Vótáil: PR-STV Ballot Counting Software for Irish Elections

Dermot Cochran and Joseph R. Kiniry

IT University of Copenhagen, Denmark
(dero,kiniry)@itu.dk

Abstract. *Vótáil* is an open source Java implementation of Irish Proportional Representation by Single Transferable Vote (PR-STV). Its functional requirements, derived from Irish electoral law, are formally specified using the Business Object Notation (BON) and refined to a Java Modeling Language (JML) specification. Formal methods are used to verify and validate the correctness of the software. This is the first public release of a formally verified PR-STV open source system for ballot counting and the most recent of only about half a dozen releases of formally verified e-voting software.

1 Introduction

Vótáil is the Irish Gaelic word for Vote. Many aspects of the election process are apparently suitable for automation. For example, voter registration records are stored in computer databases, postal voting has sometimes been replaced with internet voting, paper ballots with voting kiosks, and ballot counting is sometimes done by machine. However, many attempts to introduce electronic voting have failed, or at least received much criticism, due to a litany of software and hardware errors, many of which are avoided through the use of formal methods.

To give evidence to this claim, as well as to continue to play a scientist-activist role in the public eye, we present the *Vótáil* system: a verified implementation of Irish PR-STV. This work in novel on several fronts. First, *Vótáil* represents one of the largest and most complex case studies in the verification of an object-oriented system. As such, it helps validate our verification-centric approach to software design and implementation [15,16]. Second, as the case study is directly relevant to one of *society's* critical systems, it represents an opportunity to influence the mindset of not just the software developer interested in quality, but in the worldview of the politician interested in trusted elections and the voter passionate about their government. Finally, this work also pushes some of the object-oriented software verification's community's tools to their limits, thus shows us where we have succeeded, and where we have failed, in our focus on usable, usable, and powerful verification tools. Consequently, we expect that this system, like others of a similar ilk (e.g., the KOA tally system [11,14]) will be used to benchmark new verification tools for object-oriented software.

Before discussing the process, tools, and techniques used in this case study, some context in election system and voting in Ireland is necessary to appreciate its size and complexity.

1.1 Electronic Voting in Ireland

In 2009, the Irish government decided to save costs by disposing of its current generation of direct recording electronic (DRE) voting machines. The decision to stop using electronic voting was due to technical problems and to more general concerns about the security of electronic voting. The current political consensus is that electronic voting (in its current form, e.g., DRE machines) will not be used in the Republic of Ireland.

In the authors' personal experience, Irish citizens are happy with the paper-based voting process and the hand-counting of votes that takes several days with gradual reporting of results through the media. There are usually one or two closely contested seats that require a full manual recount with additional observers.

After opposition parties voiced concerns about electronic voting, the Irish Government established an independent ad-hoc Commission on Electronic Voting (CEV). One of the recommendations of the CEV was the use of a Voter Verified Paper Audit Trail (VV-PAT) [10]. A VV-PAT is a printed copy of the electronic ballot which is used for manual recounts and audits of the result. However, the cost of adding a VV-PAT as well as other improvements to the software of the voting machines was seen as prohibitive, making it more economical for Ireland to abandon the use of electronic voting.

1.2 Voting Scheme

The Republic of Ireland uses Proportional Representation by Single Transferable Vote (PR-STV) for its national, local and European elections. PR-STV is a ranked choice voting system, in which each voter ranks the candidates from first to lowest preference. A quota is the minimum number of seats needed to win one seat. If a candidate has more than the quota, the surplus votes are transferred pro-rota to the next highest preference on the ballot. If not enough candidates have a quota, then the lowest candidate is excluded and his or her ballots transfered to the next highest preference.

