PhD Thesis: Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms

Dermot Cochran, IT University of Copenhagen, Denmark

10 September 2012

Contents

List of Figures

List of Tables

Acknowledgements

Thanks I am grateful to my father, Robert Cochran, for his comments on my draft dissertation, based on his knowledge of electronic voting in Ireland. My interest in voting systems began as a teenager when we used to watch the election results unfold on Irish television during the various rounds of counting.

I would also like to thank Assistant Professor Daniel M. Zimmerman and my PhD colleague Josu Martinez for their helpful comments on my research papers.

My mid-term evaluation committee at IT University, consisted of Professor Søren Lauesen (chair), Associate Professor Carsten Schürmann, and Associate Professor Thomas Hildebrand; they also had helpful comments on the direction of my research.

I am also appreciative that my employer Siemens A/S provided me with enough flexibility and support to allow me to finish my dissertation on time.

Supervision This dissertation was supervised by Professor Joseph R. Kiniry both at the IT University of Copenhagen, Denmark and at University College Dublin, Ireland. His enthusiasm and insightful advice was invaluable to my progress as a doctoral research student.

The early stages of my research were co-funded by the LERO Graduate School of Software Engineering at University College Dublin.

Assessment Committee I would like to thank my assessment committee for their helpful comments on my preliminary dissertation and at my defense. Much of their feedback was incorporated into the final draft of this dissertation and the remainder will be addressed in a forthcoming journal paper.

- Professor Michael Butler, University of Southampton
- Associate Professor, Carsten Schürmann, IT University of Copenhagen, (chair)
- Associate Professor, Dan Wallach, Rice University

Dedication

This dissertation is dedicated to my wife Margaret, who faithfully supported me through the long and difficult years of doctoral research, and without her none of this would have been possible. She encouraged me throughout the entire process from the initial decision to pursue the PhD, the move from Dublin to Copenhagen and giving me time and space for putting the finishing touches on my write-up.

Abstract There are many valid arguments both for and against the use of electronic voting in real world elections.

My thesis is that a verification-centric software implementation of a particular algorithm for counting of ballots can be proven correct or shown to be incorrect by a combination of formal specification, static analysis and testing. Hand counting of paper ballots might no longer be necessary, if we can assume that the digital ballots are valid, untampered with and a true representation of the paper ballots etc.

As a case study, the Danish and Irish voting schemes are analyzed in this dissertation, including a discussion of how to generate test cases for the Irish voting scheme.

Chapter 1

Introduction

1.1 Background

In a democratic society, the will of the people is expressed either directly through referenda or indirectly through the election of public representatives and other public officials.

1.1.1 The Secret Ballot

To avoid coercion or intimidation of voters it is common practice to require that ballots remain secret, so that an individual voter cannot reveal how he or she voted.

The use of printed paper ballots instead of handwritten votes to preserve secrecy is sometimes known as the Australian Ballot [29, 35].

However the integrity of a paper ballot still depends on physical security controls. Historically, failed security controls have led to modified, spoiled, and stolen ballots, as well as to stuffed ballot boxes [9].

There is also a potential for mistakes in the manual process of counting paper

ballots by hand. Goggin and Bryne have observed that there can be up to two percent inaccuracy [34, 33].

1.1.2 Electronic Voting Machines

There are many good arguments in favor of electronic voting. It avoids the chain-of-custody problems of handling physical ballot papers and the human error associated with manual counting. It also defeats the problem of ballot papers being stolen or false ballot papers added, so-called *ballot stuffing* [67].

Many countries have used electronic voting with apparent success, others have failed completely and some have continuing problems [36, 48, 60, 26, 50]

Voting and Elections in Ireland

Electronic voting machines are no longer used in the Republic of Ireland, for example. The decision to stop using e-voting was based on several different factors, including an inability to verify the accuracy of the proprietary vote counting software [37, 22, 51].

1.1.3 Internet Voting

Remote voting through the Internet provides convenience and access to the electorate. At the same time, the security concerns facing any distributed application are magnified when the task is so crucial to democratic society. In addition, some of the electoral process loses transparency when it is encapsulated in information technology [66, 42, 61].

Man in the Middle (MITM) attack

Internet voting is also vulnerable to the Man-in-the-Middle (MITM) attack, in which the attacker impersonates the client machine to the server and the server machine to the client. This might be avoided, for example, by physical distribution of the cryptographic certificates to each voter [56, 27, 7].

1.1.4 Cryptographic Voting

Various researchers have suggested cryptographic schemes for secure electronic voting [43, 10, 4, 64]. These technologies are complementary to, but outside the scope of my dissertation.

Sandler, Derr and Wallach have proposed a solution called *VoteBox*, that provides strong end-to-end security guarantees without use of a *paper* audit trail but instead using a concept called the *Auditorium* which

> joins all voting machines together all election events are signed and broadcast each broadcast is logged by every machine

and timestamps are connected using *hash-chaining*[65].

In contrast, my dissertation uses the Irish and Danish voting schemes as a case study. For the purpose of my dissertation, I simply assume that there exists a suitable technology similar to the *Auditorium* in *VoteBox*.

1.2 Motivation

Accidentally or maliciously altering a small number of votes can alter the outcome of an election. Every vote needs to be counted. This means that even a very

small error can have wide ranging consequences, undermining confidence in the democratic process and weakening the moral authority of elected leaders [59].

1.3 Thesis Statement

The following is the core statement of my thesis:

A verification-centric software implementation of a particular algorithm for counting of ballots can be proven correct or shown to be incorrect by a combination of formal specification, static analysis and testing. Hand counting of paper ballots might no longer be necessary, if we can assume that the digital ballots are valid, untampered with and a true representation of the paper ballots.

In other words, we might not always need a voter verifiable paper audit trail (VVPAT), to verify the counting, although it might be needed for other reasons e.g. to verify the collection of votes; and that the votes cast are the same as those counted, but not to verify the actual count itself. The VVPAT might be needed for matching up and verifying the authenticity of the ballots, but not for verification of the counting. The auditors should be able to choose an electronic ballot at random, and match it to a paper ballot in the audit trail, but there is no need to count the audit trail by hand [70, 62].

Chapter 2

Formalisation of Electoral Law

A voting scheme is an algorithm for counting of ballots. A *preference voting scheme* allows the voter to rank two or more candidates ($\mathcal{C}$) in order of preference from first to last. In contrast, a *plurality voting scheme* requires the voter to pick one candidate, and thus is equivalent to the preference scheme when only the first preference is used.

2.1 Single Transferable Vote (PR-STV)

Definition 1 (Spoilt Ballot) *A ballot is spoilt if it contains no valid preferences. A paper ballot can also be spoilt if it contains writing that could identify the person casting the vote*[1].

[1] However, in practice, where the difference between two candidates is very small, ballots initially discarded can be looked at again (often with representatives and even lawyers for each candidate present!). If there is agreement that the intention is clear, even if the ballot paper is strictly invalid, then it may be accepted. So for example, if the name of one candidate is underlined, and no mark against any other name and no numeric preference shown, that might be regarded as an effective No 1 vote. Also an X or a tick against one name only is usually accepted as a No 1 vote. This scenario was not included in my model of PR-STV

Definition 2 (Quota) *The* quota *of ballots for election is calculated so that at most N candidates can reach the quota, where N is the number of seats in the constituency. The formula for the* Droop Quota *is $1 + V/(N + 1)$, where V is the number of unspoiled ballots cast.*

Definition 3 (Surplus) *The* surplus *for a candidate is the number of votes in excess of the quota. Surplus ballots may be transfered to other candidates.*

Definition 4 (Exclusion) *The continuing candidate with least votes is eligible for* exclusion*; his or her ballots are then available for re-distribution to other continuing candidates.*

Definition 5 (Continuing Candidate) *A* continuing candidate *is a candidate who has not yet been either* elected *or* eliminated*.*

Definition 6 (Wasted Vote) *A ballot with an incomplete set of preferences becomes wasted, if and when it is assigned to an* excluded *candidate or within the surplus of an* elected *candidate, and cannot be transferred to any of the* continuing *candidates.*

Filling of Last Seat When there is one seat remaining and two continuing candidates, then the candidate with the greater number of votes is deemed elected without reaching the quota.

Variants of PR-STV To highlight the complexities of election schemes, consider the following variants of the voting scheme. For example, Australia, Ireland, Malta, Scotland, and Massachusetts use different variants of PR-STV for their elections [6].

- **Australia** - Australia uses a single-seat variant of PR-STV to elect its House of Representatives and an open list system for its Senate, where voters can choose either to vote for individual candidates using PR-STV or to vote "above-the-line" for a party. If voters choose to use PR-STV then all available preferences must be used [28].

- **Ireland** - Ireland uses PR-STV for local, national and European elections. Transfers are rounded to the nearest whole ballot, so the order in which ballots are transferred makes a difference to the result [54]. Not all preferences need to be used, so voters may choose to use only one preference, as in Plurality voting, if desired.

- **Malta** - Malta uses PR-STV for local, national and European elections. For national elections Malta also adds additional members so that the party with the most first preference votes is guaranteed a majority of seats.

- **Scotland, UK** - Scotland uses PR-STV for local elections. Rather than select which ballots to include in the surplus, fractions of each ballot are transferred [32].

- **Massachusetts, USA** - Cambridge in Massachusetts uses PR-STV for city elections. Candidates with less than fifty votes are excluded in the first round and surplus ballots are chosen randomly, rather than in proportion to second and subsequent preferences.

The fact that a single complex voting scheme like PR-STV has this many variants in use highlights the challenges in reasoning about and validating a given software implementation.

2.2 Abstract Model

In this case study, the core concepts of ballot counting are modeled, for example: *ballots*, *ballot boxes*, *candidates*, and *election results*.

Candidates *Candidates* are identified by (distinct) names. The set of all candidates is denoted $\mathcal{C}$. These examples use canonical candidate names, e.g. Alice, Bob, etc., like those used in cryptographic protocol analysis.

Ballot An ordinal or preference ballot b is a strict total order on a set of candidates $\mathcal{C}$. The length of a ballot, $|b|$, is the number of preferences expressed. The minimum number of preferences is one, except in systems like that used in Australia where all preferences must be used. In a plurality voting scheme the maximum number of preferences is one, otherwise the maximum length of a ballot is the number of candidates in the election, or at fixed limit determined by electoral law.

2.3 Irish Electoral Law

Article 16, section 2, subsection 5 of the Irish Constitution[2] states only that:

> The members shall be elected on the system of proportional representation by means of the single transferable vote.

This leaves considerable freedom for the electoral regulations to define the precise means of vote casting and counting in the official Commentary on Count Rules [23, 21].

[2] `http://www.constitution.ie/constitution-of-ireland/default.asp`

The functional requirements were read from the electoral legislation and CEV guidelines and listed in a table by Cochran and Kiniry [14, 46] e.g.

- Counting does not begin until all votes are loaded.
- The total number of first preference votes must remain the same after each count.
- The (Droop) quota is equal to ((Number of valid votes cast)/(Number of seats being filled + 1)) + 1, ignoring any remainder.
- Any candidate with a quota or more than a quota of votes, is deemed to be elected.
- The surplus for an elected candidate is the difference between the quota and his/her total number of votes.
- The minimum number of votes required for a candidate to secure the return of his/her deposit is one plus one-quarter of a quota based on the total number of seats in the constituency.

2.4 Mechanics of PR-STV

To give context, we now discuss the mechanics of PR-STV in more detail.

Preference Ballots The voter writes the number "1" beside his or her favorite candidate. There can only be one first preference.

The voter then considers which candidate would be his or her next preference if his or her favorite candidate is either excluded from the election or is elected

with a surplus of votes.

The second preference is marked with "2" or some equivalent notation. There can be only one second preference; there cannot be a joint second preference[3]. Likewise for third and subsequent preferences. Not all preferences need to be used, but there should be no gaps in the rank order e.g. there cannot be a fourth preference without a third preference.

Multi-seat constituencies Each electoral constituency in Dáil Éireann , the lower house of the Irish state legislature, is represented by either three, four or five members.

Definition 7 (The Droop Quota) *The quota is calculated so that not all candidates can reach the quota. The Droop Quota is* $\lceil \frac{V}{1+S} \rceil$*, where* V *is the total number of valid votes cast and* S *is the number of vacancies (or seats) to be filled [30]. The quota is chosen so that any candidate reaching the quota is automatically elected, and so that the number of candidates with a quota is less than or equal to the number of seats.*

For example, in a five-seat constituency a candidate needs just over one-sixth of the total vote to be assured of election.

Definition 8 (Surplus) *The surplus for each candidate, is the number of ballots in excess of the quota (if any). The surplus ballots are then available for redistribution to other continuing candidates.*

The selection of which ballots belong to the surplus is a complex issue, depending on the round of counting. In the first round of counting, any surplus is divided

[3]This is because a physical ballot paper is not intended to be split in two, but in a digital voting system with fractional ballots it would be possible to have equality of preferences

into sub-piles for each second preference, so that the distribution of the ballots in the surplus is proportional to the second-preferences. In later rounds the surplus is taken from the last parcel of ballots received from other candidates. This surplus is then sorted into sub-piles according to the next available preference.

For example, if the quota is 9,000 votes and candidate A receives 10,000 first preference votes, then the surplus is 1,000 votes. Suppose 5,000 ballots had candidate B as next preference, 3,000 had candidate C and 2,000 had candidate D. Then the surplus consists of 500 ballots taken from the 5,000 for candidate B, 300 from the 3,000 for candidate C and 200 from the 2,000 for candidate D[4].

Exclusion of weakest candidates When there are more candidates than available seats, and all surplus votes have been distributed, the continuing candidate with least votes is excluded.

All ballots from the pile of the excluded candidate are then transferred to the next preference for a continuing candidate, or to the pile of non-transferable votes.

This continues until another candidate is elected with a surplus or until the number of continuing candidates equals the number of remaining seats.

Filling of Last Seat and Bye-elections When there is only one seat remaining to be filled, i.e. the number of candidates having so far reached the quota is one less than the number of seats, or in a bye-election for a single vacancy, then the algorithm becomes the same as *Instant Runoff Voting*; no more surplus distributions are possible, and candidates with least votes are excluded until only two remain.

[4]Ideally each subset would also be sorted according to third and subsequent preferences, but this does not happen under the current procedure for counting by hand, nor was it mandated in the previous guidelines for electronic voting in Ireland

Last Two Continuing Candidates When there are two continuing candidates and one remaining seat, then the algorithm becomes the same as single-winner First Past The Post (FPTP) plurality; the candidate with more votes is deemed elected to the remaining seat, without needing to reach the quota.

In the next chapter I will discuss some of the various technologies that can be used in verification, and then in chapter 4, I will discuss existing examples of verification in voting, before talking about Danish and Irish voting in the following chapters.

Chapter 3

Verification Technologies

This chapter describes and compares an assortment of verification technologies.

3.1 Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language that can be used to specify the behavior of Java modules. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus [8, 13].

3.2 Extended Static Checking

Extended Static Checking (ESC) is the transformation of a program into a set of verification conditions, which can then be checked using an automated theorem prover.

ESC/Java2 is a programming tool that attempts to partially verify JML annotated Java programs by static analysis of the program code and its formal annotations. It translates the specifications into verification conditions that are modularly discharged by an automatic theorem prover [47].

3.3 Alloy

Alloy is a textual, declarative modeling language based on first-order relational logic. An Alloy model consists of Signatures, Relations, Facts and Predicates. Signatures represent the entities of a system and Relations depict the relations between such entities. Facts and Predicates specify constraints, which apply on the Signatures and Relations [39].

Alloy comes with a tool, the Alloy Analyzer, which supports fully automated analysis of Alloy models. The analyser provides two main functionalities, Simulation and Assertion checking. Simulation produces a random instance of the model, which conforms to the specification. This ensures that the developed model is not inconsistent. Assertions are constraints, which the model needs to satisfy.

The Alloy Analyzer works by translating Alloy formulas to boolean expressions, which are analysed by SAT solvers embedded within the analyser. A user-specified scope on the model elements bounds the domain. If an instance that violates an assertion is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope.

An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network. The Alloy Analyzer,

is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand.

At its core, the Alloy language is a simple but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski relational calculus. Alloy syntax is designed to make it easy to build models incrementally, and was influenced by modeling languages (such as the object models of OMT and UML). Novel features of Alloy include a rich subtype facility for factoring out common features and a uniform and powerful syntax for navigation expressions.

Alloy is influenced by Z, but has a syntax that mimics object-oriented languages, making it accessible to Java developers.

3.4 SAL

SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is an intermediate language for describing transition systems. This language is intended to serve as the target for translators that extract the transition system description for other modeling and programming languages, and as a common source for driving different analysis tools[1].

[1] http://sal.csl.sri.com/

3.5 ASM Tools

Gurevich's Abstract State Machines (ASMs) constitute a high-level state-based modeling language, which has been used in a wide range of applications. The ASM Workbench is a comprehensive tool environment supporting the development and computer-aided analysis and validation of ASM models. It is based on a typed version of the ASM language, called ASM-SL, and includes features for type-checking, simulation, debugging, and verification of ASM models [19].

3.6 PVS

PVS is a prototype verification system: that is, a specification language integrated with support tools and a theorem prover. It is intended to capture the state-of-the-art in mechanized formal methods and to be sufficiently rugged that it can be used for significant applications. PVS is a research prototype: it evolves and improves as we develop or apply new capabilities, and as the stress of real use exposes new requirements [58, 57].

3.7 Event-B and Pro-B

Event-B[2] is a formal method for system-level modeling and analysis. Key features of Event-B are the use of set theory as a modeling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels [1, 63, 68, 2].

[2] `http://www.event-b.org/`

Pro-B offers very similar functionality to the Alloy Analyzer; it can generate counterexamples to assertions fully automatically.

3.8 KeY Tool

KeY is a tool that provides facilities for formal specification and verification of programs within a commercial platform for UML based software development. Using the KeY tool, formal methods and object-oriented development techniques are applied in an integrated manner. Formal specification is performed using the Object Constraint Language (OCL), which is part of the UML standard [3].

3.9 Analysis and Comparison

My criteria for assessing which tools, language and methodology to use for analysis of ballot counting are:

1. Ability to express first-order logic
2. Object-orientedness
3. Java support

which led to the selection of BON, JML and Alloy as design, specification and modeling languages, respectively, with the most emphasis on JML and Alloy, and with a lessor emphasis on BON. In particular, I did not use the more formal part of BON since that can be derived directly from the JML, and vice versa.

Chapter 4

Verification in Voting

This chapter will discuss the use of verification technology in existing voting systems e.g. KOA [46]. KOA is interesting because of the use of JML to specify its tally system.

4.1 Waterford Institute of Technology

Meagher used Z and B to model the PR-STV based election process at Waterford Institute of Technology [55]. The election process was a variant of PR-STV with gender and faculty limits. On reaching the quota for election, a candidate who exceeded one of the limits would be excluded instead of deemed elected, and all of his or her votes would be made available for transfer. Likewise, in some cases the least continuing candidate is exempt from exclusion if he or she is needed to fulfill one of the lower bounds, and therefore the second lowest candidate is perhaps excluded instead. In other words the WIT election scheme is PR-STV with special rules for exclusion or non-exclusion of candidates.

4.2 Voting Machines in the Netherlands

Electronic voting machines (EVMs) were introduced in the Netherlands around 1998[1]. The primary supplier of these machines is Nedap[2], the same supplier as in Ireland.

However, as attention has been focused the world over on EVMs, the Dutch parliament had begun to re-evaluate its approach. In keeping with this reassessment, the Dutch parliament decided to conduct experiments with the next natural step in the use of technology for voting: remote voting using both the internet and telephone.

4.2.1 Kiezen op Afstand (KOA)

The genesis of KOA stemmed from a promise made by the Dutch government to parliament that they would investigate possible developments in the Dutch voting system. This promise was fulfilled in the KOA experiment by allowing expatriates to vote in the elections to the European Parliament via the internet and by telephone.

However, Dutch national election law is quite explicit about what is permitted with respect to how votes may be cast. Therefore, in order to conduct an experiment in voting over the internet, some amendments to this general law were formulated. This formed the legal foundation for the KOA project.

Apart from the general rules governing internet voting, it also included some additional rules detailing a citizen's right to vote from a different polling booth other than the one originally appointed. However, in this chapter the KOA project

[1] `http://wijvertrouwenstemcomputersneit.nl/English`
[2] Nedap — `http://www.nedap.com/`.

is described purely as if it was an internet voting experiment.

Internet Voting in The Netherlands The elections to the European Parliament of June 2004 allowed remote voting via the internet and telephone. It was limited to expatriates who were required to explicitly register beforehand. It was thought that such a small-scale use (thousands of voters) would provide a useful real-world test for the technology.

The main reason why it was thought that an internet-based solution was suitable is decidedly non-technical. Essentially, by significantly constraining the remote voting problem, particularly with respect to the registration and voting process itself, it was believed that a "sufficiently secure" and reliable system could be constructed. In particular, the system needed to be at least as secure and reliable as the existing remote voting system which was based upon postal ballots.

The Remote Voting Process When a citizen registers to use KOA, the voter chooses their own personal access code (a PIN). Some time later, a customized information packet is mailed to the voter. This packet contains general information about the vote itself (date, time, etc.), as well as voter-customized details that are known to only that voter. These details include information for voter authentication, including an identification code and the previously chosen access code.

Also included is a list of all candidates. Each candidate is assigned a large set of unique random numbers[3], and exactly one of those numbers is given to each voter. The set of codes per voter is determined randomly but is not unique.

To vote, a registered voter logs in to a web site with their voter code and access

[3] 1,000 codes were generated for each candidate for the elections to the European Parliament in 2004.

code. They then step through a series of simple web pages, typing in their candidate codes as appropriate for their choices. The system shows the voter the actual names and parties of the candidates in question to confirm the accuracy of the vote. When a voter is finished, a transaction code is provided. This code can later be used to check in a published list that the voter's choices were included correctly in the final tally.

All votes are stored in a doubly-encrypted fashion; each vote is encrypted by a symmetric key per voter[4] and the public key of the voting authority.

Use and GPL Release The trial during the elections to the European Parliament in June, 2004 was restricted to roughly 16,000 eligible Dutch expatriates. Expatriates could vote either via the internet or by telephone. The telephone votes were fed into the KOA tally system. 5,351 people used one or other system.

Subsequently, in July 2004, the Dutch Government released the majority of the source code for the KOA system under the GNU General Public License (GPL) making it the first Open Source internet voting system in the world.

Vote Counting System As seen in the previous section, one of the results of the recommendation to split the responsibilities of the parties involved, was that the government decided to accept bids for the creation of a separate vote counting subsystem, to be implemented in isolation by a third party. This separate tally application would allow the vote counting to be independently verified. The Security of Systems (SoS) (now called Digital Security) group at Radboud University Nijmegen[5] put forward a proposal to write this application, and were successful

[4] This symmetric key is generated by hashing the assigned identification code.

[5] `http://www.ru.nl/ds/`

in this bid. The key idea behind their tender was that the vote counting program should be formally verified using the JML and ESC/Java2 tools.

The vote counting system formed a small but important part of the whole KOA system. This provided the SoS group with a suitable opportunity to test the use of some of the formal techniques and practices that they had been developing. Given the severe time constraints placed upon them due to the impending election, the application was built by three members of the group over a barely-sufficient period of four weeks. Java was chosen as the programming language in which to implement the system so that JML could be used as the formal specification language. Due to the time constraints, verification was only attempted with the core modules.

