
  @    !(\exists int j;
  @       i < j && j < partiMandatMap.size();
  @       partiMandatMap.keySet()[i]
  @       .equals(partiMandatMap.keySet()[j])));
  @
  @ maintaining
  @  (\forall int i;
  @    0 <= i && i < partiMandatMap.size() &&
  @    partiMandatMap.keySet()[i] != null;
  @    partiMandatMap.get(partiMandatMap.keySet()[i]) ==
  @    (\num_of int j;
  @      0 <= j && j < \index &&
  @      kvotienter[j].getPartikode()
  @      .equals(partiMandatMap.keySet()[i])));
  @
  @ decreasing kvotienter.length - \index;
  @
  @ assignable partiMandatMap.keys,
  @            partiMandatMap.values;
  @*/
for (PartiKvotient kvotient : kvotienter) {
  String partikode = kvotient.getPartikode();
  if (partiMandatMap.containsKey(partikode)) {
    int antallMandaterForPartiet = partiMandatMap.get(
 partikode);
    partiMandatMap.put(partikode,
 antallMandaterForPartiet + 1);
```

```
    } else {
      partiMandatMap.put(partikode, 1);
    }
  }
  return partiMandatMap;
}
```

Listing 4.40: Proposed loop invariants for verifying conformance with the specification in Listing 4.15.

The loop invariants defined by the `maintaining` clauses are as follows:

1. The index of the *enhanced-for* loop, represented by `\index`, ranges from `0` to the length of the array `kvotienter` that the loop iterates over.
2. The invariant for the map is upheld by the loop.
3. Loop invariant representing the precondition that the elements of the supplied array is non-null, their invariants hold, and they all have non-null `partikode` fields.
4. The map's key-set does not contain null.
5. Invariant for the first postcondition: no key exists twice in the map's key-set.
6. Invariant for the second postcondition: all keys have values that represent how many times the key has occurred in the elements of `kvotienter`.

The first four loop invariants are proven to hold initially and are proven to be upheld by the loop's body. However, they are not strong enough to prove the method's postconditons, and merely exists to assist the two final loop invariants.

The fifth and sixth loop invariants were proven to hold initially, upheld by the method's body, and were strong enough to prove the equivalent postconditions. The loop invariants were separately verified, and not verified together due to lack of resources. However, as previously discussed, the set of valid loop invariants is closed under conjunction. Further, the fifth loop invariant had to be verified in parts similarly to the loop invariants of class `Valgdistrikt` in Section 4.5.2. The annotated source-code and proofs are found in [47].

4.5.5 Bugs and Glitches Encountered

Verification efforts rarely conclude without encountering problems. In this section, problems that did not exclusively stem from the verification target nor the specification will be presented. This is done in hopes of achieving three goals:

1. There are occurrences where there are no flaws in the source code of a method nor its specification, and the method conforms to the specification. In such a case, assuming only supported features are utilized, KeY should be able

to prove so. Unfortunately, this is not guaranteed to happen as KeY is not without its flaws. Therefore, it is important that the bugs and glitches that KeY possess are well-documented. If they are, then verifiers are given a new reference to study when no more errors are discovered in the source code or specification. Otherwise, the verifier might loose well-placed faith in his/her verification-effort.

2. The KeY Project is a long-term research project that is continuously evolving; any errors that remain unnoticed might linger long into later versions. As almost any textbook on Software Engineering will state, the longer a bug remains unnoticed, the more resources are required to fix it. Therefore, this section will hopefully bring some of the bugs to light so that they may be fixed.

3. Some of the issues only occurred when utilizing specific types of elements. For instance, throughout the verification effort in this thesis, some Java theories and types were preferred to their corresponding KeY-JML variants, mainly due to how some properties were more difficult (or even impossible) to prove for the KeY-JML variants. One prominent example of this will be displayed later on, where trivial conditions could not be proven for the KeY-JML theory *sequents*. By making others aware of the (dis)advantages of using specific theories, less time will be wasted on re-implementing stubs and re-writing specifications that could have utilized a "safer" type.

Starting of, anyone attempting tool-assisted deductive verification has to be aware of contradictions. When using KeY, the verifier should be aware of branches that seem to be quickly closed by applying the rule *close false*. However not necessarily an error on KeY's part, if the stubs are wrong they can introduce a contradiction which may cause KeY to reduce an expression in the antecedent to false. Consequently, using this stub, all properties can be "proved" as the branch can be closed with the *close false*-rule.

One of the more mysterious issues was encountered while experimenting with the tool, and occurred when changing object types. For instance, the verification target was a method similar to

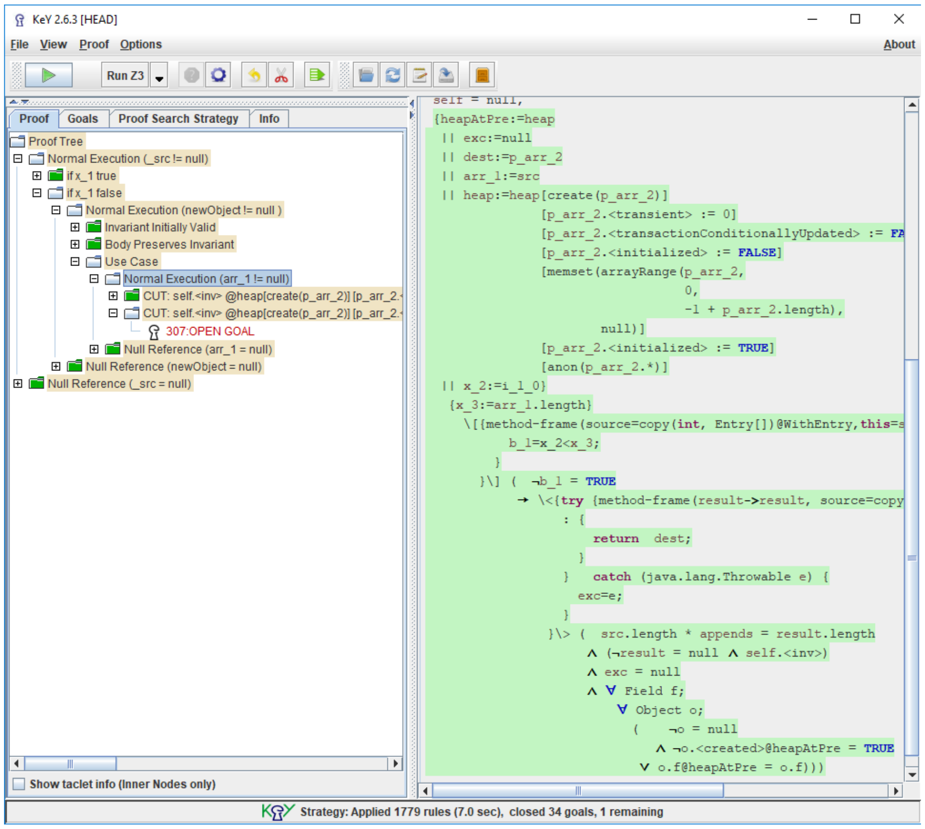
```
public A[] copy(int max, A[] src);
```

which returns an array where the elements `0` to `max` are copied from the respective indexes in `src`. The objects in `src` are not accessed in any way, their pointers are simply copied over from `src` to the result. Some properties were specified, and KeY was able to load the contracts for `copy(int,A[])` and prove the contracts valid. Then, for experimental purposes, all occurrences of type `A` were changed to type `B`. The method signature was then similar to