Oireachtas Éireann, the National Parliament of the Republic of Ireland, has two chambers. The people directly elect Dáil Éireann, the lower chamber of the Oireachtas, for a term of up to five years by a quota-based single transferable vote system in multi-seat constituencies. The upper house, called the Seanad, also uses PR-STV, but uses postal ballots and is indirectly elected, except for the six seats elected by university graduates. The Seanad has an advisory role and a smaller electorate. It is therefore a lesser target for electoral fraud, and a low risk election, so it is more interesting and important to look at verification of Dáil elections.

Note that Irish legislation uses the term 'vote' as a noun to mean the contents of a ballot paper rather than as a verb for the action of casting a ballot [9]. For the purpose of clarity, *vote* means the full set of candidate preferences recorded by a voter at an election.

The political significance of lost, corrupted or altered votes depends on the type of voting system (e.g., STV) and the closeness of the election race. In PR-STV, it is not unusual to see the final seat in a multi-winner constituency determined in the last round of counting by a small number of votes.

Manual recounts are often called for closely contested seats, as the results often vary slightly, indicating small errors in the manual process of counting votes. Paper-based voting with counting by hand is popular in Ireland, and recent attempts at automation were frustrated by subtle logic errors in the ballot counting software [6]. The logic errors exist, in part, due to the complexities and idiosyncrasies with regard to tie breaking, especially involving the rounding of vote transfers. Other errors relate to the rounding up or down of ballot transfers and to the randomisation effect of ballot shuffling, which does not have have a precise legal definition. As every ballot and every preference on a ballot can make a difference when the last seat of a multi-seat constituency is being decided, these subtle errors can have an enormous effect on the outcome of an election.

There has been some desire in Ireland to simplify matters. Referenda to introduce plurality (first past the post) voting were rejected twice by the Irish electorate, in 1959 and again in 1968 [19]. Since then, there have been no further legislative proposals to change the voting scheme used in Ireland.

The following are selected quotes from the CEV report on the previous electronic voting system used in Ireland [10]:

> – Design weaknesses, including an error in the implementation of the count rules that could compromise the accuracy of an election, have been identified and these have reduced the Commission's confidence in this software.
> – The achievement of the full potential of the chosen system in terms of secrecy and accuracy depends upon a number of software and hardware modifications, both major and minor, and more significantly, is dependent on the reliability of its software being adequately proven.
> – Taking account of the ease and relative cost of making some of these modifications, the potential advantages of the chosen system, once modified in accordance with the Commission's recommendations, can make it a viable alternative to the existing paper system in terms of secrecy and accuracy.

Thus, Ireland wishes to keep its current complicated voting scheme, is critical of the existing attempts to implement that scheme in e-voting, but keeps the door slightly ajar for the introduction of e-voting in the future. Consequently, we believe that this combination of factors makes our work timely, relevant, and, potentially, high-impact. In the end, our meta-goal is to show that, if a handful of researchers working in their spare time can design and implement a

verified voting system for one of the most complex voting schemes in the world, citizens and governments must *demand* that their e-voting systems are of at least this level of quality. *Verified elections effected, in part, through formally verified voting software are mandatory for future e-democracies.*

1.3 Related Work

The authors are unaware of any peer-reviewed published related work on the formal specification and implementation of PR-STV. We are aware of some unpublished or unfinished work relating to previous attempts at formalization of PR-STV, including some Prolog work by Naish and an implementation of the Scottish STV system in CLEAN by researchers at the Radboud University Nijmegen. The only peer-reviewed published related work of interest is a protocol for the tallying of encrypted STV ballots [20] and verifying properties of voting protocols, not software (e.g., several papers by Ryan [7]).

There have been numerous pieces of work on contract-guided or refinement-centric software verification, particularly from the correctness-by-construction community. Only a few focus on the particular challenges inherent in modern object-oriented systems (e.g., the work of Nunes and colleagues [18]), and none that we are aware of include support for traceable refinement from requirements and features to verified software.

Finally, some work on the use of logics to understand and reason about law is relevant, e.g., the work of van der Meyden has influenced us [21]. We do not attempt to use such (deontic) logics in our refinement from law to formal models, though doing such may improve the quality and correctness of our specification and its refinement.