Counting votes within KOA proceeds offline using a separate tally application. The input to this application consists of two XML files (one containing the list of candidates and their codes, and one containing the encrypted votes), and a public/private key pair used to decrypt the votes.

As the informal requirements of vote-counting are obvious (for every candidate in the candidate list count the number of votes for that candidate), the functional specification [53] (in Dutch) mostly prescribes details of file formats and encryption algorithms to be used.

Nevertheless, the functional specification does impose some requirements that greatly influence the structure of the Java application and its JML specification. First, the different tasks that need to be performed in order to count the votes (reading in the two files, reading in the keys, decrypting the contents of the votes file, counting the votes, generating reports) are made explicit in this document and, more importantly, the order in which they have to be performed is specified. Sec-

ond, the document provides a rough sketch of the user interface and its contents. Finally, the document gives some bounds on the data, such as the lengths of fields or the maximum number of candidates in each list, which are incorporated in the JML specifications of the data structures.

In accordance with the above high-level specification, the resulting tally application consists of some 30 classes, which can be grouped into three categories: the data structures, the user interface, and the tasks.

The data structure classes form an excellent opportunity to write JML specifications. Typical concepts from the domain of voting, such as candidate, district and municipality can be modeled with detailed JML specifications.

The different tasks associated with counting votes were mapped to individual classes. After successful completion of a task, the application state is changed. A task can only be started if the application is in an appropriate state. The life-cycle model of the application that therefore emerges is maintained in the main class of the application inside a simple integral field. This life-cycle model can be specified in JML using invariants and constraints, essentially stating that on successful completion of the application, the application went from “initial state” to “votes counted state”. The state of attributes associated with the individual tasks can be linked to the application life-cycle state using invariants. For instance, such an invariant could read: ‘after the application reaches the “keys imported state”, the private key field is no longer null’. This is stated in `MenuPanel.java` as follows:

```
/*@ invariant
  @    (state >= PRIVATE_KEY_IMPORTED_STATE
```

```
  @     implies privateKey != null);
 */
```

A graphical user interface is usually not very amenable to formal specification. Nonetheless, some light-weight specifications were written. One of the requirements defined in the original informal specification was that users should not be allowed to start certain tasks before certain other tasks are successfully completed. For instance, a user should (by means of the user interface) not be able to start decrypting votes before the votes are read in from file. In the graphical user interface, this demand is met by only enabling certain buttons when the application reaches certain states in the life-cycle model. The fact that the graphical user interface complies with the life-cycle model can be neatly specified in the GUI classes by referring to the application state.

Process As already stated, ESC/Java2 was only used to verify the core of the tally application. This means that it was used to verify reading in the XML-files with the candidates and the votes, decryption of the votes and counting the votes. The final generation of the reports is not checked with JML.

Using JML on reading XML files is quite straightforward. Essentially, for every object that is read, some methods are called that specify that the total number of objects will be increased by exactly one. Naturally, in order to verify code that uses functionality provided in external libraries, some of the corresponding APIs must also be specified. The JML community has provided specifications for most of the APIs that come with Sun's standard edition of Java. However, APIs dealing with cryptography, XML parsing, and PDF generation, as used by the tally application had not previously been specified. These APIs were specified in a light-

weight manner: the specifications mostly deal with purity and non-null references in the API methods which makes verification of client code using ESC/Java2 much easier.

Naturally, the counting process is likewise formally specified in JML, which ensures that each valid vote is counted for exactly one candidate. This also implies that specifications are easy to check to make sure that the total number of votes a party list receives is equal to the sum of votes for each candidate[6] on this party list.

The JML run-time assertion checker was also used in the development process. First, for testing the data structure classes, the checker was used to generate unit tests. Second, they ran the full application, including user interface, using the checker.

Bounded Verification Dennis et al, found errors in the JML specification for KOA, using bounded verification techniques [20].

[6]Including the 'blanco' or 'blank ballot' candidate.

Chapter 5

Danish Voting

My first case study concerns formal specification of ballot counting in Denmark. The formalisation of the Danish Voting System (DiVS) was achieved in collaboration with MSc student Ólavur Kjölbro [49], who contributed the background knowledge about the workings of the Danish voting system, and worked out the details of the algorithm and proofs.

5.1 Analysis

List Voting is the most common form of proportional representation. Denmark uses an open list voting system; it allows voters to overrule the recommended order of candidates on a party list, because Danish citizens vote for either for one person or else for one party.

In Denmark the D'Hondt method is used to allocate compensatory seats(mandates) by the method of highest averages. Any party with sufficient constituency votes is eligible for compensatory party seats. The second ranked candidate gets half of the

party vote, the third ranked candidate gets one-third of the party vote and so on. Ranking of candidates is based on personal votes received

The Folketing (Danish parliament) is unicameral and has 179 seats of which the Faroe Islands and Greenland have two seats each; the elections in the Faroe Islands and Greenland follow different rules and are not included in this analysis. Out of the 175 seats 135 seats are constituency seats, and 40 are additional seats; the additional seats are used to level unevenness. The electoral map of Denmark consists of 3 provinces and 10 multi-member constituencies.

The allocation of seats is done in 6 steps:

- Allocate constituency seats on constituency level
- Determine threshold on national level
- Allocate additional seats to parties on national level
- Allocate additional seats to parties on province level
- Allocate additional seats to parties on constituency level
- Select candidates

The formal specification covers only the most interesting aspects of the election procedures. These things are left out:

- Register electoral map
- Register parties and candidates
- Party list

- Party gets more seats in a constituency than there are candidates (this rarely happens)
- Substitution list
- Lots drawn on district level when distributing party votes
- Handling rounding errors

5.1.1 State Charts

From the election procedures, state charts can be derived to model the election. The states in figures 5.1 and 5.2 have been grouped together on two levels so that the result is a two-tier state machine. These are referred to as the outer and the inner state machines respectively. The former covers the election procedures, and the latter covers the computation of the results.

The states in the state charts are capitalized in case it is a normal state representing the state that the system (the election) is in currently, e.g. *Election Closed*. The transitions between the states, consist of lower case letters in order to distinguish them from normal states. The actions are denoted by the oblong shapes on the charts.

The states in the outer state machine are all sequential. The state *Before Election* is a before the election when everything is prepared. The states *Election Open* and *Election Closed* are on the Election Day. *Preliminary Results Computed* is usually late on election night. The final counting starts the very next day after Election Day and the *Final Results Computed* is usually about two days after the election.

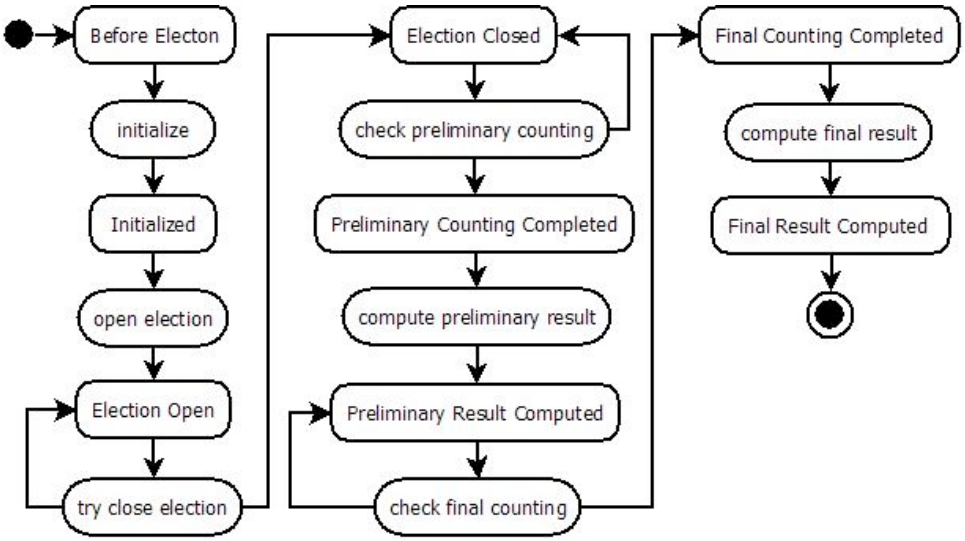

Figure 5.1: Outer state chart

Inside the *compute preliminary results* and *compute final results* action state there is an inner state machine shown in figure 5.2.

These inner states follow directly from the procedure described in the Danish electoral law. If there are no ties that must be broken, then only the action states in the middle are used (i.e. all states/transitions inside the dotted square). The action states starting with the word *resolve* are meant for resolving draw issues. According to the Danish electoral law, ties must be broken by drawing lots. The state *No Passing Parties* is added, since there is no guarantee that any party will pass the threshold. The next transition from *Step 4 Done* is either *resolve step 4 draw* or *allocate additional seats to provinces* dependent on the presence of draws.

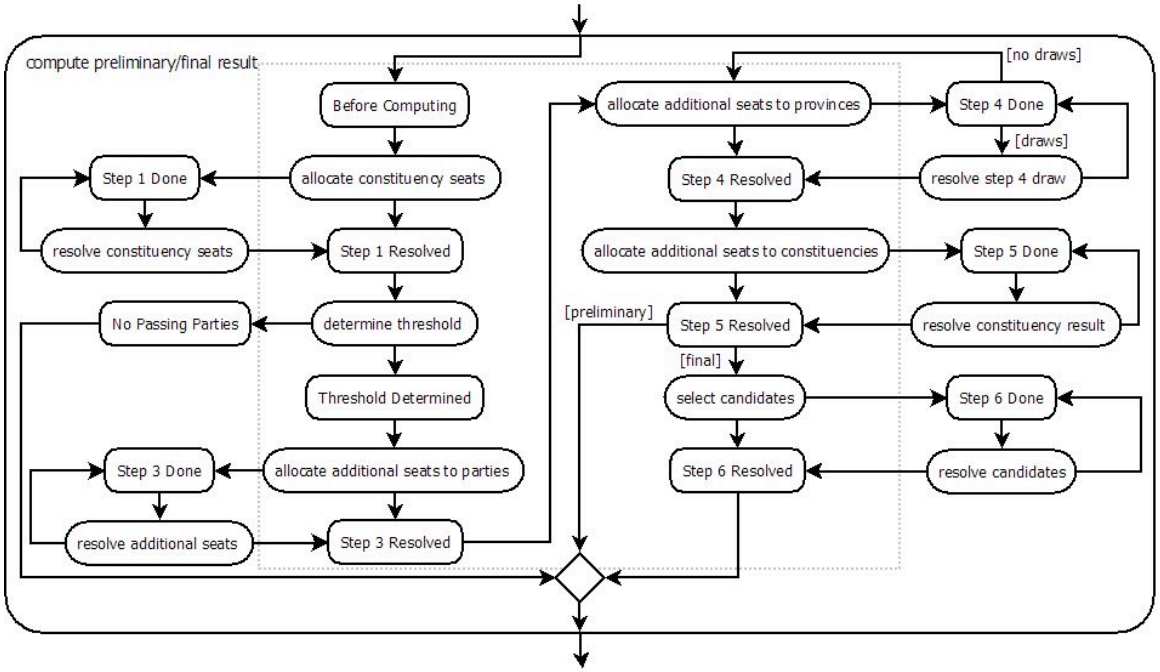

Figure 5.2: Inner state chart for Danish Voting

5.2 Verification and Tests

The intention had been to use an extended static checking (ESC) tool to check the implementation of DiVS, which has been implemented using Java 1.6. It was not possible to use ESC/Java2, because ESC/Java2 only supports Java 1.4. The OpenJML plug-in for Eclipse features an ESC tool that was also tried out. OpenJML uses the JML4c compiler, which does not yet support quantified expressions. Therefore it was left out too. Instead of rewriting the code in Java 1.4 a choice was made to outline a proof of correctness instead. This can be thought of as manual static checking; i.e. the proofs inspect the methods to ensure that they do what the methods must guarantee according to the specifications.

5.2.1 Outlined Proof

In this section three of the transitions are outlined and proven. The first transition is *Check Preliminary Counting* in the outer state machine, the second and third are the *Allocate Constituency Seats* and the *Allocate Constituency Seats to Provinces*; both in the inner state machine. The rest of the transitions are outlined and proven in a similar way for both the outer and inner state machine[1].

Allocate Constituency Seats (Inner State Machine)

The pre-conditions for this transition are:

- For each constituency result there must be a list containing all parties and a list of all independent candidates and their votes. The list of constituency results must be of size 10, which is the number of constituencies in Denmark
- No constituency seats have been taken yet
- No parties have taken any seats yet
- No candidates have been elected yet
- All parties running in the constituencies must be found in the list of parties on the national level as well as on the provincial level for the province in which the constituency belongs
- The inner state must be *Before Computing*

The post-conditions for this transition are:

[1] http://code.google.com/p/danishvotingsystem/wiki/OutlinedCorrectnessProof

- For all constituency results the number of seats taken is greater than or equal to the number of constituency seats belonging to constituency
- If there are no ties to break then all parties on the national level are updated with the number of constituency seats won on the constituency level
- The inner state changes to *Step 1 Done* or *Step 2 Resolved* (if there are no ties to break)

In order to verify the post conditions, it is necessary to look into how the algorithm works:

```
1 for all constituency result
2   FindSeats(ConstituencyResult i)
3 if  (for all constituency) seats taken =
    constituency seats allocated to
    constituency  then
4 update result on the national level

  FindSeats(ConstituencyResult cr)
5 create one div and add it to the div list
    of every party in cr
6 while seats taken < constituency seats
    allocated to constituency  do
7 find the highest quotient among
    parties and vote count for unelected
    independent candidates
```

```
8 locate party and/or un-elected independent
    candidate with highest div value
9 if party then add one seat to party and
    add one more div to div list
    using the d'Hondt method
10 if independent candidate then
     set him/her to elected
11 if seats taken > constituency seats
     available in constituency then
12 mark all parties and/or independent
     candidates with quotient = highest
```

The sixth pre condition ensures that as long as the predicate in the while-loop is true, we will always find the highest quotient in step 7, above. If at least one party gets at least one vote, then the quotient of those parties will always be larger than zero. This is because the number of votes divided by 1, 2, 3, etc. is always larger than 0. At some point the seats taken will be equal or greater than the number of constituency seats belonging to constituency, at which time the algorithm terminates. In the last while-loop iteration, however, there might be quotients that are equal among the involved parties and independent candidates. If they are, then multiple seats get allocated in this last loop, and therefore the number of seats taken in the province might be more than allocated to the province. This is exactly what the first post conditions claims and is hereby verified.

The last if-statement of the algorithm says that if there are more seats taken than there are seats available, then all involved parts are marked. If there is, during

the last iteration of the while-loop, only one seat allocated, then *seats taken* would equal constituency seats available. But if the if-statement in step 11 is true, then there must be more than one allocated during the last iteration and involved parts are marked. This is what the second post condition claims, and it is hereby verified.

At the end of the transition there is a check that investigates the presence of any issues/draws in the constituencies (step 3). If there are no issues, then the number of seats on the national level is updated. This satisfies the third post condition.

It has already been shown that the algorithm will terminate.

Allocate Additional Seats to Provinces (inner state machine)

The algorithm that allocates additional seats to the provincial level is complicated. This subsection verifies both the allocation of the additional seats and the resolving of possible draws. The pre conditions to this transition are:

- The number of additional seats taken must be 0
- For all provinces the number of additional seats taken must be 0
- For all parties in all provinces the number of additional seats taken must be 0
- The total number of additional seats allocated to parties on the national level must be 40
- The total number of additional seats allocated to the provinces is equal to 40
- For all parties in all provinces the size of the div list must equal the number of constituency seats + 1

- All passing parties get at least one vote in all provinces

The pre conditions to this computational step are that all initializations are done properly and no unresolved draws are left from previous transitions. The last pre-condition, above, states the most common case in the elections.

The post conditions are:

- The number of additional seats taken is 40 (total number of additional seats in Denmark)
- All provinces get exactly the number of additional seats that they are allocated before hand
- On the provincial level, all parties, in total, receive the same number of additional seats that they get on the national level
- The inner state is *Step 4 Resolved*

All post conditions in this transition merely state that there are no draws and that all seats get allocated to the parties on the provincial level. In order to verify that the post conditions are met, we need to look at the algorithm, that allocates the additional seats on the provincial level, shown in pseudo code below. An explanation follows.

```
1. while additional_seats_taken < 40 and not
     TooManySeatsTaken do
2.   highest := -1
3.   for all un-arrested passing parties in
       all un-arrested provinces
```

```
4.     find highest quotient
5.     if  no highest quotient found  then  break
6.     ranking := additional_seats_taken
7.     count := 0
8.   for all passing parties in all provinces
9.     for each quotients/div with value = highest
10.  count := count + 1;
11.  set quotients ranking = ranking
12.  add 1 to province.additional_seats_taken
13.      add 1 to party.additional_seats_taken
           (national level)
14.  add 1 to party.additional_seats
           (province level)
15.  add 1 to div list for party on
          province level
16.    accumulate additional_seats_taken by count

  TooManySeatsTaken()
17.  for all province results
18.    if  additional_seats_taken by province
         result > additional_seats allocated to
        province  then
19.   return true
20. for all passing parties on the national level
21.   if  additional_seats_taken by party >
```

```
            additional_seats  then
22.     return true
23.  return false
```

A party result is put to rest if the party result has received its additional seats allocated on the national level or more. A party that is un-arrested has not yet received its additional seats. The same reasoning applies for provinces. Step 4 states: *find highest quotient*.

This means implicitly *find highest quotient that is not crossed out*. The most common case involves no draws. Let's first look into that. The while-loop (line 1) will stop when at least 40 seats are taken. Because of the seventh pre condition it will always be possible to find a highest quotient (line 3 and 4), and therefore the if-statement on line 5 is always false. This causes every iteration of the while-loop to allocate at least one additional seat to a party and a province. The algorithm will therefore terminate at some point since all the additional seats will get allocated.

It is possible, however, that while-loop iterations allocate more than just one additional seat. This doesn't imply, however, that a draw is created; which is what we assume for the moment. Therefore, when there are no draws, all post conditions get verified.

The while-loop will terminate as soon as a draw is present. The definition of a draw is: a party result takes, on the national level, more additional seats than it has allocated, or a province takes more additional seats than it has allocated. The sub-routine *TooManySeatsTaken* illustrates how the presence of draws is found.

The only possible way for a draw to occur is that two or more quotients are equal. Furthermore, these quotients must belong to either the same party in several

provinces (e.g. the last additional seat of a party getting allocated to two provinces) or to the same province for several parties (e.g. the last additional seat of the province getting allocated to two parties).

The worst case scenario is when several parties in several provinces draw for the same seat; i.e. all parties in all provinces have a just claim on a seat, because their quotients are the same. This would be the result of all possible combinations of all passing parties in all provinces; e.g. if there are 10 passing parties then there are 30 involved parts in the draw, taking into account that there are 3 provinces in Denmark.

When the allocation of additional seats to the provincial level stops and the inner state is not *Step 4 Resolved*, it means that there is a draw present. The algorithm below works out the involved parts:

```
    GetStepFourDraw
24.   ranking := -1
25.   for all passing parties in all provinces
26. find the highest quotients ranking
          among the quotients
27. for all passing parties in all provinces
28.   if quotients.ranking = ranking
29.     StepFourDraw.addParty
30.     StepFourDraw.addProvince
```

The algorithm first locates the highest ranking value among the parties on the provincial level. The reason for this is that since the while-loop of the main algorithm breaks as soon as a draw presents itself, the involved parts must have the

highest ranking number. In order to know the involved parts, one simply has to invoke GetStepFourDraw. Steps 29 and 30 add a party-province pair to a StepFourDraw instance.

The electoral law says that lots must be drawn to resolve ties. DiVS must be informed about the result of the lots drawn. DiVS resolves issues through the algorithm ResolveStep4Draw. The algorithm receives three parameters: rank, party result, and province result. The semantics of the parameters is that the party result in province result wins the ranking, and all others lose. The pre condition for the algorithm is that there is a draw that must be resolved, and the post condition is that there is not a draw that must be resolved.

```
    ResolveStep4Draw(int rank, party pres,
      province pr)
31. for all parties in all provinces
32.   if quotients.ranking = rank and
        ( party != pres or province != pr )
33. subtract 1 from
          province.additional_seats_taken
34. subtract 1 from
          party.additional_seats_taken
          (national level)
35. subtract 1 from
          party.additional_seats
          (province level)
36. remove from div list the last div and
```

```
          set ranking of last div to 0
37. subtract 1 from additional_seats_taken
```

The algorithm above resolves the draw issue by making all parts lose the draw except for 1 party-province pair. This means that there are no draws left after this issue, and the next transition from *Step 4 Done* is *allocate additional seats to provinces*. The if-statement in line 32 ensures that all rankings are found, except for the winning party-province pair that the algorithm receives as parameters. This means that if the draw involves 3 party-province pairs and this method is called, then the method removes all but what corresponds to the parameters received; i.e. 2 party-province pairs. Line number 37 is therefore called twice in this working example. And when the main algorithm eventually resumes, then the ranking in line 6 is set to a number which is 1 higher than it was before.

There's no reason to suggest that *ResolveStep4Draw* doesn't terminate. It has already been shown, that when there are no draws then all additional seats are allocated on the provincial level. When draws are involved this is also the case. A possible draw slows down the process since it needs to be resolved before the algorithm is resumed. After resolving one issue, the *additional seats taken* gets closer to 40 by at least 1. It has been shown that *ResolvStep4Draw* completely resolves a draw. Putting these two things together implies that eventually all draw issues are resolved. This is exactly what the post conditions state, and it must therefore be concluded that the algorithm is correct. It has already been shown that the algorithms will terminate.

It is important to note that the last pre condition says that all passing parties get votes in all provinces. If this pre condition is removed, then all parties might run

in only some provinces or just one. The law says explicitly that if there are parties that run in only one or two provinces then their additional seats should be allocated before the others to ensure that the province is not arrested before party's quotient becomes the highest in the main algorithm. This is assuming that there are only a few passing parties that don't run in all provinces and allocating the additional seats to these parties effectively resolves the issues. However, if all parties don't run in all provinces then there is a good chance that the situation mentioned above occurs anyway. This is shown in the counterexample below.

All provinces have 45 constituency seats and 13 or 14 additional seats allocated. 27 parties are running. They all get the same number of votes and they all get 5 constituency seats each. The parties are only running in 1 province. In computation step 3, the parties get allocated 6 or 7 seats total. This means that they get 1 or 2 additional seats. Since all get the same number of votes their fractions are all the same and therefore lots must be drawn.

The lots provide all parties in one province with 2 additional seats. This means e.g. that 18 seats belong to parties in this province. The province has only 13 or 14 additional seats, and therefore the province gets arrested before all the additional seats of the parties running in the province are allocated. In the main algorithm the while-loop will only go through 2 iterations before producing a draw. In the first, 27 additional seats get allocated.

This results in 14 parties getting arrested, since their only additional seat is taken, but no provinces are arrested. In the second, the remaining 13 parties become involved in a draw issue. This issue gets resolved by drawing lots with only one winner, the ResolveStep4Draw is called, and the main algorithm is resumed. The result is, that 15 parties are arrested and 12 parties are involved in a draw.

Again this issue gets resolved, and it goes on.

At some point the province gets arrested, as well as all parties in the other provinces. This means that line 4 will not find any highest quotient, and the if-statement on line 5 becomes true. At this point there are no draws, since TooMany-SeatsTaken returns false. But there exists on the national level a few parties that haven't received their additional seats yet on the provincial level. This violates the post condition that all seats should be allocated.