```
public B[] copy(int max, B[] src);
```

and no other alterations were made. Now, KeY would attempt to load the contracts for `copy(int,B[])`, but would not be able to do so. Instead, it simply did not load anything, nor did it provide any indication that an error had occurred. After a couple more attempts at loading the contracts, KeY would give the option to once again load the contracts for `copy(int,A[])` instead. When loading the contracts, KeY was no longer able to verify these contracts. The issue was resolved by creating a new file with a different filename, and thereby a different classname, and loading that file. KeY was then able to load the contracts for `copy(int,B[])`, and prove them all valid.

Furthermore, when incrementally adding conditions to the method `copy(int max, B[] src)`, KeY would suddenly not be able to verify the same properties twice in a row. Instead, it would be unable to close a branch that was not present in the previous valid proof tree, and the resulting proof tree would look like in Figure 4.2. This bug would seemingly disappear as unexpectedly as it appeared — at times KeY had to be restarted, files reloaded, specifications reverted, or contents moved into new files; at other times, only one of those measures had to be taken.


KeY 2.6.3 [HEAD]
File View Proof Options
About
Run Z3
Proof Goals Proof Search Strategy Info
Show taclet info (Inner Nodes only)

```
Proof Tree
⊟ Normal Execution (_src != null)
  ⊞ if x_1 true
  ⊟ if x_1 false
    ⊟ Normal Execution (newObject != null )
      ⊞ Invariant Initially Valid
      ⊞ Body Preserves Invariant
      ⊟ Use Case
        ⊟ Normal Execution (arr_1 != null)
          ⊞ CUT: self.<inv> @heap[create(p_arr_2)] [p_arr_2.
          ⊟ CUT: self.<inv> @heap[create(p_arr_2)] [p_arr_2.
              307:OPEN GOAL
        ⊞ Null Reference (arr_1 = null)
    ⊞ Null Reference (newObject = null)
⊞ Null Reference (_src = null)
```