1.4 Outline of Paper

The rest of the paper is organized as follows. Section 2 describes our methodology for refinement of requirements from electoral law to Java software. Section 3 contains a summary of the requirements and features demanded for PR-STV ballot counting. Section 4 reviews the formal specification of PR-STV. Next, in section 5, the verification and validation of the software system are detailed. Finally, section 6 concludes the paper with some reflections.

2 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.

2.1 Business Object Notation

Business Object Notation (BON) provides a high-level object orientated description of a system [22]. 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. Specifications written in formal BON are essentially semantic dependent types. Refinement from informal to formal BON is described in the aforementioned text and supported by our BONc tool suite[1].

2.2 Java Modeling Language

The Java Modeling Language (JML) is a formal behavioral interface specification language used to specify the behavior of Java software [17]. It extends Java with annotations for specifying simple formal statements in a design-by-contract (DBC) style [2] and model-based specifications a la Larch [1]. Informal BON is either refined to a formal specification in formal BON or directly to a formal object-oriented specification language such as JML. Support for performing and checking such refinements is provided by our Beetlz tool[2].

2.3 A Verification-centric Development Process

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, we follow a strict design-by-contract based approach to software engineering.

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 1 provides an overview of these artifacts and their interrelationships. Details of this process and how these refinements are represented and reasoned about is not the focus on this paper. The interested reader is encouraged to examine our other published work on this front [15,16].

3 Requirements for PR-STV Ballot Counting

In addition to the general requirements for e-voting, like ensuring privacy of the voter and accuracy of the count, electoral-specific requirements are also derived from electoral law and government regulations about the counting of votes. In

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

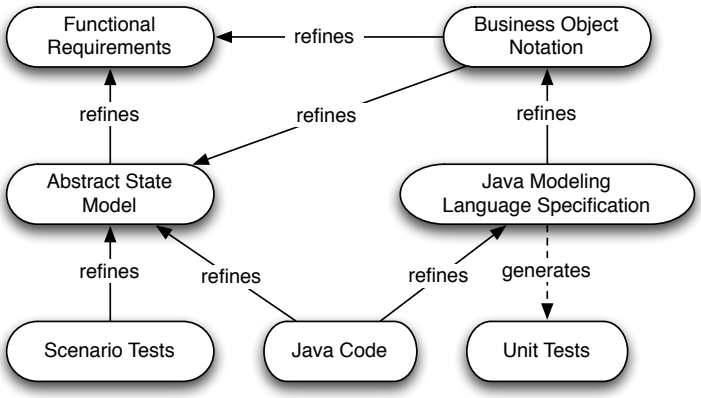

Fig. 1. Relationships between software engineering artifacts.

the Irish context, these requirements come from a commission on voting and electoral law and the electoral act itself.

We insist that a tracable interpretation of these requirements and features from law or similar to a concrete software system is mandatory.

3.1 Quality Requirements

The Commission on Electronic Voting (CEV) laid down several guidelines for development of electronic voting systems, including the following [10]:

- clear definition of functional requirements and specifications
- robust and formal approach to design and development
- separation of critical concerns, e.g., voter registration and ballot counting
- publication or public inspection of the source code (preferably open source)
- open public testing of the system

The pilot e-voting project in Ireland failed to meet any of these criteria, leading to its rejection, whereas Vótáilfulfills all five of the quality requirements listed above.

3.2 Functional Requirements in Electoral Law

The 1992 Electoral Act, including subsequent amendments, and the Commentary on Count Rules issued by the CEV [8], is the starting point for our requirements analysis. In previous work 39 semi-formal statements are used to describe these functional requirements for ballot counting in elections to the Dáil [4].

A few example formal statements from our previous work are listed and cross-referenced, as shown in Table 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.

Table 1. Cross-referencing functional requirements and the law.