Therefore the pre condition stating that all parties must get votes in all provinces is kept.

Chapter 6

Irish Voting

The second and more complete case study is about formal specification and testing of Irish PR-STV voting.

6.1 Methodology

To appreciate the rigor involved in formally specifying and verifying a ballot counting system for a non-trivial electoral system like PR-STV, discussing details about our methodology is warranted.

6.1.1 Derivation of formal requirements from legal documents

This is potentially a complex issue. However, the legal framework for voting in Ireland has been in existence for many elections and thus has been well tested and is well understood. In addition, there are official guidelines for voting machine developers and vendors. Thus, the requirements are already laid out in a semi-formal structure, that is relatively unambiguous.

6.1.2 Business Object Notation

Business Object Notation (BON) provides a high-level object orientated description of a system [71]. BON can be thought of as a rigorous subset of UML. BON has two flavors: informal BON and formal BON. Informal BON looks like a structured natural language, but is checked for well-formedness in a variety of ways. Formal BON looks like a strongly typed object-oriented, parametric class-based programming language with contracts and behavioral specifications. Refinement from informal to formal BON is described in the aforementioned text and supported by Fairmichael's BONc tool suite[1].

6.1.3 Java Modeling Language

The Java Modeling Language (JML) is a formal behavioral interface specification language used to specify the behavior of Java software [52]. It extends Java with annotations for specifying simple formal statements in a design-by-contract (DBC) style [12] and model-based specifications a la Larch [11]. Informal BON is either refined to a formal specification in formal BON or directly to a formal object-oriented specification language such as JML. Partial support for checking such refinements is provided by the Beetlz tool[2].

The initial JML specifications for Vótáil were written by hand based on the ASM, the table of functional requirements and the informal BON. For the final version of Vótáil, *Beetlz* was used to help derive the formal BON from the existing JML specification.

The JML specifications cover the critical sub-systems of Vótáil, namely the

[1] `http://www.kindsoftware.com/products/opensource/BONc/`
[2] `http://www.kindsoftware.com/products/opensource/Beetlz/`

election.tally package.

Beetlz Beetlz is a tool for checking the consistency between BON and JML specifications. It is neither sound nor complete, but it does help to speed up the process of writing formal specifications, by reducing the need for manual translation from JML to BON, although as with any automated translation, the results are not always precisely correct.

Therefore the refinement from BON to JML needs to be manually inspected and corrected by hand.

6.1.4 Abstract State Machines

Abstract State Machines (ASM) are embedded within the JML specification,[3] so as to ensure that methods are called and used in the correct order. Whereas normally, a precondition applies only to parameters and not to fields, we use a field called 'state' which represents the ASM for the class. Each transition in the ASM is implemented as a *Java* method, for which the precondition is to be in one state and the postcondition is to be in another state. The principal class, *BallotCounting* has a two tier state machine. The *Ballot* and *Candidate* classes each have their own simpler ASM.

Outer ASM for Count Process

The outer ASM is as described in previous work [14, 44]. Some examples of the invariants for each outer state, excluding technical constraints such as non-nullity,

[3]The JML specification is a refinement of the ASM as well as being a refinement of the BON design.

States	Invariant	JML
PRELOAD	No ballots loaded yet.	totalVotes == 0;
PRECOUNT	No candidates elected yet.	numberElected == 0;
PRECOUNT	No candidates excluded yet.	numberEliminated == 0;
COUNTING	There is at least one seat for election.	1 <= seats;
COUNTING	Number of seats remaining	remainingSeats == (seats - numberElected);
FINISHED	All seats filled.	numberElected == seats;

Figure 6.1: Invariants for some of the outer states

From State	To State	Java Method
Ready to Count	No Seats Filled Yet	BallotCounting
Surplus Available	Ready for Next Round of Counting	distributeSurplus

Figure 6.2: Part of the Inner ASM for ballot counting process

are shown in figure 6.1.

Inner ASM for Count Process

The inner ASM models the detailed count process. The possible states and transitions are described in figure 6.2.

The inner ASM is implemented in the *AbstractCountStatus* class.

Candidate ASM

The Candidate can be in one of three states. The final state is either *Elected* or *Excluded* and the initial state is *Continuing*.

Ballot ASM

The specification of the *Ballot* class also enforces the process of transferring the ballot to another candidate. The transfer function moves to the next effective pref-

erence on the ballot paper.

6.1.5 A Verification-centric Development Process

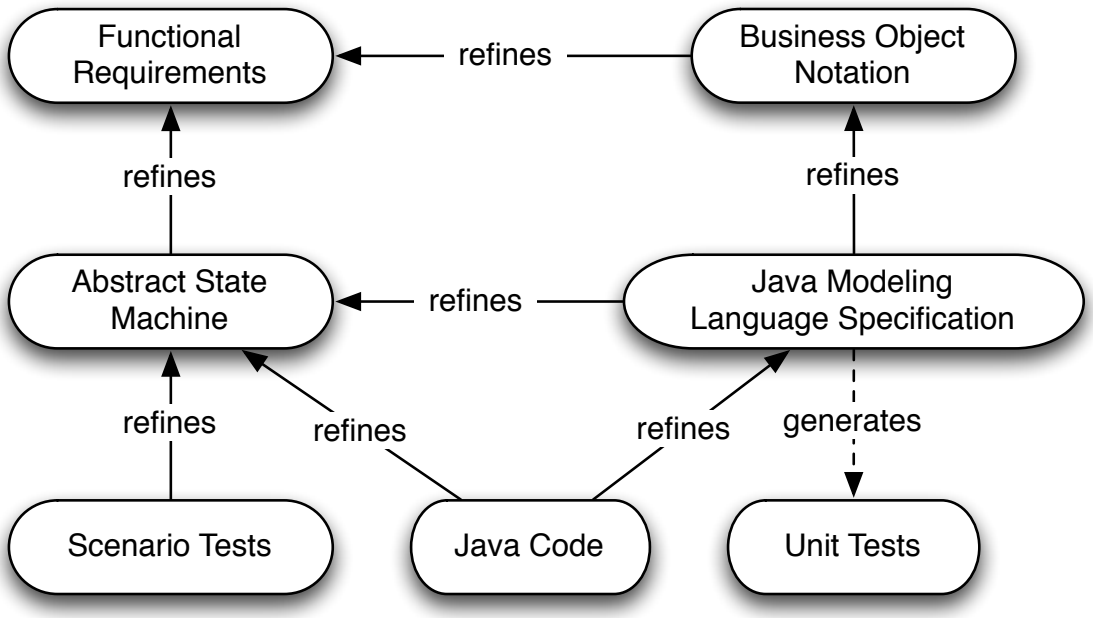

Figure 6.3: Relationships between software engineering artifacts.

A set of functional requirements and features, derived from electoral law, is a semi-formal specification, although written in a structured way. To translate the ballot counting process, as defined by law, into an executable software system we define an abstract state machine (ASM). This ASM and a set of functional requirements (described later) are refined into an object-oriented system design using BON, which is in turn refined into a JML contract-based specification. The JML specification and ASM are then implemented in Java. Thus, the development follows a strict design-by-contract based approach to software engineering.

Table 6.1: Cross-referencing functional requirements and the law.

ID	Functional Requirement	Section	Item	Page
9	Not more than one surplus is distributed in any one count.	4	3	16
10	Where there are seats remaining to be filled, but no surpluses available for distribution, the lowest continuing candidate or candidates must be excluded.	4	4	16
11	There must be at least one continuing candidate for each remaining seat.	4	4	16
	...			

Validation is accomplished via testing. Automated tests are generated from the JML specification, and scenario tests are derived from the ASM. Finally, the entire system is verified using extended static checking, a kind of automated functional verification.

Figure 6.3 provides an overview of these artifacts and their interrelationships.

6.2 Formal Specification

Requirements are derived from electoral law and regulations for the counting of votes. In the Irish context, these requirements come from a commission on voting and electoral law and from the electoral act itself.

The 1992 Electoral Act, including subsequent amendments, and the Commentary on Count Rules issued by the CEV [23], is the starting point for the requirements analysis. In previous work 39 semi-formal statements are used to describe these functional requirements for ballot counting in elections to the Dáil [14].

A few example formal statements from our previous work are listed and cross-referenced, as shown in Table 6.1. The *section*, *item* and *page* column titles refer to

the CEV Commentary on Count Rules, which in turn refers back to the Electoral Acts.

The formal specification has several aspects. First, we must formalize the ballot counting process — the steps through which one must pass to convert a pile of legal ballots into a tally. Secondly, we must capture the various stages through which each key element of the counting process (e.g., a candidate, a ballot, a ballot box, etc.) can pass. The formalization of these two different, but interrelated, facets of the specification are done via the use of ASMs.

6.2.1 Abstract State Machine

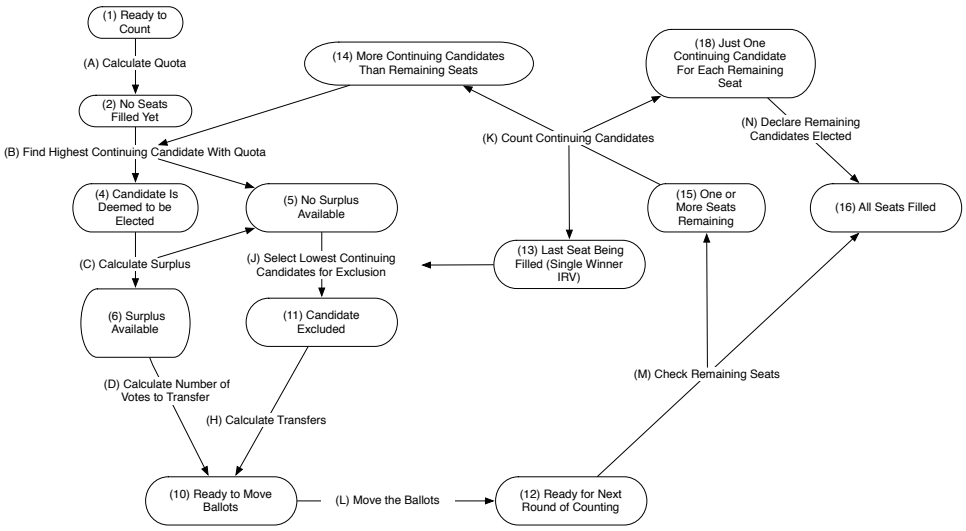

Figure 6.4: Abstract State Model, lower tier (sub-states)

A two tier Abstract State Machine (ASM) is used to represent the 39 functional requirements. The upper tier of the ASM describes the state of the election (`EMPTY`, `SETTING_UP`, `PRELOAD`, `LOADING`, `PRECOUNT`, `COUNTING`, `FINISHED`,

AUDIT, REPORT) in a linear way, in which there is only one possible transition into and out of each state, whereas the lower tier of the ASM (shown in 6.4) is more complex and describes more detailed sub-states and transitions within the COUNTING state.

6.2.2 Invariants

An invariant is a predicate about a set of objects in the system that must always hold during stable/quiescent states during system execution. In essence, the invariants of an object and its class hierarchy explain what constitutes a valid instance of the object in question. Likewise, invariants about the states of an ASM explain what must be true of the process and the objects on which it operates for the process itself to be valid.

```
/** Number of candidates elected so far */
public model int numberElected;
public invariant 0 <= numberElected;
public invariant numberElected <= seats;
public invariant (state <= PRECOUNT) implies numberElected == 0;
protected invariant (COUNTING <= state) implies
  numberElected == (\num_of int i;
  0 <= i && i < totalCandidates; isElected(candidateList[i]));
public invariant (state == FINISHED) implies numberElected == seats;
```

Figure 6.5: A JML specification describing the number of candidates elected.

Each election state has a number of invariants that must hold. For example, in the fragment of JML seen in figure 6.5, the *Finished* state has as an invariant that the number of candidates elected equals the number of seats available, whereas the *Pre-Count* state has an invariant that the number of candidates elected (so far) is zero. Invariants are coupled to states in the ASM through a variable state

denoting the obvious. Some invariants must only hold in a given state and others must hold for that state and all future states, as in the example.

6.2.3 Coupling State Transitions

Reasoning about the overall correctness of the counting algorithms implementation boils down to reasoning about the top-level ASM. If we can show that each transition in the ASM is valid (it only goes from legal pre-states to legal-post states, as defined by law), then we can guarantee, by transitivity, that the overall algorithm is correct. The correctness of these transitions is captured entirely by invariants, which are preserved in the top-level state transitions.

Similar reasoning is used to analyze the correctness of each invariant on each state of the ASM, invariants that span ASM states, as well as the legitimacy of transitions between states.

6.2.4 Other Examples of Invariants

All invariants must hold in every state, not just those state pairs at the end of transitions in the top-level ASM. These legal invariants are expressed by class and object invariants in the JML specification. Consequently, when a transition between states occurs, the invariants of both the old and new state must hold during the transition (i.e., during any helper methods that are called while the software is moving between states).

Transitions into	Invariant for new state
End of Count	The number of candidates elected equals the number of open seats.
No Surplus Available	All continuing candidates have less than a quota of votes.
No Seats Filled Yet	The number of elected candidates is zero.
Candidates Have Quota	There exists a continuing candidate with at least one quota of votes.
Candidate Excluded	This candidate had fewer votes than any other continuing candidate.
Last Seat Being Filled	The number of elected candidates is one fewer than the number of open seats.
Seats Remaining	The number of elected candidates is less than the number of open seats.

Figure 6.6: Examples of invariants for each sub-state, translated from JML to English for the reader.

6.2.5 Refinement to BON

To formally capture legal requirements, as expressed through invariants, and to rigorously refine our ASMs into a software system, the architecture of our ballot counting system (i.e., its classifiers and their relations) and its correctness properties (i.e., its invariants) are formally specified in the Business Object Notation.

Each state transition in the Abstract State Model is represented either by a command or a query in BON. In BON, a command is an action that changes the state of an object, for example, moving a ballot from one pile to another, whereas a query returns some information about the system. A query is implemented in JML either as a field with invariants or as a pure method.

The example in figure 6.7 shows an informal BON description of the Ballot Counting process.

```
class_chart BALLOT_COUNTING
explanation
  "Count algorithm for tallying of the votes
   in Dail elections"
query
  "How many continuing candidates?",
  "How many remaining seats?",
  "What is the quota?",
  "Who is/are highest continuing candidate(s)
   with a surplus?",
  "What is the surplus?",
  "What is the transfer factor?"
command
  "Distribute the surplus ballots",
  "Select lowest continuing candidates for
   exclusion",
  "Declare remaining candidates elected",
  "Close the count"
end
```

Figure 6.7: An Informal BON description of the Ballot Counting class.

6.2.6 Refinement to JML Specification

The BON design contains 1 cluster[4] with 5 classifiers, 20 queries, 5 commands and 6 constraints. These are refined to 1 package with 10 classes, 104 methods, 70 invariants, 192 preconditions and 117 postconditions in JML.

We used a version of JML that extends Java 1.4, because of existing mature tool support. We also minimize our use of the JDK and use simple data structures such as arrays. When using arrays in JML we make assumptions about the maximum number of candidates and the maximum number of ballots, based on past elections and the theoretical maximum population of a constituency.

[4] A BON cluster is a collection of related concepts, roughly similar to a Java package.

```
/** Number of votes needed to win a seat */
requires 0 <= seats;
ensures \result == 1 + (totalVotes / (seats + 1));
public pure int getQuota();
```

Figure 6.8: An example of a JML specification for a BON query.

Two examples of JML are shown in figures 6.8 and 6.9, one for a query and one for a command, and are examples of the initial JML specification written during refinement. Such a specification contains only the signature of each method without implementation code (the implementation is "bottom," aka "assert false."). Note also that the method signature specification in this example states in a precondition that none of the parameters can have null values.

6.2.7 Limited Verification

Our toolset for verification has some important limitations. This means that we cannot achieve full program verification with the existing generation of JML tools.

Length of Verification Conditions

In many cases, specifications need to be rewritten and simplified so as to avoid overly complex verification conditions, that ESC/Java2 (or its theorem provers) are unable to handle.

Loop Invariants, Summation and Generalized Quantifiers Some calculations were specified as a summation and implemented using loops. ESCJava2 could neither verify the loop invariants nor the post-condition of such methods, for example in the method *Candidate.getTotalVote*:

```
/**
 * Transfer votes from one candidate to
 * another.
 *
 * @param fromCandidate
 *   Elected or excluded candidate
 * @param toCandidate
 *   Continuing candidate
 * @param numberOfVotes
 *   Number of votes to be transferred
 */
requires fromCandidate.getStatus() != CandidateStatus.CONTINUING;
requires toCandidate.getStatus() == CandidateStatus.CONTINUING;
ensures countBallotsFor(fromCandidate.getCandidateID()) ==
  \old (countBallotsFor(fromCandidate.getCandidateID())) -
  numberOfVotes;
ensures countBallotsFor(toCandidate.getCandidateID()) ==
  \old (countBallotsFor(toCandidate.getCandidateID())) +
  numberOfVotes;
public abstract void transferVotes(
  final non_null Candidate fromCandidate,
  final non_null Candidate toCandidate,
  final int numberOfVotes);
```

Figure 6.9: An example of a JML specification for a BON command.

The solution, in this case, is to *strengthen* the loop invariant, so as to make it easier to verify.

In cases where this is not possible, the alternative solution is to isolate the unverified part of each method, using refactoring, so that the offending code would be easy to inspect manually. It is hoped that future verification tools such as OpenJML will have more success with this code.

The Java code is written in such a way as to be *verification friendly*, in many cases simplified so that it is easy for verification tools (such as ESCJava2) to understand, for example by avoiding reliance on third-party or non-standard APIs. This also helps make the code easy to read.

```
/**
 * Total number of votes received by or added to this candidate.
 *
 * @return Gross total of votes received
 */
requires lastCountNumber < votesAdded.length;
ensures \result == (\sum int i;
  0 <= i && i <= lastCountNumber; votesAdded[i]);
public pure int getTotalVote() {
  int totalVote = 0;

  loop_invariant votesAdded[i] <= totalVote;
  for (int i = 0; i <= lastCountNumber; i++) {
    totalVote += votesAdded[i];
  }

  return totalVote;
  }
```

Figure 6.10: An example of a JML Loop Invariant.

Decision to use Java instead of a functional programming language

Java, version 1.4, is used because it is multiplatform and has existing tool support from the JML community, and others. However, a functional programming language or domain specific language might lead to a simpler specification and implementation.

However many of the ESC warnings found would have been unnecessary in a more strongly typed language, or even in a more recent version of Java, but there was no JML tool support for modern Java, when this work was started.

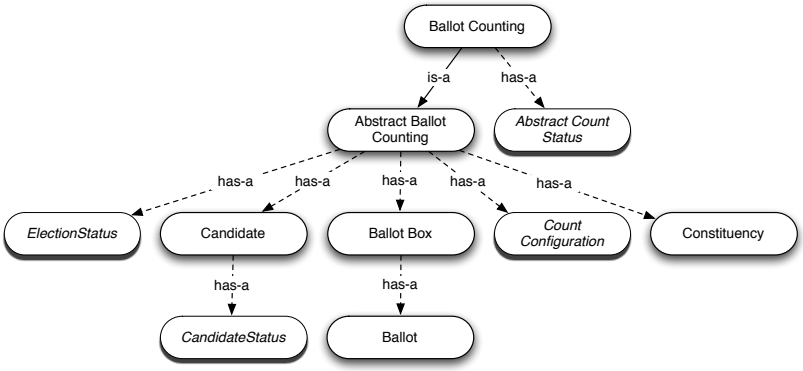

Figure 6.11: Relationship between classes

6.2.8 Architecture

election.tally package

There are 10 Java classes in the election.tally package, representing the actors in the system, for example Ballots, Ballot Boxes and Candidates. Figure 6.11 shows the relationship between the Java classes. The `BallotCounting` class contains the specifics of PR-STV, whereas the `AbstractBallotCounting` class contains the more general properties of ballot counting algorithms. Class names shown in italics are supporting classes that were added in the Java implementation but were not refined from BON.

other supporting packages

ie.votail.test This package contains ten hand-written scenario tests derived from the inner ASM of the *BallotCounting* class. These tests cover 97 % of the *election.tally* package, according to the metrics used by the Emma plugin for Eclipse.

ie.votail.model, ie.uilioch These packages contain the framework for model-based test generation, which will be described in the next chapter.

6.3 The Vótáil Hypothesis

Due to the manner in which the formal specification of the ballot counting algorithms is accomplished and the aforementioned argument about the correctness of state transitions (section 6.2), we summarize via informal refinement the overall theorem expressed by this ballot counting system as the *Vótáil Hypothesis*.

> **Vótáil Hypothesis:** Given a valid set of candidates up for election for a set of seats, and a ballot box containing a valid set of ballots, after the ballot counting algorithm executes, we guarantee that the candidates deemed elected by Vótáil are exactly those elected by Irish law, where *casting of lots* and *shuffling of ballots* are simulated by sorting the ballots and candidates into a secret order beforehand.

In other words, the candidates elected by a machine count are the same as would be elected in a *correct* manual count of paper based ballots.

6.4 Verification and Validation

6.4.1 Open Source Implementation

The Vótáil source code is open source, under the terms of the MIT open source license[5], and is available from the Mercurial repository on *JavaForge* `http://`

[5] `http://www.opensource.org/licenses/MIT`

javaforge.com/hg/voting[6]. The source code contains 723 Java statements in 11 classes and 92 methods.

6.4.2 Scenario Tests

Ten hand-written scenario tests are derived from the ASM and provide 97 percent code coverage. To measure coverage we use the EclEmma[7] code coverage plug-in for the Eclipse Integrated Development Environment[8] and run tests using JUnit.

However 100% code coverage can be achieved using automatic generation of test data, derived from a formal model of the ballot counting process. This is described in chapter 7.

6.4.3 Extended Static Analysis

The Extended Static Checker for Java version 2 (ESC/Java2) is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations [16]. Users control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas. ESC/Java2 is used to type check the JML specifications and to check that the Java implementation fulfills these specifications.

No flags were provided to ESC/Java2, I used the default configuration for the ESC/Java2 Eclipse plugin as part of the Mobius Program Verification Environment (PVE) configuration for Eclipse. I used the *loopsafe* option, as there are many loops in the algorithm.

[6] http://www.javaforge.com/project/5342
[7] http://www.eclemma.org
[8] http://www.eclipse.org

Frame Conditions Each method has a full assignable clause. I relied on ESC/-Java2 to check these frame conditions.

ESC/Java2 is used to both type check the JML specifications and to check that the Java implementation fulfills these specifications.

This verification is complemented by the aforementioned testing because ES-C/Java2 is neither sound nor complete. While we have used its functionality to check that specifications are sound [41] and that we have not ventured into any territory that touches on the soundness and completeness issues inherent in the tool's design and implementation [45], only via rigorous, well-designed testing are we assured that the system is functioning correctly in an actual execution environment.

Chapter 7

Alloy Model

7.1 Model Driven Testing

As stated previously, static analysis, in itself is neither sound nor complete, whereas Runtime Assertion Checking (RAC) is sound but requires a complete set of test cases. We wish to achieve full statement and path coverage, however we cannot test all possible inputs to the system in a reasonable time frame.