```
self = null,
{heapAtPre:=heap
 || exc:=null
 || dest:=p_arr_2
 || arr_1:=src
 || heap:=heap[create(p_arr_2)]
              [p_arr_2.<transient> := 0]
              [p_arr_2.<transactionConditionallyUpdated> := FA
              [p_arr_2.<initialized> := FALSE]
              [memset(arrayRange(p_arr_2,
                                 0,
                                 -1 + p_arr_2.length),
                      null)]
              [p_arr_2.<initialized> := TRUE]
              [anon(p_arr_2.*)]
 || x_2:=i_1_0}
  {x_3:=arr_1.length}
   \[{method-frame(source=copy(int, Entry[])@WithEntry,this=s
         b_1=x_2<x_3;
       }
     }\] (  ¬b_1 = TRUE
          → \<{try {method-frame(result->result, source=copy
                : {
                  return  dest;
                }
              }   catch (java.lang.Throwable e) {
                exc=e;
              }
            }\> (  src.length * appends = result.length
                 ∧ (¬result = null ∧ self.<inv>)
                 ∧ exc = null
                 ∧ ∀ Field f;
                     ∀ Object o;
                       (    ¬o = null
                          ∧ ¬o.<created>@heapAtPre = TRUE
                        ∨ o.f@heapAtPre = o.f)))
```

Strategy: Applied 1779 rules (7.0 sec), closed 34 goals, 1 remaining

Figure 4.2: Uncloseable proof tree deduced from valid contract.

Occasionally, after running the automatic execution, KeY will hang and be unable to display the proof tree. Furthermore, saving the current proof tree or loading other proofs becomes impossible, and the only way to get KeY to respond is to restart the prover. This occurs at specific steps in the process, and usually occurs only on specific uncloseable branches. Unfortunately, as the proof tree could not be displayed, it can be challenging to find which branch KeY hangs on. To find the branch, decrease the number of automatic rule applications, and run the automatic execution in stages. For instance, consider an example where KeY hangs on step 4688 in the automatic execution. It could then be beneficial to set the maximum number of automatic rule applications to 500, and run 10 iterations of automatic execution. Then, when KeY hangs on the 10th iteration, it becomes clear that the issue lies somewhere between step 4500 and 5000. Keep narrowing down until the error-inducing branch becomes apparent, and study the program flow that induces that branch. Then, alter the source code or specification to eliminate the occurrence of that branch. For larger proof trees, where KeY hangs on some step larger than 50k, it can be beneficial to run the automatic execution up until a known safe point. Then, the proof tree may be saved such that the safe point functions as a checkpoint. When narrowing down the possibilities, the checkpoint may be loaded so that time is not wasted on re-running the first 50k steps every iteration. However, keep in mind that if KeY hangs, it might not be able to save the proof tree[17], and if the proof tree is saved, then KeY might throw an `OutOfMemoryError` when loading it[18]. During the verification effort in this thesis, this bug was encountered mainly when failing to satisfy preconditions of methods called inside the verification target. Some of the methods called had no explicit preconditions, i.e., their only `requires`-clause was `requires true`. However, all methods have the implicit precondition `requires \invariant_for(this)`, and, if any parameters are passed to the method, there is an implicit `non_null` requirement for the parameters. Further, constructors have the implicit precondition `\static_invariant_for(C )`, where `C` is the constructor's class, instead of `\invariant_for(this)`[19]. By ensuring that the invariants of objects hold inside the body of the verification target, most cases of this bug can be eliminated.

Specifying loop invariants may at times not be as straight-forward as one might hope; an issue regarding verification of trivial properties was discovered during the verification effort in Section 4.5.3. More specifically, when attempt to create a model that could calculate a given number of divisors, KeY was unable to prove certain trivial conditions. Consider the following method:

```
/*@ public normal_behavior
  @ requires keys.length == values.length;
  @ requires keys.length > 0;
```

[17]When KeY starts hanging, it will not be able to save the proof tree. The resulting proof saved will only contain the proof meta-data, and not the proof tree.

[18]At times this error occurred even when KeY had utilized less than the memory allocated by the Java Virtual Machine

[19]The expression `\invariant_for(this)` should never be used as a precondition for constructors, as `this` does not exists prior to invoking the constructor.

```
  @ requires iterations > 0;
  @ ensures \result.length == keys.length * iterations;
  @ assignable \nothing;
  @*/
public /*@ nullable @*/ PartiKvotient[]
    invalidTrivialCondition(String[] keys, int[] values, int
     iterations){

  int antallKvots = keys.length * iterations;
  PartiKvotient[] res = new PartiKvotient[antallKvots];
  int index = 0;
  /*@ maintaining 0 <= n && n <= iterations;
    @ maintaining res.length >= keys.length;
    @ decreasing iterations - n;
    @ assignable \nothing;
    @*/
  for (int n = 0; n < iterations; n++){
    /*...*/
  }
  return res;
}
```

As `iterations` is bounded to be greater than 0, then `res.length` should, in the minimal case of `iterations = 1`, be *at least* equal to `keys.length`. However, KeY is unable to verify that this is true and fails to close the *Invariant Initially Valid* branch.

The former issue is not the only instance where KeY struggles when proving valid loop invariants. For instance, consider the following formula which states that for all integers between 0 and some limit, the predicate $p(i)$ is true.

$$(\forall\, int\, i\, ;\ 0\ <=\ i\ \wedge\ i\ <\ \text{LIMIT}\, ;\ p(i))$$

Now, if that formula is valid, then the following should also be valid

$$(\forall\, int\, j\, ;\ i\ <\ j\ \wedge\ j\ <\ \text{LIMIT}\, ;\ p(j))$$

as the possible values of j is a subset of the possible values of i. Unfortunately, when similar formulas are applied as loop invariants, such as the latter in Listing 4.41, KeY is unable to automatically verify that the loop invariant holds.