ID	Functional Requirement	Section	Item	Page
8	If the number of continuing candidates is equal to the number of seats remaining unfilled, or the number of continuing candidates exceeds by one the number of unfilled seats or there is one unfilled seat, then do not distribute any surplus unless it could allow one or more candidates with at least one vote to save their deposits.	4	2	15
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
	...			

4 Formal Specification

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 of are done via the use of ASMs.

4.1 Abstract State Machine

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 Figure 2) is more complex and describes more detailed sub-states and transitions within the `COUNTING` state.

4.2 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.

Each election state has a number of invariants that must hold. For example, in the fragment of JML seen in Figure 3, the *Finished* state has as an invariant

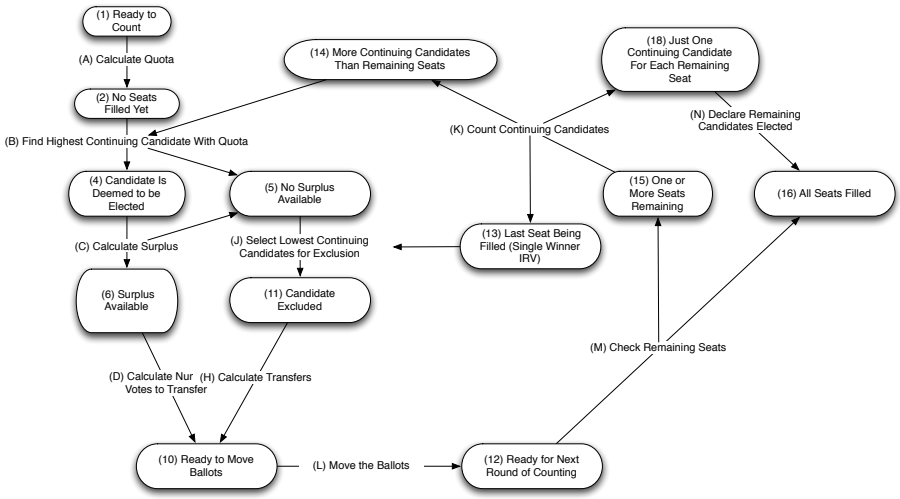

Fig. 2. Abstract State Model, lower tier (sub-states)

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.

4.3 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, thus an example of how the invariants are preserved in the top-level state transitions is illuminating.

Example 1. Let E be the number of candidates elected (a variable), and let S be the number of seats being filled (a constant), and let R be the number of seats remaining to be filled, a function defined to mean $S - E$. Consider the transition in the example from the *Pre-Count* state to the *Counting* state.

At *Pre-Count*: $E = 0$, and for all candidates C, the status of C is *Continuing*. After transition to *Counting* $E = 0$. The new invariant that must hold is that E equals the number of *Candidate* objects C where C is *Elected*. All candidates have *Continuing* status to begin with, so $E = 0$ and the invariant is satisfied.

Likewise, consider the transition from the *Counting* state to the *Finished* state. Before the transition from *Counting* to *Finished*, the inner state machine

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

Fig. 3. A JML specification describing the number of candidates elected.

must be in the *End-of-Count* sub-state and $S = E$, therefore $R = 0$. The inner state machine can only reach the *End-of-Count* sub-state when $R = 0$. Therefore $E = S$ and the invariant for the *Finished* state holds.

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.

4.4 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).

4.5 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 4 shows an informal BON description of the Ballot Counting process.

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 Elected	The number of elected candidates is less than the number of open seats.
This Candidate Status is Elected	This candidate had 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.
More Continuing Candidates than Remaining Seats	The number of open seats is less than the sum of the number of elected candidates and the number of continuing candidates.
Seats Remaining	The number of elected candidates is less than the number of open seats.
One Continuing Candidate per each Remaining Seat	The sum of the number of elected candidates and the number of continuing candidates is equal to the number of open seats.

Table 2. Examples of invariants for each sub-state, translated from JML to English for the reader.

```
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
```

Fig. 4. An Informal BON description of the Ballot Counting class.