Many researchers have already written about the limitations of random test generation [18, 15]. A more promising approach, albiet with some limitations, is the use of model checkers to generate witnesses or counter examples. At first glance, this might appear to be similar to our approach, but has the disadvantage of state-space explosion, and does not guarantee full path coverage [5, 38].

Dijkstra's Dictum *Program testing can be used to show the presence of bugs, but never to show their absence* [25].

Jackson's Small Scope Hypothesis *Most bugs have small counterexamples* [40].

In order to show the absence of bugs with the smallest possible test cases I decided to use the Alloy model finder to describe the PR-STV count algorithm. In this chapter I describe my formalization of Irish PR-STV using the Alloy Analyzer for the purpose of test generation. The model contains 2 enumerations, 5 signatures, 60 facts and 24 assertions for PR-STV. A *fact* in Alloy is an axiom, and an *assertion* is a lemma. Later in this chapter, I show how to use the *Alloy Analyzer* to model PR-STV and then generate a complete set[1] of PR-STV test cases for use with Runtime Assertion Checking (RAC). Remember that RAC is sound (unlike Extended Static Checking which is neither sound nor complete), but its completeness depends on the completeness of the test data. The test data needs to be complete, not in the sense of covering every possible combination of ballot papers, but in covering every possible type of election outcome and every branch through the vote counting algorithm.

Number of Different Combinations of Ballot Papers

To concretize completeness we need to analyze the space complexity of elections.

The number of distinct permutations of non-empty preferences is $\sum_{l=1}^{C} (C)_l$, where $C = |\mathcal{C}|$ and partial ballots are allowed, so that the number of preferences used range in length from one to the number of candidates. For a ballot of length l, $(C)_l$ is the number of distinct preferences that can be expressed.

In general, the number of possible ballots grows factorially (i.e. faster than exponential), while the number of equivalence classes due to symmetries grows linearly.

[1] a set that contains at least one representative from each equivalence class (election scenario)

Unfortunately, there is no obvious way to leverage these symmetries in the generation of tests from a naive ballot-centric point-of-view because the number of ballot boxes grows exponentially. For example, even if we know that there are only a linear number of equivalence classes of ballots, how do we know how many ballot boxes to generate to fully test a system?

One approach to validating/testing electoral systems (if they are tested at all) is to randomly generate hundreds of thousands (or, indeed, even millions) of ballot boxes and then to compare the results of executing two or more different implementations of the same voting scheme. If different results are found, then the ballots are counted manually to determine which result is correct [17].

This methodology has limited effectiveness, because even if one generates billions of ballots in non-trivial election schemes, the fraction of the state space explored is vanishingly small. To make this clear, we will analyze the number of distinct combinations of preferences possible in PR-STV ballots.

Number of Distinct Outcomes

If B is the number of distinct non-empty ranked choices that can be made, and $V = |\mathcal{B}|$ is the number of votes cast, then the number of possible combinations of ballots is B^V if the order of ballots is important, and $\frac{B^V}{V!}$ if not.

For example, consider a five seat constituency with a voting population of 100,000 and 24 candidates. Consequently, the number of possible ballot boxes is $(\sum_{l=1}^{24} (24)_l)^{100,000}$, an astronomical number of tests which would be impossible to run, even if each individual test took only a fraction of a second.

To avoid this explosion, we partition the set of all possible ballot boxes into equivalence classes with respect to the counting algorithm chosen. We consider

the equivalence class of election results for PR-STV. Each election result consists of the set of Election Outcomes for each candidate.

7.1.1 Election Outcomes

The key idea is that election scenarios represent an equivalence class of election outcomes, thereby letting us collapse the testing state space due to symmetries in candidate outcomes. We will return to this point in detail below in the early examples.

Each election result is described by an *election scenario* which is a vector of *candidate outcome events*. Both of these terms are defined in the following.

The inner ASM of the ballot counting algorithm (see figure 6.4) is modeled as a set of events defined in figure 7.1. The paths through the Inner ASM for each election outcome are shown in figure 7.2. Note the election outcomes are defined more precisely than the ASM (to capture the movement of ballot transfers as well as the final outcome for each candidate) so that two or more Election Outcomes can traverse the same path through the ASM. In particular the ASM does *not* show a separate branch for handling of non-transferable votes, whereas both the JML refinement and the Alloy refinement do include the notion of non-transferable votes, of the *tie break* and of the *threshold* for example.

The formal definition of an election outcome is found in the axioms that reference that outcome, for which an example will be given later after the definition of each signature in the model.

7.1.2 Ballots and Ballots Boxes

The ranked choice (preference) ballot is modelled as follows:

Outcome	**Informal Definition**
WinnerNonTransferable	elected on first round with at least one non-transferable surplus vote
SurplusWinner	elected on first round with at least one surplus vote (all surplus votes are transferable)
Winner	elected in the first round of counting by quota but with no surplus
AboveQuotaWinner	elected with transferable surplus votes on second or later round after receipt of transfers
QuotaWinnerNonTransferable	elected on second or later round after receiving transfers with at least one non-transferable surplus vote
QuotaWinner	elected with quota after transfers from another candidate, but without a surplus
CompromiseWinner	elected on the last round of counting without quota but by plurality
TiedWinner	elected by tie breaker in the last round
TiedLoser	defeated only by tie breaker but reaches the threshold
TiedSoreLoser	defeated only by tie breaker but does not reach threshold
Loser	defeated in last round but reaches the minimum threshold of votes
SoreLoser	defeated in last round, but does not even reach the minimum threshold of votes
EarlyLoserNonTransferable	reaches threshold but is excluded before last round with some non-transferable votes
EarlyLoser	reaches threshold but is excluded before last round, all votes are transferable
EarlySoreLoser	excluded before last round, and below threshold, all votes are transferrable
EarlySoreLoserNonTransferable	below threshold and excluded before last round with at least one non-transferable vote

Figure 7.1: Informal Definition of each Election Outcome

States	Outcome
Candidate Deemed Elected, Surplus Available	SurplusWinner
Candidate Deemed Elected, No Surplus Available	Winner
Candidate Deemed Elected, Surplus Available	WinnerNonTransferable
Candidate Deemed Elected, Surplus Available	AboveQuotaWinner
Candidate Deemed Elected, Surplus Available	QuotaWinnerNonTransferable
Candidate Deemed Elected, No Surplus Available	QuotaWinner
Last Seat Being Filled, All Seats Filled	CompromiseWinner
Last Seat Being Filled, All Seats Filled	TiedWinner
Last Seat Being Filled, All Seats Filled	TiedLoser
Last Seat Being Filled, All Seats Filled	TiedSoreLoser
Last Seat Being Filled, All Seats Filled	Loser
Last Seat Being Filled, All Seats Filled	SoreLoser
Candidate Excluded	EarlyLoserNonTransferable
Candidate Excluded	EarlyLoser
Candidate Excluded	EarlySoreLoser
Candidate Excluded	EarlySoreLoserNonTransferable

Figure 7.2: Examples of paths through the Inner ASM for each Election Outcome

```
sig Ballot {
  assignees:   set Candidate,
    -- Candidates to whom this ballot has transferred
  preferences: seq Candidate  -- Ranking of candidates
}
```

where the *assignees* field is used to track the movement of the ballot. In the first round of counting, each ballot has just one assignee, then new assignees are added if the ballot is transferred. There is no need to know the order of assignees because they follow the same order as the preferences, although preferences can be skipped over, if the intervening candidates have been already elected or excluded, since a ballot will be transferred to the highest preference for a *continuing* candidate (not already elected or excluded).

The Ballot Box is defined as follows:

```
one sig BallotBox {
  spoiltBallots:    set Ballot, -- empty ballots excluded from count
  nonTransferables: set Ballot,
     -- ballots for which all preferences are used up
  size:      Int  -- number of unspoilt ballots
}
{
  no b: Ballot | b in spoiltBallots and b in nonTransferables
  size = #Ballot - #spoiltBallots
  all b: Ballot | b in spoiltBallots iff #b.preferences = 0
  all b: Ballot | some c: Candidate |
    b in nonTransferables implies b in c.wasted
}
```

note that to avoid duplication of data, and adding too many fields to the model, there is no direct mapping between the Ballot Box and every single ballot. There is however a mapping between ballots and candidates.

7.1.3 Candidates

The stack of ballots in favor of a given candidate is modelled as follows:

```
sig Candidate {
  votes: set Ballot,     -- First preference ballots assigned
  transfers: set Ballot, -- Second and subsequent preferences
  surplus: set Ballot,   -- Ballots transferred to another
  wasted: set Ballot,    -- Ballots non-transferable
  outcome: Event         -- Election outcome for each candidate
```

```
}
```

In Alloy an axiom is either labelled as a *fact* or else appended to a type signature *sig*. However it was found that the Alloy API when used from Java does not load *facts* unless they are appended to the signature.

For example, the following appended axiom (on Ballots) enforces consistency between the definition of ballots and candidates.

```
all c: Candidate | preferences·first = c iff this in c·votes
```

The appended axioms on Candidate enforce the meaning of the various outcomes, for example, certain outcomes are possible only when there is a least one non-transferable (wasted) vote:

```
0 < #wasted iff (outcome = WinnerNonTransferable or
  outcome = QuotaWinnerNonTransferable or
  outcome = EarlyLoserNonTransferable or
  outcome = EarlySoreLoserNonTransferable)
```

7.1.4 Electoral Constituency

The constituency for the election is modeled as follows:

```
one sig Election {
  seats:             Int,    -- number of seats for election
  constituencySeats: Int,    -- full number of seats
  method:            Method -- PR-STV or plurality
}
{
```

```
  0 < seats and seats <= constituencySeats
  seats < #Candidate
  method = Plurality or method = STV
}
```

In a special election, or by-election, the number of seats to be filled can be less than full number of seats. If only one seat is being filled then PR-STV reduces to Instant Runoff Voting (IRV). The model also has parameters to support Plurality elections by narrowing the range of possible outcomes and disallowing of transfers.

7.1.5 Electoral Scenarios

The Electoral Scenario is defined informally as a sequence of Election Outcomes, but more formally as a set of winners and a set of losers, with a subset of (early) losers excluded before the last round. The scenario is effectively the *input* to the Alloy solver and the set of ballots is the *output* for a given model.

```
one sig Scenario {
  losers:     set Candidate,
  winners:    set Candidate,
  eliminated: set Candidate, -- Candidates excluded before last round
  threshold:  Int,           -- Minimum number of votes for funding
  quota:      Int,           -- Minimum number of votes for election
  fullQuota:  Int            -- Quota for full election
} {
  // Mandatory pairs of outcomes: ties between winners and losers
  all a: Candidate | some b: Candidate |
   (a.outcome = TiedLoser or a.outcome = TiedSoreLoser)
```

```
    iff (b·outcome = TiedWinner)
}
```

The following appended axioms require the *winners* and *losers* are disjoint, but that *eliminated* is a subset of *losers*.

```
all c: Candidate | c in winners + losers
no c: Candidate | c in losers & winners
eliminated in losers
```

Now that the signatures are defined, the election outcomes can be given a more formal definition.

7.1.6 Formal Definition of Election Outcomes

The formal definition of each Election Outcome is the set of axioms that reference it. As an indication, the axioms for the *AboveQuotaWinner* are listed below:

```
all c: Candidate | (c·outcome = AboveQuotaWinner) implies (
  (Scenario·quota < #c·votes + #c·transfers) and
  (not Scenario·quota <= #c·votes) and
  (#c·surplus = #c·votes + #c·transfers - Scenario·quota) and
  (c·surplus in c·transfers) and
  (#c·wasted = 0))
```

The rest of the appended axioms can be found in appendix C.

The axioms represent rules expressed in electoral law, although in a more precise way. For example, the electoral law talks about *elected*, *excluded* and *continuing* candidates, but does not explicitly talk about election outcomes. This contrasts with the BON/JML specification and the ASM in which each state and transition

can be referenced to a paragraph or subparagraph of electoral law. Therefore the definition of an Election Outcome is somewhat arbitrary and alternative refinements might have been possible.

7.2 Validation of the PR-STV Alloy Model

How then do we know if the axioms are complete and non-contradictory?

The internal consistency of the Alloy model is checked in two ways, using Alloy assertions (lemmas) and Alloy predicates (scenarios). The axioms are refinements of explicit rules in the electoral law, whereas the implicit requirements are stated as lemmas.

7.2.1 Lemmas

If the assertion contradicted the existing axioms, then the counter-example, from the unsatisfiability core, was examined carefully, to see which axiom or lemma was correct. This is one of the strengths of using the *Alloy Analyzer*.

For example, the following assertion requires that if one candidate won by tie breaker, then another candidate must have lost by tie breaker, *with an equal number of votes and transfers*[2].

```
assert wellFormedTieBreaker {
  some w,l : Candidate | (w in Scenario·winners and
    l in Scenario·losers and #w·votes = #l·votes and
    #w·transfers = #l·transfers) implies
    w·outcome = TiedWinner and
```

[2]Strictly it would require an equal number of votes and transfers on each round of counting, but the model does not specify in which round the transfers were received

```
    (l·outcome = TiedLoser or l·outcome = TiedSoreLoser) }
```

The next example of a lemma is an implicit rule that there cannot be more preferences than candidates:

```
assert lengthOfBallot {
  all b: Ballot | Election·method = STV implies
  #b·preferences <= #Candidate }
```

7.3 Automated Test Generation using Alloy Predicates

The question arises, how do we find witnesses for each distinct outcome, that is how do we find the smallest set of test ballots required for each outcome, where each outcome represents a distinct path through the algorithm.

These generated test cases are used to complement existing hand-written unit tests. To accomplish this task, one needs to be able to generate the ballots in each distinct kind of ballot box identified using the results of the earlier sections of this paper. Effectively, the question is one of, "Given the election outcome R, what is a legal set of ballots B that guarantees R holds?"

7.3.1 Generation of Ballot Boxes for Test Cases

Using the *ie.votail.model.factory.ScenarioFactory* class it is possible to enumerate through all the possible thousands of combinations of Election Outcomes for a given number of seats and a given number of candidates. Scenarios which contain incompatible sets of outcomes, e.g. a CompromiseWinner with a TiedLoser, are discarded.

The remaining valid scenarios are then loaded one at a time as predicates into the Alloy model. If a solution is not found for a small scope, then the predicate is run with a larger scope. If no solution is found for a scope as large as 30, for example, then it is noted in a log file for further investigation to confirm that the scenario has no solution.

The solution space for each scenario is then parsed so as to extract the ballots which are then saved into a test database.

This test database then forms the dataset for testing with RAC.

If all configurations are explored and the expected scenario matches the actual results, then we know that the implementation correctly implements all branches of the voting scheme's specification. If some of the results are incorrect, then we have found a bug in the ballot counting implementation, assuming that the model is perfect. If code coverage is less than complete, but the model is complete, then we have discovered hidden functionality (or dead code) in the system under test, keeping in mind that the model is still an *abstraction* of the underlying software. For example, the earliest version of the PR-STV model did not distinguish between transferable and non-transferable surpluses.

The *ElectoralScenario* class contains an ordered sequence of Election Outcomes e.g. one winner above quota, one compromise winner, two early losers etc.

7.3.2 Coverage Analysis

Beyond the issue of correctness, we can also analyze the completeness and efficiency of the implementation. In particular, by running all validation tests while measuring code coverage, we can discover:

- which parts of the code are exercised most, and thus are the prime focus for the application of verification technologies,
- which parts of the code are not exercised at all, and thus are dead code, or are code representing misinterpretations of the voting scheme's counting rules, and
- if the system crashes, the inputs that were unexpected, which in turn tells us about misinterpretations of the scheme

Results The test and coverage analysis can be seen in section 8.3.

Chapter 8

Conclusions

The most vulnerable part of the system is the link between the polling station and the counting system, namely that candidate ID numbers could be swapped before loading of the ballot data into the counting system. The twin defenses against such an attack are an independent audit trail and homomorphic encryption of ballots [31], which are outside the scope of this dissertation.

Therefore the correctness of the implementation cannot be guaranteed, but instead a measure of the completeness of verification and validation is given.

However the use of lightweight formal methods remains a viable alternative to code inspection and intensive manual testing of software. In particular, a test strategy could be designed to focus on those parts of the system with less coverage from the verification tools.

8.1 Limitations

1992 Electoral Act, Section 121, subsection (8) There is one scenario which I did not include in my model of PR-STV, the lowest continuing candidate can be excluded before the distribution of a surplus, if that candidate already has enough votes to reach the quarter-quota threshold, or if the surplus would not be enough to make a difference.

8.2 Vulnerabilities

STV, without fractional counting, is subject to two forms of counting attack in addition to the *ballot signing* attack. [1] The first attack is to bias the shuffling of ballots so that ballots with third and subsequent preferences for the chosen candidate are closer to the top of the pile, and the second would be to bias the breaking of ties in favor of a chosen candidate.

8.3 Results

Despite the use of a verification-centric process, and 100% statement coverage of the code, the following issues are outstanding, representing a potential inconsistency in the JML specifications.

[1] Ballot signing is possible when anonymized ballots are made available for third-party inspection and counting. The lower preferences can be chosen in such a way as to identify the vote to a coercer. The defense is either not to reveal the ballot data or only to reveal the higher preferences which were exercised during the count [24].

8.3.1 ESC Warnings

Figure 8.1 lists the unresolved ESC warnings in each class. These are assumed to represent false negatives, unless confirmed by code inspection or by RAC, since ESC/Java2 is not sound.

For example the *Possible Negative Array Index* warning on line 220 of *AbstractBallotCouting.java* makes no sense given the precondition on line 212 which restricts the index parameter to non-negative values.

The warning *Code Not Checked* means that the assumptions are inconsistent, that the method might not terminate, *or* that the static analysis is imprecise. The warning *Method Not Checked* means that the prover limit (maximum number of verification conditions) was exceeded.

The number of lines of specification with unresolved ESC warnings in each class, per statement are shown in figure 8.2 and methods with unresolved warnings in figure 8.3. The count of specification methods includes Java methods with JML specifications, and also JML model methods. Coverage refers to the percentage of lines of specification, or methods of specification without unresolved warnings.

8.3.2 RAC failures

Figure 8.4 lists the unresolved RAC failures in each class. Unfortunately, the very first failure was in the constructor of the main class, so it may be hiding other RAC failures later on. [2]. This first RAC failure does not appear when the same RAC tests are run in debug mode using Eclipse, to discover which invariant had been violated, thus revealing the other RAC errors.

[2]The RAC failures can be seen by running *make cibuild* on the command line, at the root of the Vótáil project.

File	Method	Line	Warning
AbstractBallotCounting.java	isDepositSaved	220	Negative Array Index
AbstractBallotCounting.java	setup	277	Precondition
AbstractBallotCounting.java	load	301	Object Invariant
AbstractBallotCounting.java	allocateFirstPrefrences	347	Modifies Clause
AbstractBallotCounting.java	countBallotsFor	370	Loop Invariant
AbstractBallotCounting.java	countFirstPreferences	397	Loop Invariant
AbstractBallotCounting.java	getPotentialTransfers	426	Loop Invariant
AbstractBallotCounting.java	getNextContinuingPreference	464	Precondition
AbstractBallotCounting.java	load	308	Object Invariant
AbstractBallotCounting.java	allocateFirstPreferences	342	Loop Invariant
AbstractBallotCounting.java	allocateFirstPreferences	351	Postcondition
AbstractBallotCounting.java	getNextContinuingPreference	464	Precondition
AbstractBallotCounting.java	isContinuingCandidateID	491	Precondition
AbstractBallotCounting.java	isContinuingCandidateID	497	Postcondition
AbstractBallotCounting.java	getActualTransfers	551	Postcondition
AbstractBallotCounting.java	getTransferShortfall	608	Loop Invariant
AbstractBallotCounting.java	isHigherThan	713	Precondition
AbstractBallotCounting.java	compareCandidates	789	Precondition
AbstractBallotCounting.java	getTotalTransferableVotes	832	Loop Invariant
AbstractBallotCounting.java	findHighestCandidate	891	Loop Invariant
AbstractBallotCounting.java	findLowestCandidate	939	Loop Invariant
AbstractBallotCounting.java	eliminateCandidate	986	Precondition
AbstractBallotCounting.java	redistributeBallots	1008	Null Reference
AbstractBallotCounting.java	redistributeBallots	1009	Negative Array Index
AbstractBallotCounting.java	transferBallot	1033	Precondition
AbstractBallotCounting.java	electCandidate	1072	Precondition
AbstractBallotCounting.jml	getSumOfTransfers	366	Null Reference
AbstractBallotCounting.jml	getCandidateRanking	513	Array Index
AbstractBallotCounting.jml	getCandidateRanking	518	Precondition
AbstractBallotCounting.jml	numberTransferable	659	Null Reference
AbstractBallotCounting.jml	getOrder	693	Null Reference
BallotBox.java	accept	117	Precondition
Candidate.java	addVote	195	Precondition
Candidate.java	getCandidateRanking	n/a	Method Not Checked
Constituency.java	setNumberOfCandidates	102	Code Not Checked

Figure 8.1: Extract from the list of unresolved ESC warnings

Class	Lines with Warnings	Total	Coverage %
AbstractBallotCounting	28	292	90
AbstractCountStatus	0	10	100
Ballot	0	23	100
BallotBox	1	23	96
BallotCounting	0	64	100
Candidate	1	71	99
Constituency	1	35	97
ElectionStatus	0	7	100
Total for *election.tally* package	31	534	94

Figure 8.2: Unresolved ESC warnings by line for each JML specification class

Class	Methods with Warnings	Total	Coverage %
AbstractBallotCounting	26	54	50
Ballot	1	21	95
BallotBox	0	5	100
BallotCounting	0	19	100
Candidate	1	16	94
Constituency	1	8	87
AbstractCountStatus	0	4	100
ElectionStatus	0	3	100
***election.tally* package**	29	130	78

Figure 8.3: Unresolved ESC warnings by method for each JML specification class

Java Source File	Method	Line	Failures
BallotCounting	Default Constructor	41	Object Invariant
BallotCounting	removeNonTransferableBallots	126	Loop Invariant
BallotCounting	incrementCountNumber	400	Postcondition
Constituency	constructor	44	Object Invariant

Figure 8.4: Full List of RAC failures

The number of lines of code with unresolved RAC failures in the each class are shown in figure 8.5 and number of methods with unresolved RAC failures in figure 8.6. Four such failures are found, confirming the related ESC warnings.

Since there is no possibility of unsoundness in RAC, these failures indicate an (as yet) undiscovered inconsistency in the formal specifications.

This is the major limitation of the methodology chosen, despite having sufficient test cases and establishing the consistency of the Alloy model, there remains the possibility of a subtle mistake in the JML specifications, such as *over-specification* leading to false negatives.

Whereas Alloy predicates can be used to detect an over-specification, this can be harder to do in JML. One technique that was used before implementing the JML specification was to place *assert false* in each empty method body. Any such method that passed would be shown to have an inconsistent specification. Although this was done before implementation, it was clearly not sufficient.

At first glance, such an inconclusive result, might seem to undermine confidence in both the specification and implementation of Vótáil. Nevertheless each of the RAC failures can be shown to be false negative by code inspection.

If the same test cases were to be used for black box testing of a third-party implementation without a JML specification, and perhaps without source code, then it would still be possible to check the expected result for each set of test ballots as well as the overall code coverage. The main limitation of this approach is that the generated test ballots were stored in Vótáil specific format, rather than a standard format such as the Election Markup Language (EML) [69] which considerably increases the amount of software engineering effort required to test each system.