```
// This is valid

/*@ public normal_behavior
  @ requires \invariant_for(kvotienter);
  @ requires
  @  (\forall int i;
  @    0 <= i && i < kvotienter.length;
  @    \invariant_for(kvotienter[i]) &&
  @    kvotienter[i] != null &&
  @    kvotienter[i].getPartikode() != null);
  @
  @ requires kvotienter != null;
  @ requires kvotienter.length > 0;
  @
  @ ensures \result != null;
  @
  @ assignable \nothing;
  @*/
private static TreeMap validTest(PartiKvotient[] kvotienter
    ) {
  TreeMap partiMandatMap = new TreeMap();

  /*@ maintaining 0 <= \index &&
    @             \index <= kvotienter.length;
    @ maintaining partiMandatMap.size() <=
    @             kvotienter.length;
    @ maintaining partiMandatMap.keys.length <= \index;
    @ maintaining \invariant_for(partiMandatMap);
    @ maintaining
    @  (\forall int i; 0 <= i && i < \index;
    @    kvotienter[i] != null &&
    @    kvotienter[i].getPartikode() != null &&
    @    \invariant_for(kvotienter[i]));
    @
    @ maintaining
    @  (\forall int i;
    @    0 <= i && i < partiMandatMap.keys.length;
    @    partiMandatMap.keys[i] != null );
    @
    @ decreasing kvotienter.length - \index;
    @ assignable partiMandatMap.keys,
    @            partiMandatMap.values;
    @*/
  for (PartiKvotient kvotient : kvotienter) {
    String partikode = kvotient.getPartikode();
    if (partiMandatMap.containsKey(partikode)) {
      int antallMandaterForPartiet = partiMandatMap.get(
    partikode);
      partiMandatMap.put(partikode,
    antallMandaterForPartiet + 1);
    } else {
      partiMandatMap.put(partikode, 1);
    }
  }
  return partiMandatMap;
}

// But this is not

```

```
/*@ public normal_behavior
  @ requires \invariant_for(kvotienter);
  @ requires
  @  (\forall int i;
  @    0 <= i && i < kvotienter.length;
  @    \invariant_for(kvotienter[i]) &&
  @    kvotienter[i] != null &&
  @    kvotienter[i].getPartikode() != null);
  @
  @ requires kvotienter != null;
  @ requires kvotienter.length > 0;
  @
  @ ensures \result != null;
  @
  @ assignable \nothing;
  @*/
private static TreeMap invalidTest(PartiKvotient[]
    kvotienter) {
  TreeMap partiMandatMap = new TreeMap();

  /*@ maintaining 0 <= \index &&
    @             \index <= kvotienter.length;
    @ maintaining partiMandatMap.size() <=
    @             kvotienter.length;
    @ maintaining partiMandatMap.keys.length <= \index;
    @ maintaining \invariant_for(partiMandatMap);
    @ maintaining
    @  (\forall int i; 0 <= i && i < \index;
    @    kvotienter[i] != null &&
    @    kvotienter[i].getPartikode() != null &&
    @    \invariant_for(kvotienter[i]));
    @
    @ maintaining
    @  (\forall int i;
    @    0 <= i && i < partiMandatMap.keys.length-1;
    @   partiMandatMap.keys[i] != null &&
    @   (\forall int j;
    @     i < j && j < partiMandatMap.keys.length;
    @     partiMandatMap.keys[j] != null));
    @
    @ decreasing kvotienter.length - \index;
    @ assignable partiMandatMap.keys,
    @            partiMandatMap.values;
    @*/
  for (PartiKvotient kvotient : kvotienter) {
    String partikode = kvotient.getPartikode();
    if (partiMandatMap.containsKey(partikode)) {
      int antallMandaterForPartiet = partiMandatMap.get(
    partikode);
      partiMandatMap.put(partikode,
    antallMandaterForPartiet + 1);
    } else {
      partiMandatMap.put(partikode, 1);
    }
  }
  return partiMandatMap;
}
```

Listing 4.41: Methods with respectivly valid and invalid loop invariants.

On a similar note, KeY seems to struggle proving simple properties of sequents. For instance, given the basic list structure motivated by Chapter 9 of [17]:

```
//@ private instance ghost \seq theList;

/*@ public normal_behavior
  @ ensures theList == \seq_empty;
  @ assignable this.theList;
  @*/
public List();

/*@ public normal_behavior
  @ ensures
  @  theList == \seq_concat(\seq_singleton(elem),
  @                         \old(theList));
  @ assignable this.theList;
  @*/
public void add(String elem);

/* The following is more or less a KeY-JML alias for
 * the postcondition of contains(String elem)
 * and may be used instead.
 *
 * ensures \result ==
 *  (\dl_seqIndexOf(theList, elem) !=
 *   \dl_seqGetOutside());
 */

/*@ public normal_behavior
  @ ensures
  @  \result == (\exists int i;
  @    0 <= i && i < theList.length;
  @    ((String) theList[i]).equals(elem));
  @
  @ assignable \strictly_nothing;
  @*/
public boolean contains(String elem);
```

When utilizing this structure, the following is automatically proven by KeY...