4.6 Refinement to JML Specification

The BON design contains 1 cluster[3] with 5 classifiers, 20 queries, 5 command 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.

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

Fig. 5. An example of a JML specification for a BON query.

Two examples of JML are shown in figures 5 and 6, 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.").

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

```
/**
 * 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);
```

Fig. 6. An example of a JML specification for a BON command.

Note also that the method signature specification in this example states in a precondition that none of the parameters can have null values.

4.7 Architecture

There are 10 Java classes in the implementation, representing the actors in the system, for example Ballots, Ballot Boxes and Candidates. Figure 7 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.

4.8 The Vótáil Theorem

Due to the manner in which the formal specification of the ballot counting algorithms is accomplished (section 3), and the aforementioned argument about the correctness of state transitions (section 4), we summarize via informal refinement the overall theorem expressed by this ballot counting system as the *Vótáil theorem*.

> **Vótáil Theorem:** 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.

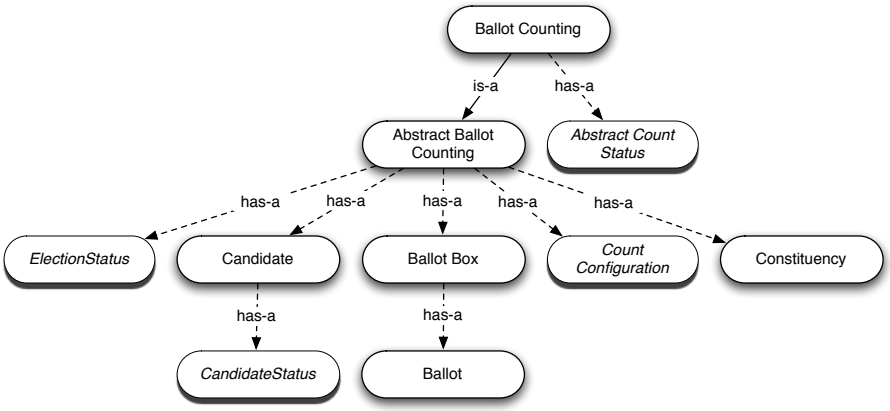

Fig. 7. Relationship between classes

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.

To derive the above "theorem," we essentially translate the formal preconditions of the first state of our ASM and the formal postconditions of the last state of our ASM back into English that is digestible by the voter.

5 Verification and Validation

Unlike the vast majority of e-voting systems available today, we have made our system's specification, implementation, validation, and verification available for public review.

5.1 Open Source Implementation

The source code is open source, under the terms of the MIT open source license, and is available via our Trac server[4]. The source code is managed using a subversion server hosted on our website[5] and developed using the Mobius Program Verification Environment (PVE)[6]. The source code contains 723 Java statements in 11 classes and 92 methods.

[4] https://trac.ucd.ie/repos/Votail/tags/0.0.1b
[5] https://trac.ucd.ie/repos
[6] mobius.ucd.ie

5.2 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.

The remaining 3 percent of code is accounted for by non-executable statements, such as declarations of constants that were either not exercised directly by the unit tests, or not measured by the coverage tool. We also run all scenario tests with JML runtime assertion checking (RAC) enabled to double-check their consistency and correctness.

5.3 Automatic Generation of Unit Tests

Just over 7,000 unit tests were generated from the JML specifications. The JMLUnit tool allows the automatic generation of unit tests [3]. JMLUnit requires guidance on which data types and values are interesting for testing and will then generate tests for each precondition, postcondition and invariant for all permutations of the test data. The interesting values of this case study were derived manually via a careful examination of the Vótáil architecture and its legal requirements.

5.4 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 [5]. 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.