Nevertheless, in principle, there is now a complete set of test data, suitable for

Class	Lines with Failures	Total	Coverage %
AbstractBallotCounting	0	306	100
AbstractCountStatus	0	20	100
Ballot	0	52	100
BallotBox	0	29	100
BallotCounting	3	191	98
Candidate	0	81	100
Constituency.java	1	47	98
CandidateStatus.java	0	7	100
CountConfiguration	0	5	100
ElectionStatus.java	0	11	100
***election.tally* package**	4	749	99

Figure 8.5: Unresolved RAC failures by line for each Java class

testing any PR-STV implementation, and more importantly a means of updating the model behind this data and regenerating the test data, if required.

Vótáil also offers a large and interesting JML specification against which new JML tools can be deployed and tested, knowing that there are potential specification errors waiting to be discovered.

8.4 Contribution

This is the first complete formal specification of the Irish PR-STV ballot counting procedure. The requirements are traceable from the legislation, through BON and JML to the Java code. The specifications and source code are publicly available for comment and criticism. There are no other publicly available works of this kind.

PR-STV is one of the most complex voting systems in use today, particularly with regards to formal specification and verification. It is also one of the most complex which can be implemented and understood using paper ballots.

Counting of ballots is only one facet of the entire process, but is a critical

Class	RAC Failed Methods	Total	Coverage %
AbstractBallotCounting	0	34	100
BallotCounting	3	19	84
Candidate	0	14	100
Ballot	0	9	100
Constituency	1	8	87
BallotBox	0	5	100
CountConfiguration	n/a	0	n/a
ElectionStatus	n/a	0	n/a
CandidateStatus	n/a	0	n/a
AbstractCountStatus	0	1	100
***election.tally* package**	4	90	96

Figure 8.6: Unresolved RAC failures by method for each Java class

component.

Even in simple first-past-the-post counting it is not uncommon to discover computer tally errors, for example in the spreadsheets used for the Maine Caucus[3]. If this is the case for simple plurality voting, the scope for errors in PR-STV are even greater.

There appears to be no valid reason why a software system for counting of votes should contain any errors whatever, provided the software has been rigorously verified, and also installed in a secure tamper-proof manner in the machines.

Finally, a discrepancy between the paper count and electronic count does not *always* mean that the electronic count is wrong.

[3] `http://www.bbvforums.org/cgi-bin/forums/show.cgi?8/81922`

Bibliography

[1] J. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge Univ Pr, 2010.

[2] J. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for event-b. *Formal Methods and Software Engineering*, pages 588–605, 2006.

[3] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, et al. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[4] A. Appel. How to defeat rivests threeballot voting system. *Manuskrypt, pazdziernik*, 2006.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of ICSE'04*. IEEE Computer Society, 2004.

[6] S. Bowler and B. Grofman. *Elections in Australia, Ireland, and Malta under the Single Transferable Vote: Reflections on an embedded institution*. University of Michigan Press, 2000.

[7] R. R. Brooks and J. Deng. Lies and the lying liars that tell them: a fair and balanced look at TLS. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 59:1–59:3, New York, NY, USA, 2010. ACM.

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[9] D. Castro. Stop the presses: How paper trails fail to secure e-voting. *Information Technology and Innovation Foundation, Washington, DC, September*, 2007.

[10] O. Cetinkaya. Analysis of security requirements for cryptographic voting protocols. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 1451–1456. IEEE, 2008.

[11] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer–Verlag, 2006.

[12] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 1789–1901, 2002.

[13] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of*

the 16th European Conference on Object-Oriented Programming (ECOOP 2002), volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer–Verlag, June 2002.

[14] D. Cochran. Secure internet voting in Ireland using the Open Source Kiezen op Afstand (KOA) remote voting system. Master's thesis, University College Dublin, Mar. 2006.

[15] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of ICSE'03*. IEEE Computer Society, 2003.

[16] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362, pages 108–128. Springer, 2004.

[17] L. Coyle, P. Cunnigham, and D. Doyle. Appendix 2D - second report of commission on electronic voting in Ireland: Secrecy, accuracy and testing of the chosen electronic voting system: Reliability and accuracy of data inputs and outputs, Dec. 2004.

[18] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of ICSE'99*. IEEE Computer Society, 1999.

[19] G. Del Castillo. The ASM workbench: A tool environment for computer-aided analysis and validation of abstract state machine models. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 578–581. Springer Berlin / Heidelberg, 2001.

[20] G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. *Verified Software: Theories, Tools, Experiments*, pages 130–145, 2008.

[21] Department of Environment and Local Government, Commission on Electronic Voting. Count requirements and commentary on count rules, update no. 7: Available surpluses and candidates with zero votes, 14 April 2002.

[22] Department of Environment and Local Government, Commission on Electronic Voting. Final Report of Commission on Electronic Voting, July 2006.

[23] Department of Environment and Local Government, Commission on Electronic Voting in Ireland. Count requirements and commentary on count rules, 23 June 2000.

[24] R. Di Cosmo. On privacy and anonymity in electronic and non electronic voting: the ballot-as-signature attack. 2007.

[25] E. Dijkstra. *Structured programming*. Yourdon Press, 1979.

[26] D. Dill, R. Mercuri, P. Neumann, and D. Wallach. *Frequently Asked Questions about DRE Voting Systems*. Feb, 2003.

[27] Y. Espelid, L. Netland, A. Klingsheim, and K. Hole. A proof of concept attack against norwegian internet banking systems. *Financial Cryptography and Data Security*, pages 197–201, 2008.

[28] D. Farrell and I. McAllister. *The Australian electoral system: origins, variations, and consequences*. New South Wales University Press, Ltd., 2006.

[29] L. Fredman. *The Australian ballot: The story of an American reform*. Michigan State University Press East Lansing, MI, 1968.

[30] M. Gallagher. Comparing proportional representation electoral systems: Quotas, thresholds, paradoxes and majorities. *British Journal of Political Science*, 22(4):469–496, 1992.

[31] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[32] J. Gilmour. Detailed description of the STV count in accordance with the rules in the Scottish local government elections order 2007. *Representation*, 43(3):217–229, 2007.

[33] S. Goggin and M. Byrne. An examination of the auditability of voter verified paper audit trail (VVPAT) ballots. In *Proc. USENIX/ACCURATE Electronic Voting Technology Workshop (EVT)*, 2007.

[34] S. Goggin, M. Byrne, J. Gilbert, G. Rogers, and J. McClendon. Comparing the auditability of optical scan, voter verified paper audit trail (VVPAT) and video (VVVAT) ballot systems. In *Proceedings of the conference on Electronic voting technology*, page 9. USENIX Association, 2008.

[35] J. Heckelman. Bribing voters without verification. *The Social Science Journal*, 35(3):435–443, 1998.

[36] H. Hisamitsu and K. Takeda. The security analysis of e-voting in japan. In *Proceedings of the 1st international conference on E-voting and identity*, VOTE-ID'07, pages 99–110, Berlin, Heidelberg, 2007. Springer-Verlag.

[37] S. Hogan and R. Cochran. Electronic voting in Ireland, a threat to democracy. *Report prepared for the Irish Labour Parliamentary Party*, 2003.

[38] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of ICSE'03*. IEEE Computer Society, 2003.

[39] D. Jackson. *Software Abstractions: logic, language and analysis*. MIT Press (MA), 2012.

[40] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 14–25. ACM, 2000.

[41] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *6th International Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Dubrovnik, Croatia, 2007. Workshop at ESEC/FSE 2007.

[42] D. Jefferson, A. Rubin, B. Simons, and D. Wagner. Analyzing internet voting security. *Communications of the ACM*, 47(10):59–64, 2004.

[43] C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*, pages 3–3. USENIX Association, 2005.

[44] J. Kiniry, D. Cochran, and P. Tierney. Verification-centric realization of electronic vote counting. In *Proceedings of the USENIX/Accurate Electronic Voting Technology on USENIX/Accurate Electronic Voting Technology Workshop*. USENIX Association Berkeley, CA, USA, 2007.

[45] J. Kiniry and A. Morkan. Soundness and completeness warnings in ESC/-Java2. In *5th International Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*, Portland, Oregon, 2006.

[46] J. Kiniry, A. Morkan, D. Cochran, F. Fairmichael, P. Chalin, M. Oostdijk, and E. Hubbers. The KOA remote voting system: A summary of work to date. In *Proceedings of Trustworthy Global Computing*, 2006.

[47] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, Jan. 2005.

[48] J. Kitcat and I. Brown. Observing the English and Scottish 2007 e-elections. *Parliamentary Affairs*, 61(2):380–395, 2008.

[49] O. Kjölbro. Verifying the Danish Voting System. Master's thesis, IT University of Copenhagen, May 2011.

[50] T. Kohno, A. Stubblefield, A. Rubin, and D. Wallach. Analysis of an electronic voting system. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 27–40. Ieee, 2004.

[51] T. Lauer. The risk of e-voting. *Electronic Journal of E-government*, 2(3):177–186, 2004.

[52] G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. *Kluwer International Series in Engineering and Computer Science*, pages 175–188, 1999.

[53] LogicaCMG. Kiezen op Afstand: Hertellen Stemmen. Functional specifications, 2004.

[54] M. McGaley and J. Gibson. Electronic voting: A safety critical system. *Final Year Project Report, NUI Maynooth Department of Computer Science*, 2003.

[55] M. Meagher. Towards the development of an electronic count system using formal methods, 2001. MPhil thesis, University of Southampton.

[56] R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication–or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, 2006.

[57] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

[58] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. *Automated DeductionCADE-11*, pages 748–752, 1992.

[59] R. Posner. Florida 2000: A legal and statistical analylsis of the election deadlock and the ensuing litigation. *Sup. Ct. Rev.*, page 1, 2000.

[60] J. Pujol-Ahulló, R. Jardí-Cedó, and J. Castellà-Roca. Verification systems for electronic voting: A survey. *Gesellschaft für Informatik (GI)*, page 163.

[61] A. Rubin. Security considerations for remote electronic voting. *Communications of the ACM*, 45(12):39–44, 2002.

[62] P. Ryan. Verified encrypted paper audit trails. *Technical Report Series - University of Newcastle upon Tyne, Computing Science*, 966, 2006.

[63] M. Said, M. Butler, and C. Snook. Class and state machine refinement in UML-B. 2009.

[64] D. Sandler, K. Derr, and D. Wallach. Votebox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th conference on Security symposium*, pages 349–364. USENIX Association, 2008.

[65] D. Sandler, K. Derr, and D. S. Wallach. Votebox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 349–364, Berkeley, CA, USA, 2008. USENIX Association.

[66] G. Schryen and E. Rich. Security in large-scale internet elections: a retrospective analysis of elections in Estonia, the Netherlands, and Switzerland. *Trans. Info. For. Sec.*, 4(4):729–744, Dec. 2009.

[67] M. Shamos. Paper v. electronic voting records-an assessment. In *Proceedings of the 14th ACM Conference on Computers, Freedom and Privacy*, 2004.

[68] C. Snook, V. Savicks, and M. Butler. Verification of UML models by translation to UML-B. *Lecture Notes in Computer Science*, 6957:251, 2011.

[69] P. Spencer and B. Consulting. The election markup language. *XML Europe 2003*, 2003.

[70] A. Villafiorita, K. Weldemariam, and R. Tiella. Development, formal verification, and evaluation of an e-voting system with VVPAT. *Information Forensics and Security, IEEE Transactions on*, 4(4):651–661, 2009.

[71] K. Waldén and J.-M. Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. The Object-Oriented Series. Prentice–Hall, Inc., 1995.

Appendix A

BON Design Specification

A.1 Primitive Types

```
cluster_chart BON_TYPES
explanation
  "Standard built-in and primitive classes reusable in all BON designs"
class BOOLEAN description "A binary true or false value"
class NATURAL_NUMBER description "Non-negative whole number"
class REAL_NUMBER description "Any measurable value"
class STRING description "A sequence of text characters"
class SET description "A collection of objects of the same type"
class VALUE description "Abstract number, quantity or index"
end

class_chart BOOLEAN
indexing
  proposed_refinement: "java.lang.Boolean"
explanation
  "A binary true or false value"
inherit VALUE
end
```

```
class_chart NATURAL_NUMBER
explanation
  "Any non-negative whole number"
inherit VALUE
query
  "What is my value?",
  "Is my value higher than another natural number?",
  "Is my value equal to another natural number?"
command
  "Change my value to this natural number"
constraint
  "My initial value is zero",
  "My value is greater than or equal to zero",
  "My value is a whole number"
end

class_chart REAL_NUMBER
explanation
  "Any measurable value"
inherit VALUE
end

class_chart SET
explanation
  "Collection of objects of the same type"
end

class_chart STRING
  explanation "Sequence of text characters"
end
```

A.2 System Chart

```
system_chart VOTAIL
explanation
  "A verified verifiable open source system for remote
   electronic or computer mediated voting"
cluster COUNTING_PROCESS description "The counting of
  ballots by proportional representation with a single
  transferable vote"
end
```

A.3 Cluster Charts

```
cluster_chart COUNTING_PROCESS
explanation
  "The counting of electronic ballots"
class BALLOT description
  "Ranked list of candidates"
class BALLOT_BOX description
  "The set of all valid ballots"
class BALLOT_COUNTING description
  "The algorithm used to count the ballots"
class CANDIDATE description
  "A person eligible and nominated for election"
class CONSTITUENCY description
  "The location and size of the election"
end
```

A.4 Class Charts

```
-- Votail is designed to be used as a
```

```
-- sub-system for counting within
-- either an optical ballot scan system
-- or a remote online voting system
-- that supplies a valid set of ballots
-- to be counted and that takes care of
-- system level issues such as security,
-- authentication and data storage·

class_chart BALLOT
explanation
  "A valid set of preferences such as an
    ordered list of candidates"
query
  "How many preferences are shown on this ballot?",
  "Who is the first preference for?",
  "Who is the next preference for?",
  "Who is the highest preference continuing
    candidate on this ballot?",
  "Which elected or continuing candidate benefits
    from this ballot?"
command
   "Allocate this ballot to the next preference candidate"
constraint
  "The sequence of preferences is unbroken e·g·
  no candidate is listed twice",
 end

class_chart BALLOT_BOX
explanation
  "The set of all ballot papers with at least one valid preference"
query
  "How many valid ballot papers?",
  "How many first preference votes for each candidate?",
```

```
  "How many different piles of ballots?"
constraint
  "Every ballot paper has at least one preference",
  "The total number of first preference votes is equal to the
   total number of non-spoilt ballots",
  "The sum of the number of ballots in each pile is equal to the
   total number of ballots"
end

class_chart BALLOT_COUNTING
explanation
  "Count algorithm for tallying of the votes in Dail elections"
query
  "How many continuing candidates?",
  "How many remaining seats?",
  "What is the quota?",
  "Who is/are highest continuing candidate(s) with a surplus?",
  "What is the surplus?",
  "What is the transfer factor?"
command
  "Distribute the surplus ballots",
  "Select lowest continuing candidates for exclusion",
  "Declare remaining candidates elected",
  "Close the count"
end

class_chart CANDIDATE
explanation
  "A person eligible and nominated for election"
query
  "How many first preferences?",
  "How many second preferences?",
  "How many votes in each round?"
```

```
end

class_chart CONSTITUENCY
explanation
  "The overall parameters of the election"
query
  "How many candidates in this election?",
  "How many seats are in this constituency?",
  "How many seats are being filled in this election?"
constraint
  "At least one seat is being filled",
  "The number of seats being filled is not
     more than the number of
     seats in this constituency",
  "The number of seats being filled is
    less than the number of candidates"
end

class_chart SCENARIO
explanation
  "A set of candidate outcomes"
command
  "Add a candidate outcome to this scenario"
end

class_chart SCENARIO_LIST
explanation
  "The complete set of all scenarios"
command
  "Replay from a stored file"
  "Write to a stored file"
end
```

A.5 Event Chart

```
-- Internal (outgoing) events
-- See diagrams/DailBallotCounting

event_chart VOTAIL
outgoing
part
  "2/2"
event
  "A: The quota is calculated"
involves
  BALLOT_BOX, BALLOT_COUNTING
event
  "B: The highest continuing candidate is found"
involves
  BALLOT_COUNTING
event
  "C: The surplus, from an elected candidate, is calculated"
involves
  BALLOT_COUNTING
event
  "D: The number of votes for transfer (from surplus) are calculated"
involves
  BALLOT_COUNTING
event
   "H: Transfers, from an excluded candidate, are calculated"
involves
   BALLOT_COUNTING
event
   "J: The lowest continuing candidate is selected for exclusion"
involves
   BALLOT_COUNTING
```

```
event
   "K: The number of continuing candidates is recalculated"
involves
   BALLOT_COUNTING, CANDIDATE
event
   "L: The selected ballots are moved to the next continuing
  preference candidate"
involves
   BALLOT, BALLOT_COUNTING
event
   "M: The number of remaining seats is checked"
involves
   BALLOT_COUNTING
event
   "N: The remaining candidates are deemed elected"
involves
   BALLOT_COUNTING, CANDIDATE
event
   "P: The counting is closed and final result declared"
involves
   BALLOT_COUNTING
end
```

A.6 Scenario Chart

```
-- See diagrams/DailBallotCounting

scenario_chart VOTAIL
explanation
  "Scenarios for ballot counting"
scenario
  "01: READY_TO_START"
```

```
description
  "Ballots, candidate and election data have been loaded"
scenario
  "02: NO_SEATS_FILLED_YET"
description
  "No candidates have been elected so far"
scenario
  "03: CANDIDATE_REACHES_QUOTA"
description
  "One or more candidates achieve the minimum quota of votes"
scenario
  "04: CANDIDATE_ELECTED"
description
  "The highest continuing candidate with quota is deemed to be elected"
scenario
  "05: NO_SURPLUS_AVAILABLE"
description
  "There is no surplus available for distribution"
scenario
  "06: SURPLUS_AVAILABLE"
description
  "There is at least one surplus available for distribution"
scenario
  "10: READY_TO_MOVE_BALLOTS"
description
  "The number of ballots for transfer to each candidate has been determined"
scenario
  "11: LOWEST_CANDIDATE_EXCLUDED"
description
  "Exclude one or more of the lowest continuing candidates"
scenario
  "12: READY_FOR_NEXT_ROUND"
description
```

```
  "Ballots have been transfered from one candidate"
scenario
  "13: FILLING_OF_LAST_SEAT"
description
  "There is one remaining seat and at least two continuing
  candidates, or the number of continuing candidates is just
  one more than the number of remaining seats"
scenario
  "14: MORE_CANDIDATES_THAN_SEATS"
description
  "There are more continuing candidates than remaining seats"
scenario
  "15: SEATS_REMAINING"
description
  "There is at least one vacancy yet to be filled"
scenario
  "16: ALL_SEATS_FILLED"
description
  "The number of continuing candidates equals the
  number of seats remaining"
scenario
  "18: CONTINUING_CANDIDATES_EQUAL_REMAINING_SEATS"
description
  "The number of continuing candidates equals the number
  of remaining seats; all continuing candidates are deemed
  elected"
end
```