```
/*@ public normal_behavior
  @ ensures \result.contains(param0);
  @ assignable \nothing;
  @*/
public List validListSpec(String param0){
  List list = new List();
  list.add(param0);
  return list;
}
```

...while the following is not:

```
/*@ public normal_behavior
  @ requires !(param0.equals(param1));
  @ ensures \result.contains(param0) ||
  @         \result.contains(param1);
  @ assignable \nothing;
  @*/
public List invalidListSpec(String param0, String param1){
  List list = new List();
  list.add(param0);
  list.add(param1);
  return list;
}
```

5 Conclusion

5.1 Summary and Results

This thesis has presented and demonstrated the capabilities of deductive verification using the specification language JML and the proof assistant of the KeY System, by describing the process behind a verification project from start to finish.

Chapter 2 introduced the most basic elements of JML, and more elements and special constructs of the language were presented and discussed throughout Chapter 4. Further, Chapter 2 introduced the logical language JDL that is used to formalize the proof obligations KeY creates based on the JML specifications. Chapter 3 presented the core ideas and theory behind KeY. Specifically, it was presented how KeY transforms JML specifications into proof obligations, and a tutorial was given on the formalism of taclets that KeY uses to apply calculus rules to proof obligations.

The entirety of Chapter 4 was dedicated to verifying properties of EVA utilizing the specification language JML and the proof assistant of the KeY System. Specifications were created for methods partaking in EVA's seat allocation calculation, and the most central methods were verified to conform to their specifications. All the created specifications and all generated proofs are found in the Git repository *EVA-KeY* [47]. The classes from Section 4.4.1 and Section 4.4.2 were verified with little source-code modifications necessary; the only method that required an alternative implementation was the method `compareTo` in class `PartiKvotient`. The specifications presented in Section 4.4.3 were only partially verified due to problems arising from creating objects from stubs inside nested loops, and due to issues presented in Section 4.5.5. However, an alternate implementation with some verified properties was presented, and a less formal argument was made for the correctness of the method; it is believed that it would be possible, given sufficient time and resources, to prove that all methods in Section 4.4.3 conform to their specifications. Finally, the method for allocating seats in Section 4.4.4 was proven to conform to its specification. Furthermore, several reusable artifacts were created throughout the verification effort in Section 4.5, such as stubs for common Java libraries and loop invariants for several scenarios. Section 4.5.5 concludes the chapter by presenting any unresolved issues encountered during the verification effort.

5.2 Discussion

The previous section summarizes the results of the verification effort, which was the primary focus of this thesis. Although not every method was proven to conform to its specification, the overall verification effort contributes to reaching the goals presented in Chapter 1. Specifically,

- the important modules of EVA have been introduces and formally specified,

- some of the important modules have had some of their properties mathematically proven, which increases the confidence held in the system,
- the verification process and specification has been transparent such that the methodologies can be implemented into other projects,
- the most fundamental and useful tools, such as ghost fields, invariants, behavioral contracts, have been thoroughly introduced and explained,
- stubs for common data structures and library methods, such as `arraycopy` and `sort`, have been generated and presented, and
- the overall verification effort can be used as a motivating example for using the KeY proof assistant in large software systems, such as EVA, that are also used in practice; in our case EVA is the system used to handle the national elections in Norway.

As mentioned in Section 4.3.2, this work was limited to verifying normal behavioral contracts, and disregarded exceptional behavior and dependency contracts. The stubs presented could have been extended with exceptional behavior contracts, as some of the Java library methods they represent throws exceptions for certain parameters. However, the methods are largely shielded from `NullPointerExceptions` as parameters have an implicit non-null precondition by default, and preconditions are in place to shield the methods from other common runtime exceptions such as `IndexOutOfBoundsException` and `ArithmeticException` . Dependency contracts, however, should have received more focus as they are an important aspect of verification. Unfortunately, for generating and refining dependency contracts one needs to allocate considerable amounts of time and effort as dependency contracts are *exceptionally* more time consuming to verify than behavioral contracts [17, 34]. The reader is encouraged to study the verification effort in [34] for an example on how to specify and verify properties regarding dependency.