ESC/Java2 is used to both type check the JML specifications and to check that the Java implementation fulfills these specifications. When used carefully, ESC/Java2 performs full function verification, as we have done here. This means that, once ESC/Java2 says that a method in Vótáil is correct, then the implementation of that method fulfills its specification for all possible input values on all possible execution paths. This is an extremely strong guarantee, much stronger than even the comprehensive testing that we have done.

This verification is complemented by the aforementioned testing because ESC/Java2 is neither sound nor complete. While we have used its functionality to check that specifications are sound [12] 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 [13], only via rigorous, well-designed testing are we assured that the system is functioning correctly in an actual execution environment.

[7] www.eclemma.org
[8] www.eclipse.org

5.5 Continuous Testing

Every change to the source code committed to the version control repository causes the 7,000+ tests to be run automatically. The test results can be seen at the project website. We are using the open source Hudson Extensible Continuous Integration Server[9] to checkout the latest source code from subversion, run the tests, and summarize the results.

5.6 Beta Release

Vótáil is open source and its test results are public. The beta release is available from the project website[10].

6 Results and Conclusions

We have shown how to specify, implement, validate, and verify, in a traceable fashion, a complex voting scheme such as PR-STV using formal methods. To accomplish such requires a combination of rigorous process, delicate specification techniques, and a novel combination of quality tools. Still, there are a number of problems with our approach and some next steps are critical in realizing trusted elections.

6.1 Near Future Work

Firstly, we seek to identify the optimal minimal number of test cases involving different combinations of ballots to fully test any voting scheme. If there are S seats and C candidates, then how many equivalent election results are possible, where equivalent means either reordering of candidates or magnifying the numbers of ballots in each permutation. This challenge is discussed in a technical report available on the project website.

Secondly, we would like to compare our work with other rigorous implementations of tallying software. Unfortunately, there are no publicly available verified implementations of PR-STV for comparison with Vótáil. The authors have attempted to collaborate with other groups that claim to have work related to us. In all cases to date, it seems that test data and specifications have neither been published nor archived, so we claim that ours is the first publicly available release of formally specified and verified PR-STV software, developed independently of any previous attempt.

[9] http://hudson-ci.org/
[10] http://www.kindsoftware.com/products/opensource/Votail

6.2 Reflections

Voting is but one component of the election process. One of the benefits of electronic voting it it becomes feasible to use more advanced voting schemes that might otherwise take weeks to count by hand.

We claim 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. This suggests that using a combination of cryptographic schemes with PR-STV for electronic voting will make for an even more difficult verification challenge.

Ireland's Commission on Electronic Voting laid down several recommendations for future use of electronic voting, including the following, some of which were mentioned in section 3:

- clear definition of requirements and specifications,
- robust and formal approach to design and development,
- separation of critical concerns (election management, count rules, vote file, etc.),
- the appropriate use of open source methods,
- publication or public inspection of the source code,
- open public testing of the vote recording software and the vote counting software via an on-line web interface designed to simulate the hardware interfaces of the system, and
- full and formal process of requirements capture and functional specifications for any proposed new system.

This leads us to conclude that the next generation of electronic voting systems in Ireland (if any), will be developed in open source using formal methods, and that each functional module (e.g., ballot counting) will be developed and tested independently. At this time Vótáil is intended to be only a reference implementation that is formally guaranteed to give the correct results for any valid set of ballots. Its use in actual elections is not suggested without further work on verifying inputs and outputs of the system.

Existing verification tools do not yet provide full verification for systems written in Java. Furthermore, as of yet there is no full functional verification tool which has been fully verified in itself. Since we cannot yet achieve full assurance of verification, then we are not yet ready for electronic voting, except for very low-risk elections for example, labor unions or youth/student elections.

Verifiable Elections Verifiable Elections are important. Counting of ballots is only one facet of the entire process, but is a critical component. In small elections, it is feasible to count and recount votes by hand, but the cost of manual

counting and of managing paper ballots in a central facility does not scale for larger populations. Secure storage of ballot boxes in a central facility is often expensive, but PR-STV requires central tallying of ballots in each constituency. Some people might believe that it is less expensive to count votes by hand than to use electronic voting machines. However, if Ireland or other countries like it decide to reduce the size of its parliament, and therefore increase the size of its constituencies, then manual counting starts to become more expensive.