Appendix B

JML Design Specification

B.1 Abstract Ballot Counting Class

```
package election.tally;

/**
 * This is a Java Modeling Language (JML) design specification of the ballot
 * counting algorithm for elections to Dail Eireann - the lower house of the
 * National Parliament of Ireland.
 *
 * @see <a href="http://www.irishstatutebook.ie/1992_23.html">Part XIX of
 * the Electoral Act, 1992</a>
 * @see Requirements for this specification are taken from
 * <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">Department of
 * Environment and Local Government, Commentary on Count
 * Rules, sections 3-16, pages 12-65</a>
 * @see <a href="http://www.jmlspecs.org/">JML Homepage</a>
 */

public abstract class AbstractBallotCounting extends ElectionStatus {
  /** List of details for each candidate.
   * There are no duplicates in the list of candidate IDs and,
   * once the counting starts, there must be a ballot paper associated with
   * each vote held by a candidate. */
```

```
public invariant (PRECOUNT < state) implies
  \nonnullelements (candidateList);
public model Candidate[] candidateList;

public invariant (PRELOAD <= state) implies
  (candidateList ≠ null) &&
  (\forall int i; 0 <= i && i < totalCandidates;
    candidateList[i] ≠ null);
public invariant (PRELOAD <= state) implies
  (totalCandidates == candidateList·length);
public invariant (state == PRELOAD || state == LOADING) implies
  (\forall int i; 0 <= i && i < totalCandidates;
  candidateList[i]·getStatus() == Candidate·CONTINUING &&
  candidateList[i]·getTotalVote() == 0);
public invariant (state == PRELOAD || state == LOADING ||
  state == PRECOUNT || state == COUNTING || state == FINISHED) implies
  (\forall int i, j; 0 <= i && i < totalCandidates &&
  i < j && j < totalCandidates;
    candidateList[i]·getCandidateID() ≠
    candidateList[j]·getCandidateID());
protected invariant (state == COUNTING || state == FINISHED) implies
  (\forall int i; 0 <= i && i < totalCandidates;
  candidateList[i]·getTotalAtCount() ==
  countBallotsFor (candidateList[i]·getCandidateID()));

/** List of contents of each ballot paper that will be counted· */
public model non_null election·tally·Ballot[] ballotsToCount;

/** Total number of candidates for election */
public model int totalCandidates;
public invariant 0 <= totalCandidates;
public invariant (PRECOUNT <= state) implies
  totalCandidates <= candidateList·length;
public invariant numberElected + numberEliminated <= totalCandidates;

/** Number of candidates elected so far */
public model int numberElected;
```

```
public invariant 0 <= numberElected;
public invariant numberElected <= seats;
public invariant (state <= PRECOUNT) implies numberElected == 0;
protected invariant (COUNTING <= state) implies
  numberElected == (\num_of int i; 0 <= i && i < totalCandidates;
  isElected(candidateList[i]));
public invariant (state == FINISHED) implies numberElected == seats;
public constraint \old(numberElected) <= numberElected;

/** Number of candidates excluded from election so far */
public model int numberEliminated;
public invariant 0 <= numberEliminated;
public invariant (PRECOUNT <= state) implies
  seats + numberEliminated <= totalCandidates;
public invariant (state == COUNTING || state == FINISHED) implies
  numberEliminated == (\num_of int i; 0 <= i && i < totalCandidates;
  candidateList[i].getStatus() == Candidate.ELIMINATED);

/** Number of seats to be filled in this election */
public model int seats;
public invariant 0 <= seats;
public invariant seats <= totalSeats;
public constraint (PRELOAD < state) implies seats == \old (seats);
public invariant (state == COUNTING) implies (1 <= seats);

/** Total number of seats in this constituency.
 *   The constitution and laws of Ireland do not allow less than three or
 *   more than five seats in each Dail constituency, but this could change in
 *   future and is not an essential part of the specification.
 */
public model int totalSeats;
public invariant 0 <= totalSeats;
public constraint (LOADING <= state) implies totalSeats == \old (totalSeats);

/** Total number of valid votes in this election count
 *  There is a theoretical maximum number of votes because there
 *  must be at least one seat for every thirty thousand electors, and
```

```
   *  therefore a current maximum of 150,000 votes in a five seater
   *  constituency.  The Java primitive type <code>int</code> has a
   *  maximum value of over two billion which should be sufficient for
   *  anything less than a worldwide election.
   */
  public model int totalVotes;
  public invariant 0 <= totalVotes;
  public invariant (LOADING < state) implies
    (totalVotes == ballotsToCount.length);
  public invariant (state < LOADING) implies totalVotes == 0;
  public constraint (state == LOADING) implies
    (\old (totalVotes) <= totalVotes);
  public constraint (LOADING < state) implies
    (totalVotes == \old (totalVotes));

  /** Minimum number of votes needed to save deposit unless elected */
  public model int depositSavingThreshold;
  public invariant 0 <= depositSavingThreshold;
  public invariant depositSavingThreshold <= totalVotes;
  /** @see requirement 6, section 3, item 3, page 13 */
  invariant (PRECOUNT < state) implies
    (depositSavingThreshold == ((getQuota() / 4) + 1));

  /** Number of rounds of counting so far */
  public model int countNumber;
  public initially countNumber == 0;
  public invariant 0 <= countNumber;
  public invariant (PRELOAD <= state) implies
    countNumber <= CountConfiguration.MAXCOUNT;
  public constraint (state == COUNTING) implies
    \old(countNumber) <= countNumber;

  /** Number of seats that remain to be filled */
  public model int remainingSeats;
  public invariant 0 <= remainingSeats;
  public invariant remainingSeats <= seats;
  public invariant (state <= PRECOUNT) implies
```

```
    remainingSeats == seats;
  public invariant (state == FINISHED) implies
    remainingSeats == 0;
  public invariant (state == COUNTING) implies
    remainingSeats == (seats - numberElected);

  /**
   * Determine if the candidate has received enough votes for election
   *
   * @param candidate This candidate
   * @return True if the candidate has at least a quota of votes
   * @see "http://www·cev·ie/htm/tenders/pdf/1_1·pdf, page 79, paragraph 120 (2)"
   */
  ensures \result == (countBallotsFor(candidate·getCandidateID()) ≥
getQuota());
  public pure boolean hasQuota (election·tally·Candidate candidate);

  /**
   * Determine if the candidate has been elected
   *
   *  It is possible for a candidate without having reached the quota
   *   to be elected in the final round of counting by virtue of being the
   *   highest continuing candidate when one seat remains·
   *
   * @see requirement 4, section 13, item 3, page 13
   *
   *
   *  The quota is the minimum number of votes needed for election, except
   * when any of the following shortcuts apply·
   * <ul>
   * <li>The number of continuing candidates is equal to the number of remaining
   * seats·
   * <li>The number of continuing candidates is one more than the number of
   * remaining seats·
   * <li>There is one remaining seat·
   * </ul>
   *
```

```
 * @see <a href="http://www.cev.ie/htm/tenders/pdf/1_1.pdf">
 * Department of Environment and Local Government,
 * Electronic Vote Counting System, Appendix E</a>
 * @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">
 * Department of Environment and Local Government,
 * Count Requirements and Commentary on Count Rules, sections 3-14</a>
 * @see <a href="http://www.irishstatutebook.ie/1992_23.html">
 * Sections 48, 118 and 124 of the Electoral Act, 1992</a>
 *
 * @param candidate This candidate
 * @return True if the candidate has been elected
 */
requires candidate ≠ null;
requires countNumber ≥ 1;
requires state == COUNTING;
ensures (\result == ((candidate.getStatus() == Candidate.ELECTED ||
  hasQuota(candidate)));
public pure boolean isElected (election.tally.Candidate candidate);

/**
 * Determine how many surplus votes a candidate has.
 *
 * The surplus is the maximum number of votes available for transfer
 * @see requirement 5, section 3, item 3, page 13
 *
 * @param candidate The candidate record
 * @return The undistributed surplus for that candidate, or zero if the
 * candidate has less than a quota of votes
 */
public pure int getSurplus (election.tally.Candidate candidate);

/**
 * Determines if a candidate has saved his or her deposit.
 *
 *  The deposit saving threshold is one plus one quarter of the
 *  full quota. It is possible for a candidate without the deposit
 *  saving threshold to be elected in the final round of counting by
```

```
   *  virtue of being the highest continuing candidate when one seat
   *  remains. This could happen, for example, if the majority of ballots
   *  contained only first preferences.
   *
   * @see requirement 7, section 13, item 3, page 13
   *
   * @param candidate This candidate
   * @return <code>true</code> if candidate has had enough votes to
   * save deposit or has been elected
   */
  requires (state == COUNTING) || (state == FINISHED);
  ensures \result ==
    ((candidateList[index].getTotalVote() ≥ depositSavingThreshold) ||
    (isElected (candidateList[index]) == true));
  public pure boolean isDepositSaved (int index);

  /**
   * Distribution of surplus votes
   *
   * @param candidateWithSurplus The candidate whose surplus is to be distributed
   *  The highest surplus must be distributed if the total surplus
   *   could save the deposit of a candidate or change the relative position
   *   of the two lowest continuing candidates, or would be enough to elect the
   *   highest continuing candidate.
   * @see requirements 14-18, section 5, item 2, page 18
   * @see requirement 8, section 4, item 2, page 15
   */
  requires isElected (candidateList[candidateWithSurplus])
    && getSurplus (candidateList[candidateWithSurplus]) > 0;
  requires state == COUNTING;
  requires getNumberContinuing() > remainingSeats;
  ensures getSurplus (candidateList[candidateWithSurplus]) == 0;
  public abstract void distributeSurplus(int candidateWithSurplus);

  /**
   * Set up candidate details and number of seats
   */
```

```
requires state == EMPTY;
ensures state == PRELOAD;
ensures totalCandidates == constituency·getNumberOfCandidates();
ensures seats == constituency·getNumberOfSeatsInThisElection();
ensures totalSeats == constituency·getTotalNumberOfSeats();
public void setup (Constituency constituency);

/** Loads all valid ballot papers·
 *
 *  @param ballotBox The set of ballots for counting
 *
 *  All ballot papers must be assigned to a valid candidate ID
 */
requires state == PRELOAD;
ensures state == PRECOUNT;
ensures totalVotes == ballotBox·numberOfBallots;
ensures (\forall int i; 0 <= i && i < totalVotes;
  (\exists int j; 0 <= j && j < totalCandidates;
  ballotsToCount[j]·isAssignedTo(candidateList[i]·getCandidateID())));
public void load(BallotBox ballotBox);

/**
 * Count the votes·
 *
 *  This is the method that starts the counting process·
 * @see requirement 1, section 3, item 2, page 12
 */
public normal_behavior
  requires state == PRECOUNT || state == COUNTING;
  assignable state, countNumber, numberElected, remainingSeats,
    candidateList, ballotsToCount;
  ensures remainingSeats == 0;
  ensures numberElected == seats;
  ensures state == FINISHED;
public abstract void count();

/**
```

```
   * Get the status of the algorithm in progress
   */
  ensures \result == state;
  public model pure byte getStatus();

  /**
   * Gets the potential number of transfers from one candidate to another·
   *
   *  This method is needed to get the proportions to use when transferring
   * surplus votes·  If the candidate was elected on the first count then all
   * votes are examined, otherwise only the last set of votes received are
   * examined·
   *
   * @see requirement 19, section 7, item 2, page 23
   *
   * @param fromCandidate Candidate ID from which to check the transfers
   * @param toCandidateID Candidate ID to check for receipt of transfered votes
   * @return Number of votes potentially transferable from this candidate to that
   *   candidate
   */
  ensures \result== (\num_of int j; 0 <= j && j < totalVotes;
    (ballotsToCount[j]·isAssignedTo(fromCandidate·getCandidateID())) &&
    (getNextContinuingPreference(ballotsToCount[j]) == toCandidateID));
  protected pure spec_public int getPotentialTransfers (
    non_null Candidate fromCandidate, int toCandidateID);

  /**
   * Get the maximum number of votes transferable to continuing candidates·
   *
   * @see requirement 20, section 7, item 2, page 24
   *
   * @param fromCandidate Candidate from which to check the transfers
   * @return Number of votes potentially transferable from this candidate
   */
  requires state == COUNTING;
  protected pure spec_public int getTotalTransferableVotes (
    non_null Candidate fromCandidate);
```

```
requires state == COUNTING;
public model pure int getSumOfTransfers(
  non_null Candidate fromCandidate) {

  int sum = 0;
  loop_invariant (0 < i) implies
    (sum == (\sum int j; 0 <= j && j < i;
    getPotentialTransfers (fromCandidate,
    candidateList[j]·getCandidateID())));
  for (int i=0; i < totalCandidates; i++ ) {
    sum += getPotentialTransfers (fromCandidate,
      candidateList[i]·getCandidateID());
      }
    return sum;
  }

/**
 * Gets the next preference continuing candidate·
 *
 * This is the _nearest_ next preference i·e·
 * filter the list of preferences to contain continuing candidates
 * and then get the next preference to a continuing candidate,
 * if any·
 *
 * @param ballot Ballot paper from which to get the next preference
 *
 * @return Candidate ID of next continuing candidate or NONTRANSFERABLE
 */
requires state == COUNTING;
ensures (\result == Ballot·NONTRANSFERABLE) ||
  (\exists int k; 1 <= k && k <= ballot·remainingPreferences();
  \result == ballot·getNextPreference(k));
ensures (isContinuingCandidateID(\result)) ||
  (\result == Ballot·NONTRANSFERABLE);
protected pure spec_public int getNextContinuingPreference
  (non_null Ballot ballot);
```

```
/**
 * Determine actual number of votes to transfer to this candidate, excluding
 * rounding up of fractional transfers
 *
 * @see requirement 21, section 7, item 3·1, page 24
 * @see requirement 22, section 7, item 3·2, page 25
 *
 *  The votes in a surplus are transfered in proportion to
 *   the number of transfers available throughout the candidates ballot stack·
 *   If not all transferable votes are accounted for the highest remainders
 *   for each continuing candidate need to be examined·
 *
 * @param fromCandidate Candidate from which to count the transfers
 * @param toCandidate Continuing candidate eligible to receive votes
 * @return Number of votes to be transfered, excluding fractional transfers
 */
requires state == COUNTING;
requires isElected (fromCandidate) ||
  (fromCandidate·getStatus() == Candidate·ELIMINATED);
requires toCandidate·getStatus() == Candidate·CONTINUING;
ensures ((fromCandidate·getStatus() == Candidate·ELECTED) &&
  (getSurplus(fromCandidate) < getTotalTransferableVotes(fromCandidate)))
  implies (\result == (getSurplus (fromCandidate) *
  getPotentialTransfers (fromCandidate, toCandidate·getCandidateID()) /
  getTotalTransferableVotes (fromCandidate)));
ensures ((fromCandidate·getStatus() == Candidate·ELIMINATED) ||
  (getTotalTransferableVotes(fromCandidate) <= getSurplus (fromCandidate)))
  implies (\result == (\num_of int j; 0 <= j && j < totalVotes;
  ballotsToCount[j]·isAssignedTo(fromCandidate·getCandidateID()) &&
  getNextContinuingPreference(ballotsToCount[j]) ==
  toCandidate·getCandidateID()));
protected pure spec_public int getActualTransfers (
  non_null Candidate fromCandidate, non_null Candidate toCandidate);

/**
 * Determine the rounded value of a fractional transfer·
```

```
 *
 *  This depends on the shortfall and the relative size of the
 * other fractional transfers·
 *
 * @see requirements 23-25, section 7, item 3·2, page 25
 *
 * @param fromCandidate
 *        Elected candidate from which to distribute surplus
 *
 * @param toCandidate
 *        Continuing candidate potentially eligible to receive transfers
 *
 * @return
 *     <code>1</code> if the fractional vote is to be rounded up
 *     <code>0</code> if the fractional vote is to be rounded down
 */
requires state == COUNTING;
requires isElected (fromCandidate);
requires toCandidate·getStatus() == election·tally·Candidate·CONTINUING;
requires getSurplus(fromCandidate) <
  getTotalTransferableVotes(fromCandidate);
ensures (getOrder (fromCandidate, toCandidate) <=
  getTransferShortfall (fromCandidate)) implies \result == 1;
ensures (getOrder (fromCandidate, toCandidate) >
  getTransferShortfall (fromCandidate)) implies \result == 0;
protected pure spec_public int getRoundedFractionalValue (
  non_null Candidate fromCandidate, non_null Candidate toCandidate);

/**
 * Determine the number of continuing candidates with a higher
 * remainder in their transfer quotient, or deemed to have a higher
 * remainder·
 *
 *  There must be a strict ordering of candidates for the purpose of
 * allocating the transfer shortfall·  If two or more candidates have
 * equal remainders then use the number of transfers they are about
 * to receive and if the number of transfers are equal then look at the
```

```
   * number of votes already received·
   *
   * @param fromCandidate
   *        Elected candidate from which to distribute surplus
   *
   * @param toCandidate
   *        Continuing candidate potentially eligible to receive transfers
   *
   * @return The number of continuing candidates with a higher
   *   quotient remainder than this candidate
   */
  requires state == COUNTING;
  requires isElected (fromCandidate);
  requires toCandidate·getStatus() == CandidateStatus·CONTINUING;
  requires getSurplus(fromCandidate) <
    getTotalTransferableVotes(fromCandidate);
  ensures \result == getCandidateRanking (fromCandidate, toCandidate);
  protected pure spec_public int getOrder(
    final non_null Candidate fromCandidate,
    final non_null Candidate toCandidate);

  ensures \result == (\num_of int i; i <= 0 && i < totalCandidates &&
    candidateList[i]·getCandidateID() ≠ toCandidate·getCandidateID() &&
    candidateList[i]·getStatus() == election·tally·Candidate·CONTINUING;
    (getTransferRemainder(fromCandidate, candidateList[i]) >
    getTransferRemainder(fromCandidate, toCandidate)) ||
    ((getTransferRemainder(fromCandidate, candidateList[i]) ==
    getTransferRemainder(fromCandidate, toCandidate)) &&
    (getActualTransfers(fromCandidate, candidateList[i]) >
    getActualTransfers(fromCandidate, toCandidate))) ||
    ((((getTransferRemainder(fromCandidate, candidateList[i]) ==
    getTransferRemainder(fromCandidate, toCandidate)) &&
    (getActualTransfers(fromCandidate, candidateList[i]) ==
    getActualTransfers(fromCandidate, toCandidate)))) &&
    isHigherThan (candidateList[i], toCandidate)));
   public pure model int getCandidateRanking (
     Candidate fromCandidate, Candidate toCandidate) {
```

```
    int counter = 0;
    for (int i=0; i < totalCandidates; i++ ) {
      if (candidateList[i].getCandidateID() ≠
        toCandidate.getCandidateID() &&
        candidateList[i].getStatus() ==
        election.tally.Candidate.CONTINUING &&
        ((getTransferRemainder(fromCandidate, candidateList[i]) >
        getTransferRemainder(fromCandidate, toCandidate)) ||
        ((getTransferRemainder(fromCandidate, candidateList[i]) ==
        getTransferRemainder(fromCandidate, toCandidate)) &&
        (getActualTransfers(fromCandidate, candidateList[i]) >
        getActualTransfers(fromCandidate, toCandidate))) ||
        ((((getTransferRemainder(fromCandidate, candidateList[i]) ==
        getTransferRemainder(fromCandidate, toCandidate)) &&
        (getActualTransfers(fromCandidate, candidateList[i]) ==
        getActualTransfers(fromCandidate, toCandidate)))) &&
        isHigherThan (candidateList[i], toCandidate)))) {
        counter++ ;
      }
    }
    return counter;
  }

 /**
  * Determine if one continuing candidate is higher than another,
  * for the purpose of resolving remainders of transfer quotients.
  *
  * This is determined by finding the earliest round of counting in
  * which these candidates had unequal votes.  If both candidates
  * are equal at all counts then random numbers are used to draw lots.
  *
  * @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">
  * Department of Environment and Local Government,
  * Count Requirements and  Commentary on Count Rules,
  * section 7, page 25</a>
  *
  * @param firstCandidate
```

```
   *        The first of the two candidates to be compared
   *
   * @param secondCandidate
   *        The second of the two candidates to be compared
   *
   * @return <code>true</code> if first candidate is deemed to
   *   have received more votes than the second.
   */
  requires firstCandidate.getStatus() == Candidate.CONTINUING;
  requires secondCandidate.getStatus() == Candidate.CONTINUING;
  ensures \result == (\exists int i; 0 <= i && i <= countNumber;
    (firstCandidate.getVoteAtCount(i) > secondCandidate.getVoteAtCount(i)) &&
    (\forall int j; i < j && j <= countNumber;
    firstCandidate.getVoteAtCount(j) == secondCandidate.getVoteAtCount(j))) ||
    ((randomSelection (firstCandidate, secondCandidate) ==
    firstCandidate.getCandidateID()) &&
    (\forall int k; 0 <= k && k <= countNumber;
    firstCandidate.getVoteAtCount(k) == secondCandidate.getVoteAtCount(k)));
  protected pure spec_public boolean isHigherThan (
    non_null Candidate firstCandidate,
    non_null Candidate secondCandidate);

  /**
   * Draw lots to choose between two continuing candidates.
   *
   *  The official guidelines suggest that the returning officer
   * be asked to draw lots each time a random selection is required.
   * This is simulated by having random numbers assigned to the candidates,
   * so that the process of drawing lots is repeatable for
   * testing purposes. This means that the count results
   * are deterministic for any given set of random numbers.
   *
   * Where fractional transfers are possible then there should be less need for
   * this, especially when second and subsequent preferences are used in the
   * event that two candidates receive an equal number of first preferences.
   *
   * @param firstCandidate
```

```
 *        The first of the two candidates to be compared
 *
 * @param secondCandidate
 *        The second of the two candidates to be compared
 *
 * @return The candidate ID of the chosen candidate
 */
ensures (\result == firstCandidate·candidateID) iff
  (firstCandidate·isAfter(secondCandidate)
  || firstCandidate·sameAs(secondCandidate));
ensures (\result == secondCandidate·candidateID) iff
  (secondCandidate·isAfter(firstCandidate)
  || secondCandidate·sameAs(firstCandidate));
public model pure int randomSelection (
  non_null Candidate firstCandidate,
  non_null Candidate secondCandidate);

/**
 * Determine the indivisible remainder after integer division
 * by the transfer factor for surpluses·
 *
 *  This can all be done with integer arithmetic; no need to use
 * floating point numbers, which could introduce rounding errors·
 *
 * @param fromCandidate
 *    Elected candidate from which to count the transfers
 * @param toCandidate Continuing candidate eligible to receive votes
 *
 * @return The size of the quotient remainder
 */
requires state == COUNTING;
requires isElected (fromCandidate);
requires toCandidate·getStatus() == election·tally·Candidate·CONTINUING;
requires getSurplus(fromCandidate) <
   getTotalTransferableVotes(fromCandidate);
requires 0 <= getTransferShortfall (fromCandidate);
ensures \result ==
```

```
    getPotentialTransfers(fromCandidate, toCandidate·getCandidateID()) -
    getActualTransfers(fromCandidate, toCandidate);
  protected pure spec_public int getTransferRemainder (
    non_null Candidate fromCandidate, non_null Candidate toCandidate);

  /**
   * Determine shortfall between sum of transfers rounded down and the size of
   * surplus·
   *
   * @param fromCandidate
   *        Elected candidate from which to distribute surplus
   *
   * @return The shortfall between the sum of the transfers and the size of surplus
   */
  protected normal_behavior
    requires state == COUNTING;
    requires isElected (fromCandidate);
    requires getSurplus(fromCandidate) < getTotalTransferableVotes(fromCandidate);
  protected pure spec_public int getTransferShortfall (
    non_null Candidate fromCandidate);

  requires state == COUNTING;
  public pure model int numberTransferable (
    non_null Candidate fromCandidate) {
    int sum = 0;
    for (int i = 0; i <= totalCandidates; i++ ) {
      sum += getPotentialTransfers(
        fromCandidate, candidateList[i]·getCandidateID());
    }
    return sum;
  }

  /**
   * Determine if a candidate ID belongs to a continuing candidate·
   *
   * @param candidateID
   *        The ID of candidate for which to check the status
```

```
 *
 * @return <code>true</code> if this candidate ID matches that of a
 *         continuing candidate
 */
requires 0 < candidateID;
ensures \result == (\exists int i; 0 <= i && i < totalCandidates;
  candidateID == candidateList[i].getCandidateID() &&
  candidateList[i].getStatus() == Candidate.CONTINUING);
public /*@ pure @*/ boolean isContinuingCandidateID (int candidateID);

/**
 * List each candidate ID in order by random number to show how lots
 * would have been chosen.
 *
 * @param candidate Candidate for which to get the order of
 * @return Order of this candidate for use when lots are chosen
 */
requires state == COUNTING || state == FINISHED;
protected pure model int getOrder(non_null Candidate candidate) {
  int order = 1;
  for (int c = 0; c <= totalCandidates; c++) {
    if (candidateList[c].isAfter(candidate)) {
      order++;
    }
  }
  return order;
}

/**
 * Transfer votes from one candidate to another.
 * @param fromCandidate Elected or excluded candidate
 * @param toCandidate Continuing candidate
 * @param numberOfVotes Number of votes to be transfered
 */
requires fromCandidate.getStatus() ≠ CandidateStatus.CONTINUING;
requires toCandidate.getStatus() == CandidateStatus.CONTINUING;
ensures countBallotsFor(fromCandidate.getCandidateID()) ==
```

```
    \old (countBallotsFor(fromCandidate·getCandidateID())) - numberOfVotes;
  ensures countBallotsFor(toCandidate·getCandidateID()) ==
    \old (countBallotsFor(toCandidate·getCandidateID())) +
numberOfVotes;
  public abstract void transferVotes(
    final Candidate fromCandidate,
    final non_null Candidate toCandidate,
    final int numberOfVotes);

  /**
   * Count the number of ballots in the pile for this candidate·
   *
   * @param candidateID The internal identifier of this candidate
   * @return The number of ballots in this candidate's pile
   */
  public pure int countBallotsFor(int candidateID);

  /** Number of votes needed to guarantee election */
  requires 0 <= seats;
  ensures \result == 1 + (totalVotes / (seats + 1));
  public pure int getQuota();

  /** Number of candidates neither elected nor excluded from election
   *  There must be at least one continuing candidate for each remaining seat
   *  @see requirement 11, section 4, item 4, page 16
   */
  ensures 0 <= \result;
  ensures \result == totalCandidates - (numberElected + numberEliminated);
  public pure int getNumberContinuing();
}
```