The thesis has also not focused on *using* the KeY System, and, as the tool is so central to the work presented, a section in Chapter 3 could have been devoted to a tutorial explaining setup, proof-tree traversal, and taclet application. The reader is referred to the KeY Book for a guide with figures and examples [17].

In terms of human resources used for this verification effort we evaluate that somewhere between 6-9 person-months were used. As such, this thesis can also show how far a verification effort can reach with these human resources, assuming that the person doing the verification has intermediate knowledge of Java and formal logic, but not assuming that this person has any former experience with formal verification, JML, KeY, or the verification target.

5.3 Outlook and Future Work

The proposed future work is divided into two categories: improving and continuing the work presented in this thesis, and improving auxiliary tools for easing the verification effort.

As for continuing and improving this work, the suggested areas to focus on would be:

- verifying Java libraries; a program is not completely verified until *all* its code is verified,
- improving the verification effort in Section 4.4.3,
- verifying the specifications and methods in Section 4.4.5, and
- improving the stubs created throughout this work and verify that they conform to the Java libraries they represent.

We believe that the verification efforts resulting from either of these points can be concluded with help of the tools and strategies presented in this thesis.

When it comes to easing the verification effort, the core focus should be on making it easier to verify Java code. JML does well in encapsulating the features of Java, but JML could be extended to reason directly about common Java libraries. In addition, KeY-JML specific keywords such as `seqIndexOf`, `seqPerm` and `array2seq` should be implemented into standard JML.

The KeY System is a part of the continuously evolving KeY Project. In order to make the system easier to use, future versions of KeY could include theories and taclets for common data structures, or where they exists, should be made more prone to automated application. Also, support for the JML keywords `new` and `real` would extend the set of possible verification targets.

Any of these points would require more focused expertise and knowledge than has been presented in this thesis. For instance, the area of verifying floating-point arithmetic spans entire chapter of textbooks [59].

The space above and below this message is intentionally left blank.

Bibliography

[1] Bernt Aardal and Johannes Bergh. "The 2017 Norwegian election". eng. In: *West European Politics* 41.5 (2018), pp. 1208–1216. ISSN: 0140-2382. URL: `http://www.tandfonline.com/doi/abs/10.1080/01402382.2017.1415778`.

[2] Luca Aceto et al. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007. DOI: `10.1017/CBO9780511814105`.

[3] *Atelier B Proof Obligations. Reference Manual.* Version 3.7. CLEARSY. URL: `https://www.atelierb.eu/wp-content/uploads/sites/3/ressources/DOC/english/proof-obligations-reference.pdf` (visited on 01/30/2020).

[4] B. Beckert, V. Klebanov, and B. Weiß. "Dynamic logic for Java". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 10001. Springer Verlag, 2016, pp. 49–106.

[5] Wolfgang Bibel. *Automated theorem proving.* eng. Braunschweig, 1987.

[6] Per Bjesse. "What is formal verification?" eng. In: *ACM SIGDA Newsletter* 35.24 (2005), 1–es. ISSN: 0163-5743.

[7] Alexander Bochman and Dov Gabbay. "Sequential Dynamic Logic". eng. In: *Journal of Logic, Language and Information* 21.3 (2012), pp. 279–298. ISSN: 0925-8531.

[8] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics.* Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. ISBN: 978-3-642-03359-9.

[9] Aaron R Bradley. "Debugging". eng. In: *Programming for Engineers: A Foundational Approach to Learning C and Matlab.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 81–92. ISBN: 9783642233029.

[10] Jl Carson. "Drawing the Lines: Constraints on Partisan Gerrymandering in US Politics". English. In: *Perspectives On Politics* 16.1 (2018), pp. 232–233. ISSN: 1537-5927.

[11] "Checking a Large Routine". eng. In: *Alan Turing: His Work and Impact.* 2013, pp. 455–463. ISBN: 978-0-12-386980-7.

[12] Oracle Corporation. *BigDecimal (Java Platform SE 8).* eng. 1993–2020. 1993. URL: `https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html` (visited on 02/27/2020).

[13] Oracle Corporation. *Lesson: Generics (Updated).* eng. 1993-2018. 1993. URL: `https://docs.oracle.com/javase/tutorial/java/generics/index.html` (visited on 04/07/2020).

[14] Daniel W Cunningham. "Mathematical Induction". eng. In: *A Logical Introduction to Proof*. 2012th ed. New York, NY: Springer New York, 2012, pp. 99–141. ISBN: 9781461436300.

[15] L. De Moura and N. Bjørner. "Z3: An efficient SMT Solver". In: vol. 4963. 2008, pp. 337–340. ISBN: 3540787992.

[16] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". eng. In: *Communications of the ACM* 54.9 (2011), pp. 69–77. ISSN: 00010782.

[17] *Deductive Software Verification – The KeY Book: From Theory to Practice*. eng. Vol. 10001. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. ISBN: 9783319498119.

[18] E.W. Dijkstra. *Notes on structured programming*. English. 2nd ed. EUT report. WSK, Dept. of Mathematics and Computing Science. Technische Hogeschool Eindhoven, 1970.