The process of specifying and formalizing the Irish PR-STV count process took almost two man-years of work to complete. As it is the focus of a critical societal system and represents a one-off cost that can be extended and customized to work with other variants of PR-STV, we believe that this is a very reasonable amount of labor.

Critical systems must be designed and constructed with care and consideration. Dependable software engineering techniques are mandatory. If electronic voting is to be used at all, then the software design must be flawless. A typical argument against the use of formal methods is cost and time. This case study shows that the resources necessary are not large and we must balance the cost of formal methods against the cost of rectifying design flaws in electronic voting machines as well as the cost of having to re-run an election. In the end, formal methods look to be faster, cheaper, and better in this particular domain.

7 Acknowledgments

This work is being supported by IT University of Copenhagen, the European project Mobius within the IST 6th Framework and national grants from Science Foundation Ireland including LERO CSET and LERO Graduate School of Software Engineering. This paper reflects only the authors' views and the EU is not liable for any use that may be made of the information contained therein.

References

1. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik 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.
2. 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.
3. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the European Conference on Object-oriented Programming ECOOP 2002*, volume 2374 of *Lecture Notes in Computer Science*. Springer–Verlag, 2006.
4. D. Cochran. Secure internet voting in Ireland using the Open Source Kiezen op Afstand (KOA) remote voting system. Master's thesis, University College Dublin, March 2006.

5. D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362, pages 108–128. Springer, 2004.
6. L. Coyle, P. Cunnigham, and D. Doyle. Secrecy, accuracy and testing of the chosen electronic voting system: Reliability and accuracy of data inputs and outputs, December 2004.
7. Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying properties of electronic voting protocols. In *Proceedings of IAVoSS Workshop On Trustworthy Elections (WOTE)*, 2006.
8. Department of Environment and Local Government, Commission on Electronic Voting. Count requirements and commentary on count rules, 23 June 2000.
9. Department of Environment and Local Government, Commission on Electronic Voting. Count requirements and commentary on count rules, section 16, 23 June 2000.
10. Department of Environment and Local Government, Commission on Electronic Voting. Final Report of Commission on Electronic Voting, July 2006.
11. Fintan Fairmichael. Full verification of the KOA tally system, May 2005.
12. Mikoláš Janota, Radu Grigore, and Michał 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.
13. Joseph Kiniry and Alan 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.
14. Joseph Kiniry, Alan Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In *Proceedings of Trustworthy Global Computing*, 2006.
15. Joseph Kiniry and Daniel Zimmerman. A verification-centric software development process for java. In *Proceedings of the 9th International Conference on Software Quality (QSIC 2009)*, Jeju, Korea, August 2009.
16. Joseph R. Kiniry and Daniel M. Zimmerman. Secret ninja formal methods. In *Proceedings of the Fifteenth International Symposium on Formal Methods (FM)*, volume 5014 of *Lecture Notes in Computer Science*, 2008.
17. 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.
18. Isabel Nunes, Antónia Lopes, and Vasco T. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Selected Papers from the 9th International Workshop on Runtime Verification (RV)*, volume 5779 of *Lecture Notes in Computer Science*. Springer–Verlag, 2009.
19. R. Sinnott. *Irish voters decide: Voting behaviour in elections and referendums since 1918*. Manchester Univ Press, 1995.
20. V. Teague, K. Ramchen, and L. Naish. Coercion-resistant tallying for STV voting. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*. USENIX Association Berkeley, CA, USA, 2008.
21. R. van der Meyden. A clausal logic for deontic action specification. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Symposium on Logic Programming*. MIT Press, 1991.
22. Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. The Object-Oriented Series. Prentice–Hall, Inc., 1995.