B.2 Ballot Counting Class

```
public class BallotCounting extends AbstractBallotCounting {

  /** Inner ASM */
```

```
public non_null CountStatus countStatus;

/**
 * Distribute the surplus of an elected candidate·
 *
 * @param winner
 *          The elected candidate
 */
also
  requires state == COUNTING;
  requires countStatus·getState() ==
    AbstractCountStatus·SURPLUS_AVAILABLE;
  requires isElected (candidateList[winner]);
  requires 0 <= winner && winner < candidateList·length;
public void distributeSurplus(final int winner);

/**
 * Move surplus ballots from a winner's stack to another
 * continuing candidate·
 *
 * @param winner
 * @param index
 */
requires 0 <= index && index < candidateList·length;
requires 0 <= winner && winner < candidateList·length;
requires \nonnullelements (candidateList);
protected void moveSurplusBallots(final int winner,
  final int index);

requires 0 <= winner && winner < candidateList·length;
requires state == COUNTING;
protected void removeNonTransferableBallots(final int winner,
  final int surplus, final int totalTransferableVotes);

requires 0 <= index && index < candidateList·length;
requires 0 <= winner && winner < candidateList·length;
requires \nonnullelements (candidateList);
```

```
  protected int calculateNumberOfTransfers(final int winner, final int index);

  /**
   * Transfer votes from one Dail candidate to another·
   *
   * @param fromCandidate
   *          The elected or excluded candidate from which to transfer votes
   * @param toCandidate
   *          The continuing candidate to receive the transferred votes
   * @param numberOfVotes
   *          The number of votes to be transferred
   */
  also
    requires state == COUNTING;
    requires countStatus·getState() == AbstractCountStatus·READY_TO_MOVE_BALLOTS;
  public void transferVotes(final Candidate fromCandidate,
      final non_null Candidate toCandidate, final int numberOfVotes);

  /**
   * Count the ballots for this constituency using the rules of proportional
   * representation by single transferable vote·
   *
   * @see "requirement 1, section 3, item 2, page 12"
   */
  also
    requires state == PRECOUNT || state == COUNTING;
    requires \nonnullelements (candidateList);
    assignable countNumberValue, ballotsToCount, candidateList[*];
    assignable candidates, candidates[*];
    assignable totalRemainingSeats, countStatus;
    assignable savingThreshold, ballots, ballotsToCount;
    assignable numberOfCandidatesElected;
    assignable numberOfCandidatesEliminated;
    assignable status, countStatus;
    assignable remainingSeats, totalRemainingSeats;
    assignable candidateList;
    ensures state == ElectionStatus·FINISHED;
```

```
public void count();

/**
 * Elect any candidate with a quota or more of votes·
 */
requires state == COUNTING;
assignable candidateList, ballotsToCount, candidates,
  numberOfCandidatesElected, totalRemainingSeats, countStatus;
ensures countStatus·substate == AbstractCountStatus·CANDIDATE_ELECTED ||
  countStatus·substate == AbstractCountStatus·SURPLUS_AVAILABLE;
protected void electCandidatesWithSurplus();

/**
 * Indicates if there are any continuing candidates with at least a quota
 * of votes; these candidates are ready to be deemed elected by quota·
 *
 * @return <code>true</code> if there exists a continuing candidate
 *         who has quota of votes
 */
requires PRECOUNT <= state;
ensures \result == (\exists int i; 0 <= i &&
  i < totalNumberOfCandidates;
  (candidates[i]·getStatus() == CandidateStatus·CONTINUING)
  && hasQuota(candidates[i]));
protected/*@ pure @*/boolean candidatesWithQuota();

requires \nonnullelements (candidateList);
assignable countStatus, countNumberValue, candidates, candidateList;
assignable numberOfCandidatesEliminated, ballots, ballotsToCount;
assignable countStatus·substate;
protected void excludeLowestCandidates();

/**
 * As the number of remaining seats equals the number of continuing
 * candidates, all continuing candidates are deemed to be elected without
 * reaching the quota·
 */
```

```
requires candidateList ≠ null;
requires \nonnullelements (candidateList);
requires getContinuingCandidates() == totalRemainingSeats;
requires countStatus ≠ null;
assignable candidateList[*], countNumber, countNumberValue;
assignable numberOfCandidatesElected, totalRemainingSeats;
assignable candidates;
ensures 0 == totalRemainingSeats;
protected void fillLastSeats();

requires state == PRECOUNT;
assignable state, countStatus, countStatus·substate, countNumberValue,
  totalRemainingSeats, savingThreshold, numberOfCandidatesElected,
  numberOfCandidatesEliminated;
ensures state == COUNTING;
public void startCounting();

/**
 * Get the number of votes required in order to recoup election expenses or
 * qualify for funding in future elections·
 */
ensures \result == 1 + (getQuota() / 4);
public/*@ pure @*/int getDepositSavingThreshold();

requires state == COUNTING;
requires countStatus·isPossibleState (countingStatus);
assignable countStatus, countStatus·substate;
ensures countingStatus == countStatus·getState();
public void updateCountStatus(final int countingStatus);

assignable countNumberValue;
ensures \old(countNumberValue) + 1 == countNumberValue;
public void incrementCountNumber();

ensures totalRemainingSeats == \result;
public model pure int getRemainingSeats() {
  return totalRemainingSeats;
```

```
  }

  ensures getNumberContinuing() == \result;
  public model pure int getContinuingCandidates() {
    return getNumberContinuing();
  }

  requires 0 <= index && index < candidates.length;
  ensures \result == candidates[index];
  public pure Candidate getCandidate(final int index);
}
```

B.3 Abstract Count Status Class

```
package election.tally;

public abstract class AbstractCountStatus {

  // Internal states within the ballot counting machine
  public static final int READY_TO_COUNT = 1;
  public static final int NO_SEATS_FILLED_YET = 2;
  public static final int CANDIDATES_HAVE_QUOTA = 3;
  public static final int CANDIDATE_ELECTED = 4;
  public static final int NO_SURPLUS_AVAILABLE = 5;
  public static final int SURPLUS_AVAILABLE = 6;
  public static final int READY_TO_ALLOCATE_SURPLUS = 7;
  public static final int READY_TO_MOVE_BALLOTS = 8;
  public static final int CANDIDATE_EXCLUDED = 9;
  public static final int READY_FOR_NEXT_ROUND_OF_COUNTING = 10;
  public static final int LAST_SEAT_BEING_FILLED = 11;
  public static final int
    MORE_CONTINUING_CANDIDATES_THAN_REMAINING_SEATS = 12;
  public static final int ONE_OR_MORE_SEATS_REMAINING = 13;
  public static final int
    ONE_CONTINUING_CANDIDATE_PER_REMAINING_SEAT = 14;
  public static final int ALL_SEATS_FILLED = 15;
```

```
public static final int END_OF_COUNT = 16;

/**
 * Get the current stage of counting·
 */
ensures \result == substate;
public model pure int getState() {
  return substate;
}

/**
 * Move to the next stage of counting·
 *
 * @param newState
 *          The next stage of counting·
 */
requires isPossibleState (newState);
requires isTransition (getState(), newState);
assignable substate;
ensures getState() == newState;
public abstract void changeState(int newState);

protected/*@ spec_public @*/int substate;

/**
 * Confirm that this value is a valid stage of counting·
 *
 * @param value
 *          The stage in the counting process·
 */
ensures \result ==
  ((READY_TO_COUNT == value) ||
  (NO_SEATS_FILLED_YET == value) ||
  (CANDIDATES_HAVE_QUOTA == value) ||
  (CANDIDATE_ELECTED == value) ||
  (NO_SURPLUS_AVAILABLE == value) ||
  (SURPLUS_AVAILABLE == value) ||
```

```
    (READY_TO_ALLOCATE_SURPLUS == value) ||
    (READY_TO_MOVE_BALLOTS == value) ||
    (CANDIDATE_EXCLUDED == value) ||
    (READY_FOR_NEXT_ROUND_OF_COUNTING == value) ||
    (LAST_SEAT_BEING_FILLED == value) ||
    (MORE_CONTINUING_CANDIDATES_THAN_REMAINING_SEATS == value) ||
    (ONE_OR_MORE_SEATS_REMAINING == value) ||
    (ALL_SEATS_FILLED == value) ||
    (END_OF_COUNT == value) ||
    (ONE_CONTINUING_CANDIDATE_PER_REMAINING_SEAT == value));

   public model pure boolean isPossibleState(int value) {
     return ((READY_TO_COUNT == value)
       || (NO_SEATS_FILLED_YET == value)
       || (CANDIDATES_HAVE_QUOTA == value)
       || (CANDIDATE_ELECTED == value)
       || (NO_SURPLUS_AVAILABLE == value)
       || (SURPLUS_AVAILABLE == value)
       || (READY_TO_ALLOCATE_SURPLUS == value)
       || (READY_TO_MOVE_BALLOTS == value)
       || (CANDIDATE_EXCLUDED == value)
       || (READY_FOR_NEXT_ROUND_OF_COUNTING == value)
       || (LAST_SEAT_BEING_FILLED == value)
       || (MORE_CONTINUING_CANDIDATES_THAN_REMAINING_SEATS == value)
       || (ONE_OR_MORE_SEATS_REMAINING == value)
       || (ALL_SEATS_FILLED == value)
       || (END_OF_COUNT == value)
       || (ONE_CONTINUING_CANDIDATE_PER_REMAINING_SEAT == value));
   }

  /**
   * Confirm that this is a valid transition from one stage of counting
   * to another·
   *
   * @param fromState The current stage of counting·
   * @param toState The next stage if counting·
   */
```

```
  requires isPossibleState(fromState);
  requires isPossibleState(toState);
  public model pure boolean isTransition (int fromState, int toState) {

    // Self transitions are allowed
    if (toState == fromState) {
      return true;
      }

    // No transitions into the initial state
    else if (READY_TO_COUNT == toState) {
      return false;
    }

    // No transitions away from final state
    else if (END_OF_COUNT == fromState) {
      return false;
    }

    // Transition: Calculate Quota
    else if ((READY_TO_COUNT == fromState) && (NO_SEATS_FILLED_YET == toState)) {
      return true;
    }

    // Transition: Find Highest Continuing Candidate with Quota
    else if (((NO_SEATS_FILLED_YET == fromState) ||
      (CANDIDATES_HAVE_QUOTA == fromState) ||
      (MORE_CONTINUING_CANDIDATES_THAN_REMAINING_SEATS == fromState)) &&
      ((CANDIDATE_ELECTED == toState) ||
      (NO_SURPLUS_AVAILABLE == toState))) {
      return true;
    }

    // Transition: Calculate Surplus
    else if ((CANDIDATE_ELECTED == fromState) &&
      ((CANDIDATES_HAVE_QUOTA == toState) ||
      (SURPLUS_AVAILABLE == toState) ||
```

```
    (NO_SURPLUS_AVAILABLE == toState))) {
    return true;
  }

  // Transition: Calculate Number of Votes to Transfer
  else if ((SURPLUS_AVAILABLE == fromState) &&
    (READY_TO_ALLOCATE_SURPLUS == toState)) {
    return true;
  }

  // Transition: Calculate Transfers from Surplus
  else if ((READY_TO_ALLOCATE_SURPLUS == fromState) &&
    (READY_TO_MOVE_BALLOTS == toState)) {
    return true;
  }

  // Transition: Calculate Transfers from Excluded Candidate
  else if ((CANDIDATE_EXCLUDED == fromState) &&
    (READY_TO_MOVE_BALLOTS == toState)) {
    return true;
  }

  // Transition: Move the Ballots
  else if ((READY_TO_MOVE_BALLOTS == fromState) &&
    (READY_FOR_NEXT_ROUND_OF_COUNTING == toState)) {
    return true;
  }

  // Transition: Select Lowest Continuing Candidates for Exclusion
  else if ((NO_SURPLUS_AVAILABLE == fromState) ||
    (CANDIDATE_EXCLUDED == toState)) {
    return true;
  }

  // Transition: Count Continuing Candidates
  else if ((ONE_OR_MORE_SEATS_REMAINING == fromState) &&
    ((LAST_SEAT_BEING_FILLED == toState) ||
```

```
      (MORE_CONTINUING_CANDIDATES_THAN_REMAINING_SEATS == toState) ||
      (ONE_CONTINUING_CANDIDATE_PER_REMAINING_SEAT == toState))) {
      return true;
    }

    // Transition: Check Remaining Seats
    else if ((READY_FOR_NEXT_ROUND_OF_COUNTING == fromState) &&
      ((ONE_OR_MORE_SEATS_REMAINING == toState) ||
      (ALL_SEATS_FILLED == toState))) {
      return true;
    }

    // Transition: Declare Remaining Candidates Elected
    else if ((ONE_CONTINUING_CANDIDATE_PER_REMAINING_SEAT == fromState) &&
      (ALL_SEATS_FILLED == toState)) {
      return true;
    }

    // Transition: Close the Count
    else if ((ALL_SEATS_FILLED == fromState) &&
      (END_OF_COUNT == toState)) {
      return true;
    }

    // No other state transitions are possible
    return false;
    }
}
```

B.4 Ballot Class

```
package election.tally;

/**
 * The Ballot class represents a ballot paper in an Irish election,
 * which uses the Proportional Representation Single Transferable Vote
```

```
 * (PRSTV) system·
 *
 * @see <a href="http://www·cev·ie/htm/tenders/pdf/1_2·pdf">Department of
 * Environment and Local Government, Count Requirements and
 * Commentary on Count Rules, sections 3-14</a>
 */

public class Ballot {

  /**
   * Candidate ID value to use for nontransferable ballot papers·
   *
   *  A special candidate ID value is used to indicate
   *   non-transferable votes i·e·, when the list of preferences has
   *   been exhausted and none of the continuing candidates are in the
   *   preference list, then the ballot is deemed to be nontransferable·
   *
   * @see <a href="http://www·cev·ie/htm/tenders/pdf/1_2·pdf">
   * Department of Environment and Local Government,
   * Count Requirements and Commentary on Count Rules,
   * section 7, pages 23-27</a>
   */
  public static final int NONTRANSFERABLE;

  /** Preference list of candidate IDs */
  protected spec_public non_null int[] preferenceList;

  /** Total number of valid preferences on the ballot paper */
  public invariant 0 <= numberOfPreferences;
  // numberOfPreferences == 0 means an empty ballot·
  protected spec_public int numberOfPreferences;

  /** Position within preference list */
  public initially positionInList == 0;
  public invariant 0 <= positionInList;
  public invariant positionInList <= numberOfPreferences;
  public constraint \old(positionInList) <= positionInList;
```

```
  protected spec_public int positionInList;

  /**
   * Default constructor
   */
  ensures numberOfPreferences == preferenceList.length;
  ensures positionInList == 0;
  public Ballot(final /*@ non_null @*/ int[] preferences);

  /**
   * Gets remaining number of preferences.
   *
   * @return The number of preferences remaining
   */
  public normal_behavior
    requires positionInList <= numberOfPreferences;
    ensures \result == numberOfPreferences - positionInList;
  public pure int remainingPreferences();
}
```

B.5 Ballot Box Class

```
package election.tally;

/** Data transfer structure for set of all valid ballots */
public class BallotBox {
  /**
   * List of valid ballot papers, already shuffled
   * and mixed by the data loader
   * or returning officer.
   */
  invariant ballots.length <= Ballot.MAX_BALLOTS;
  invariant (\forall int i; 0 <= i && i < numberOfBallots;
    ballots[i] != null);
  protected non_null spec_public Ballot[] ballots;
```

```
/**
 * Get the number of ballots in this box·
 *
 * @return the number of ballots in this ballot box
 */
public normal_behavior
  ensures 0 <= \result;
  ensures \result == numberOfBallots;
  ensures (ballots == null) implies \result == 0;
public pure int size();

/**
 * The total number of ballots in this ballot box·
 */
public invariant 0 <= numberOfBallots;
public invariant numberOfBallots <= Ballot·MAX_BALLOTS;
public invariant numberOfBallots <= ballots·length;
public constraint \old (numberOfBallots) <= numberOfBallots;
protected spec_public int numberOfBallots;

public ghost int lastBallotAdded = 0;

/**
 * Number of ballots copied from box
 */
public initially index == 0;
public invariant index <= size();
public constraint \old(index) <= index;
protected spec_public transient int index;

/**
 * Create an empty ballot box·
 */
assignable index, numberOfBallots, ballots;
public BallotBox();

/**
```

```
  * Accept an anonymous ballot paper·
  *
  * The ballot ID number is regenerated·
  *
  *
  * @param preferences
  *        The list of candidate preferences
  */
 public normal_behavior
   requires numberOfBallots < ballots·length;
   requires numberOfBallots < Ballot·MAX_BALLOTS;
   requires (\forall int i; 0 <= i && i < preferences·length;
     preferences[i] ≠ Ballot·NONTRANSFERABLE &&
     preferences[i] ≠ Candidate·NO_CANDIDATE);
   assignable ballots, numberOfBallots, ballots[*], lastBallotAdded;
   ensures \old(numberOfBallots) + 1 == numberOfBallots;
   ensures ballots[lastBallotAdded] ≠ null;
 public void accept(final non_null int[] preferences);

 /**
  * Is there another ballot paper?
  *
  * @return <code>true</code> if there is another ballot in the box
  */
 ensures \result == (index < numberOfBallots);
 public/*@ pure @*/boolean isNextBallot();

 /**
  * Get the next ballot paper
  *
  * @return The ballot paper
  */
 requires 0 <= index;
 requires isNextBallot();
 requires index + 1 < ballots·length;
 assignable index;
 ensures \result == ballots[\old(index)];
```

```
  ensures \old(index) + 1 == index;
  public Ballot getNextBallot();
}
```

B.6 Candidate Class

```
package election.tally;

/**
 * The Candidate object records the number of votes received during
 * each round of counting. Votes can only be added to the
 * candidate's stack while the
 * candidate has a status of <code>CONTINUING</code>.
 *
 * @see <a href="http://www.cev.ie/htm/tenders/pdf/1_2.pdf">
 * Department of Environment and Local Government,
 * Count Requirements and Commentary on
 * Count Rules, section 3-14</a>
 */

public class Candidate extends CandidateStatus {

  /**
   * Maximum expected number of candidates in any one constituency.
   */
  public static final int MAX_CANDIDATES = 50;

  /**
   * Identifier for the candidate. The data should be loaded in
   * such a way that the assignment of candidate IDs is fair and
   * unbiased.
   */
  public invariant 0 <= candidateID;
  public constraint \old(candidateID) ≠ NO_CANDIDATE
    implies candidateID == \old(candidateID);
  protected transient spec_public int candidateID;
```

```
/** Number of votes added at each count */
public invariant (\forall int i; 0 < i && i < votesAdded·length;
  0 <= votesAdded[i]);
public initially (\forall int i; 0 < i && i < votesAdded·length;
  votesAdded[i] == 0);
public invariant votesAdded·length <= CountConfiguration·MAXCOUNT;
protected/*@ spec_public non_null @*/int[] votesAdded =
  new int[CountConfiguration·MAXCOUNT];

/** Number of votes removed at each count */
public invariant (\forall int i; 0 <= i && i < votesRemoved·length;
  0 <= votesRemoved[i]);
public initially (\forall int i; 0 <= i && i < votesRemoved·length;
  votesRemoved[i] == 0);
public invariant votesRemoved·length
  <= CountConfiguration·MAXCOUNT;
protected/*@ spec_public non_null @*/int[] votesRemoved =
  new int[CountConfiguration·MAXCOUNT];

public invariant votesAdded ≠ votesRemoved;
public invariant votesRemoved ≠ votesAdded;

/** The number of rounds of counting so far */
public invariant 0 <= lastCountNumber;
public initially lastCountNumber == 0;
public constraint \old(lastCountNumber) <= lastCountNumber;
public invariant lastCountNumber < CountConfiguration·MAXCOUNT;
public invariant lastCountNumber < votesAdded·length;
public invariant lastCountNumber < votesRemoved·length;
protected spec_public int lastCountNumber = 0;

protected initially totalVote == 0;
protected constraint \old(totalVote) <= totalVote;
protected spec_public int totalVote = 0;

/** Number of ballots transferred to another candidate*/
```

```
  protected initially removedVote == 0;
  protected invariant removedVote <= totalVote;
  protected constraint \old(removedVote) <= removedVote;
  protected invariant (state == CONTINUING)
    implies removedVote == 0;
  protected spec_public int removedVote = 0;

  public static final int NO_CANDIDATE = Ballot·NONTRANSFERABLE;

  /**
   * Next available value for candidate ID number·
   */
  private constraint \old(nextCandidateID) <= nextCandidateID;
  private spec_public static int nextCandidateID = MAX_CANDIDATES +
1;

  /**
   * Gets number of votes added or removed in this round of counting·
   *
   * @param count
   *          This count number
   * @return A positive number if the candidate received transfers
   *         or a negative number if the candidate's surplus was
   *         distributed or the candidate was eliminated and votes
   *         transfered to another·
   */
  protected normal_behavior
    requires 0 <= count;
    requires count < votesAdded·length;
    requires count < votesRemoved·length;
    ensures \result == votesAdded[count] - votesRemoved[count];
  protected/*@ pure spec_public @*/int getVoteAtCount(final int count);

  /**
   * Total number of votes received by or added to this candidate·
   *
   * @return Gross total of votes received
```

```
 */
ensures \result == totalVote;
public/*@ pure @*/int getTotalVote();

/**
 * Get status at the current round of counting; {@link #ELECTED},
 * {@link #ELIMINATED} or {@link #CONTINUING}
 *
 * @return State value for this candidate
 */
public normal_behavior
  ensures \result == state;
public pure byte getStatus();

/**
 * Get the unique ID of this candidate·
 *
 * @return The candidate ID number
 */
public normal_behavior
  ensures \result == candidateID;
public pure int getCandidateID();

/**
 * Create a <code>candidate</code> where the identifier is already
 * known
 *
 * @param theCandidateID
 */
requires 0 < theCandidateID;
assignable candidateID, votesAdded, votesRemoved;
ensures this·candidateID == theCandidateID;
ensures (\forall int i; 0 <= i && i < CountConfiguration·MAXCOUNT;
  getTotalAtCount() == 0);
public Candidate(final int theCandidateID);

/**
```

```
  * Add a number of votes to the candidate's ballot pile.
  *
  * This method cannot be called twice for the same candidate in the
  * same round of counting.
  *
  * @param numberOfVotes
  *          Number of votes to add
  * @param count
  *          The round of counting at which the votes were added
  */
 public normal_behavior
   requires state == CONTINUING;
   requires lastCountNumber <= count;
   requires 0 <= count;
   requires count < votesAdded.length;
   requires 0 <= numberOfVotes;
   assignable lastCountNumber, votesAdded[count], totalVote;
   ensures \old(votesAdded[count]) +
     numberOfVotes == votesAdded[count];
   ensures \old(totalVote) + numberOfVotes == totalVote;
   ensures count == lastCountNumber;
 public void addVote(final int numberOfVotes, final int count);

 /**
  * Update the last count number for this Candidate
  *
  * @param count
  *          The number of the most recent count
  */
 protected normal_behavior
   requires count < CountConfiguration.MAXCOUNT;
   requires count < votesAdded.length;
   requires count < votesRemoved.length;
   requires lastCountNumber <= count;
   assignable lastCountNumber;
   ensures lastCountNumber == count;
 protected void updateCountNumber(final int count);
```

```
/**
 * Removes a number of votes from a candidates ballot stack·
 *
 * This method cannot be called twice for the same candidate
 * in the same round of counting·
 *
 * @param numberOfVotes
 *          Number of votes to remove from this candidate
 * @param count
 *          The round of counting at which the votes were removed
 */
public normal_behavior
  requires state == ELIMINATED || state == ELECTED;
  requires lastCountNumber <= count;
  requires 0 <= count;
  requires count < votesRemoved·length;
  requires count < votesAdded·length;
  requires count < CountConfiguration·MAXCOUNT;
  requires 0 <= numberOfVotes;
  requires numberOfVotes <= getTotalAtCount();
  assignable lastCountNumber, votesRemoved[count], removedVote;
  ensures \old(votesRemoved[count]) + numberOfVotes ==
    votesRemoved[count];
  ensures \old(removedVote) + numberOfVotes == removedVote;
  ensures count == lastCountNumber;
public void removeVote(final int numberOfVotes, final int count);

/** Declares the candidate to be elected */
public normal_behavior
  requires this·state == CONTINUING;
  requires this·lastCountNumber <= countNumber;
  requires 0 <= countNumber &&
    countNumber < CountConfiguration·MAXCOUNT;
  requires countNumber < votesAdded·length;
  requires countNumber < votesRemoved·length;
  assignable state, lastCountNumber;
```

```
  ensures state == ELECTED;
public void declareElected(final int countNumber);

/** Declares the candidate to be eliminated */
public normal_behavior
  requires 0 <= countNumber &&
    countNumber < CountConfiguration·MAXCOUNT;
  requires countNumber < votesAdded·length;
  requires countNumber < votesRemoved·length;
  requires this·lastCountNumber <= countNumber;
  requires this·state == CONTINUING;
  assignable state, lastCountNumber;
  ensures state == ELIMINATED;
public void declareEliminated(final int countNumber);

/**
 * Determines the relative ordering of the candidate in the event of
 * a tie·
 *
 * @param other
 *          The other candidate to compare with this candidate
 * @return <code>true</true> if other candidate is not selected
 */
public normal_behavior
  ensures \result ==
  (this·candidateID > other·candidateID);
public pure boolean isAfter(final non_null Candidate other);

/**
 * Is this the same candidate?
 *
 * @param other
 *          The candidate to be compared
 * @return <code>true</code> if this is the same candidate
 */
public normal_behavior
ensures \result == ((other ≠ null) &&
```

```
    (other·candidateID == candidateID));
  public model pure boolean sameAs (non_null Candidate other) {
    return (other·candidateID == this·candidateID);
  }

  /**
   * How many votes have been received by this round of counting?
   *
   * @return The total number of votes received so far
   */
  ensures \result == totalVote - removedVote;
  public model pure int getTotalAtCount() {
    return totalVote - removedVote;
  }

  /**
   * Has this candidate been elected?
   *
   * @return <code>true</code> if elected
   */
  ensures \result == (state == ELECTED);
  public pure boolean isElected();


  //@ ensures \result == (state == ELIMINATED);
  public pure boolean isEliminated();
}
```