[19] Electoral Directorate. *EVA er klar for valg 2019!* no. `https://www.valg.no/valg-i-norge/valggjennomforing-i-norge/elektronisk-valgadministrasjonssystem/eva-er-klar-for-valg-2019/`. 2019. (Visited on 06/10/2020).

[20] Robert G. Dixon. "Electoral College Procedure". In: *The Western Political Quarterly* 3.2 (1950), pp. 214–224. ISSN: 00434078. URL: `http://www.jstor.org/stable/443484`.

[21] Gordana Dodig Crnkovic. "Scientific Methods in Computer Science". In: (Dec. 2002), p. 8.

[22] S. Falke and D. Kapur. "When is a formula a loop invariant?" In: vol. 9200. Springer Verlag, 2015, pp. 264–286. ISBN: 9783319231648.

[23] Jean-Christophe Filliâtre. "Deductive software verification". eng. In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), pp. 397–403. ISSN: 1433-2779.

[24] J. Fitzgerald et al. "Industrial deployment of formal methods: Trends and challenges". English. In: *Industrial Deployment of System Engineering Methods*. Vol. 9783642331701. Springer-Verlag Berlin Heidelberg, 2013, pp. 123–143. ISBN: 9783642331701.

[25] Inc. Free Software Foundation. *A GNU Manual. 3.16 Options for Code Generation Conventions.* `https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/`. Version 9.2.0. 1988-2019. Free Software Foundation, Inc., 1988. URL: `https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/` (visited on 03/05/2020).

[26] Ricardo Freitas. "Scientific Research Methods and Computer Science". In: *MAP-I Seminars Workshop* (2009), p. 7.

[27] Daniel Galin. "Software Testing". eng. In: *Software Quality: Concepts and Practice*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2018, pp. 255–317. ISBN: 9781119134497.

[28] Martin Giese. *Lecture 3: LK: Soundness & Completeness.* Lecture given for course IN4070 H19. Presentation published on course web-page. University of Oslo, Sept. 2019. URL: `https://www.uio.no/studier/emner/matnat/ifi/IN3070/h19/undervisningsmateriale/lecture03-print.pdf` (visited on 12/03/2019).

[29] Martin Giese. *Lecture 4: First-order Logic.* Lecture given for course IN4070 H19. Presentation published on course web-page. University of Oslo, Sept. 2019. URL: `https://www.uio.no/studier/emner/matnat/ifi/IN3070/h19/undervisningsmateriale/lecture04-print.pdf` (visited on 11/30/2019).

[30] Martin Giese. *Lecture 5: Soundness & Completeness for 1st-order LK.* Lecture given for course IN4070 H19. Presentation published on course web-page. University of Oslo, Sept. 2019. URL: `https://www.uio.no/studier/emner/matnat/ifi/IN3070/h19/undervisningsmateriale/lecture05-print.pdf` (visited on 12/02/2019).

[31] Martin Giese. *Lecture 6: Unification, Normal Forms.* Lecture given for course IN4070 H19. Presentation published on course web-page. University of Oslo, Sept. 2019. URL: `https://www.uio.no/studier/emner/matnat/ifi/IN3070/h19/undervisningsmateriale/lecture06-print.pdf` (visited on 11/30/2019).

[32] James Gosling et al. *The Java Language Specification, Java SE 8 Edition.* 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X.

[33] Stijn de Gouw, Mattias Ulbrich, and Alexander Weigl. *verifythis-ltc-2020.* `https://github.com/KeYProject/verifythis-ltc-2020`. 2020.

[34] Daniel Grahl and Christoph Scheben. "Functional Verification and Information Flow Analysis of an Electronic Voting System". eng. In: *Deductive Software Verification – The KeY Book: From Theory to Practice.* Vol. 10001. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. ISBN: 9783319498119.

[35] Reiner Hähnle and Marieke Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science: State of the Art and Perspectives.* Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 345–373. ISBN: 978-3-319-91908-9. DOI: `10.1007/978-3-319-91908-9_18`. URL: `https://doi.org/10.1007/978-3-319-91908-9_18`.

[36] *Handbook of Model Checking.* eng. 1st ed. 2018. Cham: Springer International Publishing : Imprint: Springer, 2018. ISBN: 3-319-10575-2.

[37] *Handbook of software quality assurance.* eng. 4th ed. Boston: Artech House, 2008. ISBN: 9781596931862.

[38] David Harel. *Dynamic logic.* eng. Foundations of computing. Cambridge, Mass: MIT Press, 2000. ISBN: 0262082896.

[39] David Harel. *First-order dynamic logic.* eng. Vol. 68. Lecture notes in computer science. Berlin: Springer, 1979. ISBN: 3540092374.

[40] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. eng. In: *Communications of the ACM* 12.10 (1969), pp. 576–580. ISSN: 00010782.

[41] Chris Hobbs. “Static Analysis”. eng. In: *Embedded Software Development for Safety-Critical Systems*. 1st ed. CRC Press, 2016, pp. 263–276. ISBN: 9781498726702.

[42] Ranjit Jhala et al. “Formal Methods: Future Directions & Transition To Practice. Workshop Report”. In: *National Science Foundation Workshop*. Dec. 2012.

[43] Ministry of Justice and Public Security. *The Constitution of the Kingdom of Norway*. May 17, 1814. URL: `https://lovdata.no/dokument/NLE/lov/1814-05-17` (visited on 01/21/2020).

[44] David Karol. “US Presidential Election 2016”. eng. In: *Political Insight* 6.2 (2015), pp. 24–27. ISSN: 2041-9058.

[45] Mohamad Kassab, Joanna DeFranco, and Phillip Laplante. “Software Testing Practices in Industry: The State of the Practice”. In: *IEEE Software* PP (May 2016). DOI: `10.1109/MS.2016.87`.

[46] Gerwin Klein et al. “seL4: formal verification of an OS kernel”. eng. In: *Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles*. SOSP ’09. ACM, 2009, pp. 207–220. ISBN: 9781605587523.

[47] Henrik Torland Klev. *EVA-KeY*. `https://github.uio.no/henriktk/EVA-KeY`. 2020.

[48] R. Küsters, T. Truderung, and A. Vogt. “Verifiability, privacy, and coercion-resistance: New insights from a case study”. In: 2011, pp. 538–553. ISBN: 9780769544021.

[49] Gary T. Leavens and Yoonsik Cheon. *The JML and JUnit Way of Unit Testing and its Implementation*. Feb. 2004.

[50] Gary T. Leavens et al. *JML Reference Manual*. Version Draft Revision 2344. May 31, 2013. URL: `http://www.eecs.ucf.edu/~leavens/JML/OldReleases/jmlrefman.pdf` (visited on 02/10/2020).

[51] M. Li. “A practical loop invariant generation approach based on random testing, constraint solving and verification”. In: vol. 7635. 2012, pp. 447–461. ISBN: 9783642342806.

[52] Ministry of Local Government and Modernization. *Act relating to parliamentary and local government elections (Election Act)*. June 28, 2002. URL: `https://lovdata.no/dokument/NL/lov/2002-06-28-57` (visited on 01/21/2020).

[53] Ministry of Local Government and Modernization. *Systemdokumentasjon og kildekode i EVA*. no. `https://www.valg.no/valg-i-norge/valggjennomforing-i-norge/elektronisk-valgadministrasjonssystem/systemdokumentasjon-og-kildekode-i-eva/`. Jan. 10, 2019.

[54] Zohar Manna and Amir Pnueli. “Axiomatic approach to total correctness of programs”. eng. In: *Acta Informatica* 3.3 (1974), pp. 243–263. ISSN: 0001-5903.

[55] Jia Meng and Lawrence Paulson. “Translating Higher-Order Clauses to First-Order Clauses”. eng. In: *Journal of Automated Reasoning* 40.1 (2008), pp. 35–60. ISSN: 0168-7433.

[56] Bertrand Meyer. *Object-oriented software construction.* eng. Upper Saddle River, N.J, 1997.

[57] Jon Meyer. *Java Virtual Machine.* eng. The Java series. Cambridge: O’Reilly, 1997. ISBN: 1565921941.

[58] David Monniaux. “The pitfalls of verifying floating-point computations”. eng. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.3 (2008), pp. 1–41. ISSN: 01640925.

[59] Jean-Michel Muller. *Handbook of Floating-Point Arithmetic.* eng. Cham, 2018.

[60] Tobias Nipkow. *Isabelle/HOL : a proof assistant for higher-order logic.* eng. Berlin, 2002.

[61] C. Paulin-Mohring. “Introduction to the Coq proof-assistant for practical software verification”. In: vol. 7682. 2012, pp. 45–95. ISBN: 9783642357459.

[62] Inc. Red Hat. *Hibernate JavaDoc.* `https://docs.jboss.org/hibernate/orm/5.4/javadocs/`. Version 5.4.12.Final. 2001–2020. Red Hat, Inc., 2001. URL: `https://docs.jboss.org/hibernate/orm/5.4/javadocs/` (visited on 03/05/2020).

[63] Peter Robison. “Boeing’s 737 Max Software Outsourced to $9-an-Hour Engineers”. In: *Bloomberg* (June 28, 2019). URL: `https://www.bloomberg.com/news/articles/2019-06-28/boeing-s-737-max-software-outsourced-to-9-an-hour-engineers` (visited on 01/31/2020).

[64] Nitin Sharma. `https://www.tutorialspoint.com/difference-between-an-integer-and-int-in-java`. Sept. 16, 2019. URL: `https://www.tutorialspoint.com/difference-between-an-integer-and-int-in-java` (visited on 06/14/2020).

[65] Th Skolem. *Mathematical Interpretation of Formal Systems.* eng. Burlington, 2000.

[66] SoftwareTestingHelp.com. *Software Development And Testing Methodologies (With Pros And Cons).* `https://www.softwaretestinghelp.com/software-development-testing-methodologies/`. Nov. 10, 2019. URL: `https://www.softwaretestinghelp.com/software-development-testing-methodologies/` (visited on 03/11/2020).

[67] Priscilla Lewis Southwell. “The electoral consequences of alienation: Nonvoting and protest voting in the 1992 presidential race”. In: *The Social Science Journal* 35.1 (1998), pp. 44–51. DOI: `10.1016/S0362-3319(98)90058-1`. eprint: `https://doi.org/10.1016/S0362-3319(98)90058-1`. URL: `https://doi.org/10.1016/S0362-3319(98)90058-1`.

[68] Michael Stueben. “Defensive Programming”. eng. In: *Good Habits for Great Coding : Improving Programming Skills with Examples in Python*. Berkeley, CA: Apress : Imprint: Apress, 2018. Chap. 11, pp. 123–126. ISBN: 1-4842-3459-6.

[69] “The Norwegian Electoral System and its Political Consequences”. In: *World political science review.* 7.1 (2011). ISSN: 1935-6226.

[70] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: `10.1112/plms/s2-42.1.230`. eprint: `https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf`. URL: `https://doi.org/10.1112/plms/s2-42.1.230`.

[71] Iowa State University. *JML Implementation Documentation.* 1998–2002. 1998. URL: `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/overview-summary.html` (visited on 05/30/2020).

[72] Willem Visser et al. “Model Checking Programs”. eng. In: *Automated Software Engineering* 10.2 (2003), pp. 203–232. ISSN: 0928-8910.

[73] Phillip Webb et al. *Spring Boot Reference Documentation.* eng. Version 2.2.4.RELEASE. 2012–2020. 2012. URL: `https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/` (visited on 02/24/2020).

[74] “Wrong winner: the coming debacle in the electoral college”. eng. In: *Choice Reviews Online* 29.03 (1991), pp. 29-1766–29-1766. ISSN: 0009-4978.

[75] John Zukowski. “Using Enhanced For-Loops with Your Classes”. In: (Sept. 18, 2007). (Visited on 06/12/2020).

[76] Reinier Zwitserloot. *Project Lombok.* `https://github.com/rzwitserloot/lombok`. 2009–2020. 2009. URL: `https://github.com/rzwitserloot/lombok` (visited on 02/24/2020).