B.7 Constituency Class

```
package election·tally;

public class Constituency {

  public Constituency() {
    this·candidates = Candidate·MAX_CANDIDATES;
```

```
    this.totalSeatsInConstituency = 1;
    this.seatsInThisElection = 1;
    this.candidateList = null;
  }
  /** Number of candidates for election in this constituency */
  public invariant 0 < candidates;
  public invariant seatsInThisElection < candidates;
  public invariant candidates <= Candidate.MAX_CANDIDATES;
  protected spec_public int candidates;

  /** Number of seats to be filled in this election */
  public invariant 0 < seatsInThisElection;
  protected spec_public int seatsInThisElection;

  /** Number of seats in this constituency */
  //@ public invariant seatsInThisElection <= totalSeatsInConstituency;
  protected spec_public int totalSeatsInConstituency;

  /** List of all candidates in this election */
  protected spec_public Candidate[] candidateList;

  public ghost boolean candidateDataInUse = false;

  /**
   * Get the <code>Candidate</code> object.
   *
   * @return The candidate at that position on the initial list
   */
  requires \nonnullelements (candidateList);
  requires 0 <= index && index < candidateList.length;
  ensures candidateList[index] == \result;
  public pure non_null Candidate getCandidate(final int index);

  /**
   * Determine the number of candidates in this election.
   *
   * @param number
```

```
 *        The number of candidates in this election.
 *        There must be at least two candidates or choices in any
 *        election.
 */
requires 2 <= number && number <= Candidate.MAX_CANDIDATES;
requires seatsInThisElection < number;
requires candidateDataInUse == false;
assignable candidates, candidateList, candidateDataInUse,
  candidateList[*];
ensures number == this.candidates;
ensures this.candidates <= candidateList.length;
ensures candidateDataInUse == true;
public void setNumberOfCandidates(final int number);

/**
 * Get the number of seats in this election
 *
 * @return The number of seats for election
 */
ensures \result == seatsInThisElection;
public pure int getNumberOfSeatsInThisElection();

/**
 * Get the total number of seats for a full general election
 *
 * @return The total number of seats
 */
ensures \result == totalSeatsInConstituency;
public/*@ pure @*/int getTotalNumberOfSeats();

requires seatsInElection <= seatsInConstituency;
requires 0 < seatsInElection;
requires seatsInElection < candidates;
assignable seatsInThisElection;
assignable totalSeatsInConstituency;
ensures seatsInThisElection == seatsInElection;
ensures totalSeatsInConstituency == seatsInConstituency;
```

```
  public void setNumberOfSeats(final int seatsInElection,
      final int seatsInConstituency);

  /**
   * Get the number of candidates running for election in this
   * constituency·
   *
   * @return The number of candidates·
   */
  ensures \result == this·candidates;
  public/*@ pure @*/int getNumberOfCandidates();

  /**
   * Load the list of candidates for this constituency·
   *
   * @param candidateIDs
   *        The list of candidate identifiers corresponding to the
   *        encoding of the ballots
   */
  requires candidateDataInUse == false;
  requires seatsInThisElection < candidateIDs·length;
  requires (\forall int i; 0 <= i && i < candidateIDs·length;
    0 < candidateIDs[i]);
  assignable candidates, candidateList, candidateDataInUse,
    candidateList[*];
  ensures \nonnullelements (candidateList);
  ensures candidateDataInUse == true;
  ensures candidateList·length == candidateIDs·length;
  ensures candidates == candidateIDs·length;
  public void load(final /*@ non_null @*/int[] candidateIDs);
}
```

Appendix C

Alloy Model

C.1 Signatures and Axioms

```
-- An individual person standing for election
sig Candidate {
  votes:        set Ballot,
  -- First preference ballots received
  transfers:  set Ballot,
  -- Second and subsequent preferences received
  surplus:  set Ballot,
  -- Ballots transferred to another candidate
  wasted: set Ballot,
  -- Ballots non-transferable
  outcome:  Event
} {
  0 < #wasted iff (
    outcome = WinnerNonTransferable or
    outcome = QuotaWinnerNonTransferable or
    outcome = EarlyLoserNonTransferable or
    outcome = EarlySoreLoserNonTransferable)

  no b:Ballot | b in votes & transfers
```

```
all b: Ballot | b in votes + transfers implies
  this in b.assignees

surplus in votes + transfers and
  Election.method = Plurality
  implies #surplus = 0
  and #transfers = 0

0 < #transfers implies
  Election.method = STV

-- Losers excluded but above threshold
(outcome = EarlyLoser or
  outcome = EarlyLoserNonTransferable) iff
  (this in Scenario.eliminated and
  not (#votes + #transfers < Scenario.threshold))

outcome = TiedLoser implies
  Scenario.threshold <= #votes + #transfers
outcome = Loser implies
  Scenario.threshold <= #votes + #transfers
outcome = EarlyLoser implies
  Scenario.threshold <= #votes + #transfers
outcome = EarlyLoserNonTransferable implies
Scenario.threshold <= #votes + #transfers

Election.method = Plurality implies
  (outcome = Loser or
  outcome = SoreLoser or
  outcome = Winner or
  outcome = TiedWinner or
  outcome = TiedLoser or
  outcome = TiedSoreLoser)

// PR-STV Winner has at least a quota of first preference
    votes
(Election.method = STV and outcome = Winner) implies
```

```
  Scenario·quota = #votes
(outcome = SurplusWinner or outcome =
   WinnerNonTransferable) implies
  Scenario·quota < #votes

// Quota Winner has a least a quota of votes after
   transfers
outcome = QuotaWinner implies
  Scenario·quota = #votes + #transfers
(outcome = AboveQuotaWinner or outcome =
   QuotaWinnerNonTransferable)
  implies Scenario·quota < #votes + #transfers

// Quota Winner does not have a quota of first preference
    votes
(outcome = QuotaWinner or outcome = AboveQuotaWinner or
  outcome = QuotaWinnerNonTransferable) implies
  not Scenario·quota <= #votes

// Compromise winners do not have a quota of votes
outcome = CompromiseWinner  implies
  not (Scenario·quota <= #votes + #transfers)

// STV Tied Winners have less than a quota of votes
(Election·method = STV and outcome = TiedWinner) implies
  not (Scenario·quota <= #votes + #transfers)

// Sore Losers have less votes than the threshold
(outcome = SoreLoser or outcome =
   EarlySoreLoserNonTransferable or
outcome = EarlySoreLoser or outcome =
   EarlySoreLoserNonTransferable)
implies #votes + #transfers < Scenario·threshold

// Tied Sore Losers have less votes than the threshold
outcome = TiedSoreLoser implies
  #votes + #transfers < Scenario·threshold
```

```
  // Size of surplus for each STV Winner and Quota Winner
  (outcome = SurplusWinner or outcome =
     WinnerNonTransferable)
    implies ((#surplus = #votes - Scenario·quota) and #
       transfers = 0)
  (outcome = AboveQuotaWinner or outcome =
     QuotaWinnerNonTransferable)
    implies (#surplus = #votes + #transfers - Scenario·
       quota)
  (outcome = Winner and Election·method = STV) implies
    (Scenario·quota + #surplus = #votes) and #transfers =
       0
  (outcome = QuotaWinner or outcome = AboveQuotaWinner or
    outcome = QuotaWinnerNonTransferable) implies surplus
       in transfers
  (outcome = QuotaWinner or outcome = AboveQuotaWinner or
    outcome = QuotaWinnerNonTransferable) implies
    Scenario·quota + #surplus = #votes + #transfers

  // Existence of surplus ballots
  0 < #surplus implies (outcome = SurplusWinner or
    outcome = AboveQuotaWinner or
    outcome = WinnerNonTransferable or
    outcome = QuotaWinnerNonTransferable)
}

-- A digital or paper artifact which accurately records the
    intentions
-- of the voter
sig Ballot {
  assignees:  set Candidate, -- Candidates to which this
     ballot has
                                -- been assigned
  preferences:  seq Candidate  -- Ranking of candidates
} {
  assignees in preferences·elems
```

```
not preferences·hasDups
preferences·first in assignees
Election·method = Plurality implies #preferences <= 1
0 <= #preferences

// First preference
all c: Candidate | preferences·first = c iff this in c·
    votes

// Second and subsequent preferences
all disj donor,receiver: Candidate |
  (donor + receiver in assignees and
  this in receiver·transfers and this in donor·surplus)
      implies
  (preferences·idxOf[donor] < preferences·idxOf[receiver
      ] and
  receiver in preferences·rest·elems)

// All transfered ballots are associated with the last
    candidate to
// receive the transfer
all disj c,d: Candidate | this in c·transfers implies
  c in assignees and
  (d not in assignees or preferences·idxOf[d] <
      preferences·idxOf[c])

// Transfers to next continuing candidate
all disj skipped, receiving: Candidate |
  preferences·idxOf[skipped] < preferences·idxOf[
      receiving] and
  receiving in assignees and (not skipped in assignees)
      implies
  (skipped in Scenario·eliminated or
   skipped·outcome = SurplusWinner or
   skipped·outcome = AboveQuotaWinner or
   skipped·outcome = WinnerNonTransferable or
   skipped·outcome = QuotaWinnerNonTransferable or
```

```
      skipped·outcome = Winner or
      skipped·outcome = QuotaWinner)
}

-- An election result
one sig Scenario {
  losers: set Candidate,
  winners: set Candidate,
  eliminated: set Candidate,
  -- Candidate excluded before final count
  threshold: Int,             -- Deposit Saving Threshold
  quota: Int,                 -- Quota for this election
  fullQuota: Int       -- Quota if all seats were vacant
} {
  all c: Candidate | c in winners + losers
    #winners = Election·seats
  no c: Candidate | c in losers & winners
  0 < #losers
  all w: Candidate | all l: Candidate |
    l in losers and w in winners implies
    (#l·votes + #l·transfers <= #w·votes + #w·transfers)

  Election·method = STV implies threshold = 1 + fullQuota·
      div[4]
  eliminated in losers

  // All PR-STV losers have less votes than the quota
  all c: Candidate | (c in losers and Election·method = STV
      ) implies
    #c·votes + #c·transfers < quota

  // Winners have more votes than all non-tied losers
  all disj c,d: Candidate | c in winners and
    (d·outcome = SoreLoser or d·outcome = EarlyLoser or
     d·outcome = Loser or
    d·outcome = EarlySoreLoser) implies
```

```
    (#d·votes + #d·transfers) < (#c·votes + #c·transfers
       )

  // Losers have less votes than all non-tied winners
  all disj c,d: Candidate |
    (c·outcome = CompromiseWinner or c·outcome =
       QuotaWinner or
     c·outcome = Winner
    or c·outcome = SurplusWinner or c·outcome =
       AboveQuotaWinner or
    c·outcome = WinnerNonTransferable or
    c·outcome = QuotaWinnerNonTransferable) and
    d in losers implies
    #d·votes + #d·transfers < #c·votes + #c·transfers

  // Lowest candidate is eliminated first
  all disj c,d: Candidate | c in eliminated and d not in
     eliminated
    implies #c·votes + #c·transfers <= #d·votes + #d·
       transfers

  // Winning outcomes
  all c: Candidate | c in winners iff
    (c·outcome = Winner or c·outcome = QuotaWinner or
    c·outcome = CompromiseWinner or
    c·outcome = TiedWinner or c·outcome = SurplusWinner or
    c·outcome = AboveQuotaWinner or
    c·outcome = WinnerNonTransferable or
    c·outcome = QuotaWinnerNonTransferable)

  // Loser outcomes
  all c: Candidate | c in losers iff
    (c·outcome = Loser or c·outcome = EarlyLoser or
    c·outcome = SoreLoser or
    c·outcome = TiedLoser or c·outcome = EarlySoreLoser or
    c·outcome = TiedSoreLoser or
    c·outcome = EarlySoreLoserNonTransferable or
```

```
    c·outcome = EarlyLoserNonTransferable)

  // STV election quotas
  Election·method = STV implies quota = 1 +
    BallotBox·size·div[Election·seats + 1] and
    fullQuota = 1 + BallotBox·size·div[Election·
        constituencySeats + 1]
  Election·method = Plurality implies quota = 1 and
      fullQuota = 1

  // All ties involve equality between at least one winner
      and at
  // least one loser
  all w: Candidate | some l: Candidate | w·outcome =
      TiedWinner and
    (l·outcome = TiedLoser or l·outcome = TiedSoreLoser)
        implies
    (#l·votes + #l·transfers = #w·votes + #w·transfers)
  all s: Candidate | some w: Candidate | w·outcome =
      TiedWinner and
    (s·outcome = SoreLoser or s·outcome = TiedLoser)
        implies
    (#s·votes = #w·votes) or
    (#s·votes + #s·transfers = #w·votes + #w·transfers)
  // When there is a tied sore loser then there are no non-
      sore losers
  no disj a,b: Candidate | a·outcome = TiedSoreLoser and
    (b·outcome = TiedLoser or
    b·outcome=Loser or b·outcome=EarlyLoser or
    b·outcome = EarlyLoserNonTransferable)
  // For each Tied Winner there is a Tied Loser
  all w: Candidate | some l: Candidate | w·outcome =
      TiedWinner implies
    (l·outcome = TiedLoser or l·outcome = TiedSoreLoser)
  // Tied Winners and Tied Losers have an equal number of
      votes
  all disj l,w: Candidate |
```

```
    ((l·outcome = TiedLoser or l·outcome = TiedSoreLoser)
       and
    w·outcome = TiedWinner) implies
    #w·votes + #w·transfers = #l·votes + #l·transfers
  // Compromise winner must have more votes than any tied
     winners
  all disj c,t: Candidate | (c·outcome = CompromiseWinner
     and
    t·outcome = TiedWinner) implies
    #t·votes + #t·transfers < #c·votes + #c·transfers
  // Winners have more votes than non-tied losers
  all w,l: Candidate | w·outcome = Winner and
    (l·outcome = Loser or l·outcome = EarlyLoser or
    l·outcome = SoreLoser or
    l·outcome = EarlyLoserNonTransferable or
    l·outcome = EarlySoreLoser or
    l·outcome = EarlySoreLoserNonTransferable)
    implies
    ((#l·votes < #w·votes) or
    (#l·votes + #l·transfers < #w·votes + #w·transfers))
  // For each Tied Loser there is at least one Tied Winner
  all c: Candidate | some w: Candidate |
    (c·outcome = TiedLoser or c·outcome = TiedSoreLoser)
    implies w·outcome = TiedWinner
}

-- The Ballot Box
one sig BallotBox {
  spoiltBallots: set Ballot,
    -- empty ballots excluded from count
  nonTransferables: set Ballot,
    -- ballots for which preferences are
    -- exhausted/wasted votes
  size: Int
    -- number of unspolit ballots
}
{
```

```
  no b: Ballot | b in spoiltBallots and b in
      nonTransferables
  size = #Ballot - #spoiltBallots
  all b: Ballot | b in spoiltBallots iff #b.preferences = 0
  // All non-transferable ballots belong to an non-
      transferable surplus
  all b: Ballot | some c: Candidate | b in nonTransferables
       implies
  b in c.wasted
}

-- An Electoral Constituency
one sig Election {
  seats:             Int,    -- number of seats to be filled
  constituencySeats: Int,    -- full number of seats
  method:       Method -- type of election; PR-STV or
      plurality
}
{
  0 < seats and seats <= constituencySeats
  seats < #Candidate
}
```

C.2 Lemmas

```
assert honestCount {
  all c: Candidate | all b: Ballot | b in c.votes + c.
      transfers
  implies c in b.assignees
}
check honestCount for 15 but 6 int

assert atLeastOneLoser {
  0 < #Scenario.losers
}
check atLeastOneLoser for 15 but 6 int
```

```
assert atLeastOneWinner {
  0 < #Scenario·winners
}
check atLeastOneWinner for 14 but 6 int

assert plurality {
  all c: Candidate | all b: Ballot | b in c·votes and
  Election·method = Plurality implies c in b·preferences·
      first
}
check plurality for 18 but 6 int

assert pluralityNoTransfers {
  all c: Candidate | Election·method = Plurality implies
  0 = #c·transfers
}
check pluralityNoTransfers for 13 but 7 int

assert wellFormedTieBreaker {
  some w,l : Candidate | (w in Scenario·winners and
  l in Scenario·losers and
  #w·votes = #l·votes and #w·transfers = #l·transfers)
      implies
  w·outcome = TiedWinner and
  (l·outcome = TiedLoser or l·outcome = TiedSoreLoser)
}
check wellFormedTieBreaker for 18 but 6 int

assert validSurplus {
  all c: Candidate | 0 < #c·surplus implies
  (c·outcome = WinnerNonTransferable or
  c·outcome = QuotaWinnerNonTransferable or
  c·outcome = SurplusWinner or
  c·outcome = AboveQuotaWinner or
  c in Scenario·eliminated)
}
```

```
check validSurplus for 16 but 6 int

-- Advanced Lemmas
-- Equal losers are tied or excluded early before last
   round
assert equalityofTiedWinnersAndLosers {
  all disj w,l: Candidate | w in Scenario·winners and
  l in Scenario·losers and
  #w·votes + #w·transfers = #l·votes + #l·transfers
     implies
  w·outcome = TiedWinner and
  (l·outcome = TiedLoser or
  l·outcome = TiedSoreLoser or
  l·outcome = EarlyLoserNonTransferable or
  l·outcome = EarlySoreLoserNonTransferable or
  l·outcome = EarlyLoser)
}
check equalityofTiedWinnersAndLosers for 16 but 7 int

-- No lost votes during counting
assert accounting {
  all b: Ballot | some c: Candidate | 0 < #b·preferences
  implies
  b in c·votes and c in b·assignees
}
check accounting for 16 but 6 int

-- Cannot have tie breaker with losers both above and below
    threshold
assert tiedWinnerLoserTiedSoreLoser {
  no disj c,w,l: Candidate | c·outcome = TiedSoreLoser and
  w·outcome = TiedWinner and (l·outcome = Loser or
  l·outcome = TiedLoser)
}
check tiedWinnerLoserTiedSoreLoser for 6 int

-- Compromise winner must have at least one vote
```

```
assert validCompromise {
  all c: Candidate | c·outcome = CompromiseWinner implies
  0 < #c·votes + #c·transfers
}
check validCompromise for 6 int

-- Quota winner needs transfers
assert quotaWinnerNeedsTransfers {
  all c: Candidate | c·outcome = QuotaWinner implies
  0 < #c·transfers
}
check quotaWinnerNeedsTransfers for 7 int

-- Sore losers below threshold
assert soreLoserBelowThreshold {
  all c: Candidate | c·outcome = SoreLoser implies not
  (Scenario·threshold <= #c·votes + #c·transfers)
}
check soreLoserBelowThreshold for 10 but 6 int

-- Possible outcomes when under the threshold
assert underThresholdOutcomes {
  all c: Candidate |
  (#c·votes + #c·transfers < Scenario·threshold)
  implies
  (c·outcome = SoreLoser or
  c·outcome = TiedSoreLoser or
  c·outcome = TiedWinner or
  c·outcome = EarlySoreLoserNonTransferable or
  c·outcome = EarlySoreLoser or
  c·outcome = CompromiseWinner or
  (Election·method = Plurality and c·outcome = Winner))
}
check underThresholdOutcomes for 10 but 6 int

-- Tied Winners have equality of votes and transfers
assert tiedWinnerEquality {
```

```
  all a,b: Candidate | (a·outcome = TiedWinner and
  b·outcome = TiedWinner) implies
  #a·votes + #a·transfers = #b·votes + #b·transfers
}
check tiedWinnerEquality for 10 but 6 int

-- Non-negative threshold and quota
assert nonNegativeThresholdAndQuota {
  0 <= Scenario·threshold and 0 <= Scenario·quota
}
check nonNegativeThresholdAndQuota for 6 but 6 int

-- STV threshold below quota
assert thresholdBelowQuota {
  Election·method = STV and 0 < #Ballot implies
  Scenario·threshold <= Scenario·quota
}
check thresholdBelowQuota for 13 but 7 int

-- Plurality sore loser
assert pluralitySoreLoser {
  all c: Candidate | (c·outcome = SoreLoser and
  Election·method = Plurality) implies
  #c·votes < Scenario·threshold
}
check pluralitySoreLoser for 13 but 7 int

-- Plurality winner for a single seat constituency
assert pluralityWinner {
  all disj a, b: Candidate |
    (Election·method = Plurality and
    Election·seats = 1 and
    a·outcome = Winner) implies
    #b·votes <= #a·votes
}
check pluralityWinner for 2 but 7 int
```

```
-- Length of PR-STV ballot does not
-- exceed number of candidates
assert lengthOfBallot {
  all b: Ballot | Election·method = STV implies
  #b·preferences <= #Candidate
}
check lengthOfBallot for 7 int

-- Quota for a full election is less than for a by-election
assert  fullQuota {
  Scenario·fullQuota <= Scenario·quota
}
check fullQuota for 7 int

-- All transfers have a source either from
-- a winner with surplus or by
-- early elimination of a loser
assert transfersHaveSource {
  all b: Ballot | some disj donor, receiver : Candidate |
  b in receiver·transfers
  implies b in donor·votes and
  (donor in Scenario·winners or donor in Scenario·
     eliminated)
}
check transfersHaveSource for 7 int

-- No missing candidates
assert noMissingCandidates {
  #Candidate = #Scenario·winners + #Scenario·losers
  }
check noMissingCandidates for 7 int

-- Spoilt votes are not allocated to any candidate
assert handleSpoiltBallots {
  no c : Candidate | some b : Ballot | b in c·votes and
  b in BallotBox·spoiltBallots
}
```

```
check handleSpoiltBallots for 7 int

-- The End
```

The End.