Formal Aspects of e-voting

Jordan Neill

B.A. (Mod.) Computer Science

Final Year Project, April 2008

Supervisor: Dr. Andrew Butterfield

Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____ April 29, 2008Jordan Neill

Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_____ April 29, 2008Jordan Neill

Abstract

Electronic voting systems are becoming increasingly common in important national level elections. This is happening despite regular discoveries of serious flaws in the software used in such systems.

Formal methods have been used successfully in the specification and verification of many critical systems to provide assurances about the correctness of the software.

The aim of this project is to develop a practical remote electronic voting system using formal methods. Additionally, the system should provide independent verification, meaning that voters can trust the result without needing to trust the software.

Acknowledgements

I would like to thank Dr. Andrew Butterfield for agreeing to supervise this project and for his sound advice throughout its completion. I would also like to thank Dermot Cochran and Dr. Joesph Kiniry from the KindSoftware research group at University College Dublin for their help with KOA and JML. Finally, I must thank my friends and family for their heroic level of support during the final stages of this project.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Aims 2

1.2.1 CianVotáil 3

1.3 Outline 3

I Background **5**

2 Voting **6**

2.1 Requirements of a democratic election 6

2.2 Electoral methods 9

2.3 Electronic elections 10

2.3.1 Public Bulletin Board 13

2.4 Cryptography 14

2.4.1 Public-key cryptography 14

2.4.2 Blind signature protocol 15

2.4.3 Zero-Knowledge Proofs 17

2.4.4 Homomorphic encryption 18

2.5 Kiezen op Afstand (KOA) 19

2.5.1 Overview 19
2.5.2 Problems 20
2.5.3 Votáil 22

3 Formal Methods 23
3.1 Overview 23
3.2 Irish VDM 24
3.3 Alloy 29
3.3.1 Alloy Specification Language 29
3.3.2 The Alloy Analyzer 31
3.4 JML 32
3.4.1 ESC/Java2 35

II CianVotáil 36

4 Design 37
4.1 Overview 37
4.1.1 Software modules 38
4.2 Ballot Collector Specification 39
4.2.1 Irish VDM 40
4.2.2 Alloy 41
4.2.3 JML Specification 42
4.3 Ballot Collector Server 43
4.3.1 REST 43
4.3.2 Ballot Collector API 45

5 Implementation 48

5.1 Data classes 48
5.2 DBBallotCollector 49
5.3 Web Service 50
5.3.1 BallotCollectorResource 50
5.4 Database 52
5.5 Configuration 52

6 Evaluation **54**
6.1 Election requirements 54
6.2 Formal Verification 55
6.3 Simplicity 56
6.4 Future Work 56
6.4.1 Tamper Proof Storage 56
6.4.2 Alternative Voter Agents 57
6.4.3 Tallier 58
6.4.4 Logging 58

7 Conclusion **59**

Bibliography **60**

A Tools **63**
A.1 Java 5 63
A.2 Java Persistence API (JPA) 67
A.3 Java Architecture for XML Binding (JAXB) 69
A.4 Java API for RESTful Web Services (JAX-RS) 69

B Voter Agent **71**

C Sample output **72**

C.1 Ballot Collector server 72

C.2 Configuration files 73

1

Introduction

1.1 Motivation

> "They will be used in the election and referenda after 2007. I am not going to scrap them. We have actually paid good Irish taxpayers money for them."
>
> Dick Roche, TD

This quote is taken from the transcript of a debates at Dáil Éireann in October 2006 [1]. The then Minister for Local Government, Dick Roche, was responding to questions about a report from the Commission on Electronic Voting (CEV) [21] regarding the €52m worth of voting computers purchased by the Irish government in 2000. The report recommended that the machines not be used in the upcoming Irish general election as was intended because the election management software was "not of sufficient quality to enable its use to be confidently recommended" and that "functional testing has revealed programming errors".

The voting machines under discussion were largely identical to those which had already been used in elections in the Netherlands. The Dutch government decertified their machines in October 2007 after fundamental system vulnerabilities were exposed by a group called "Wij vertrouwen stemcomput-

ers niet" (We don't trust voting computers). The group managed to undetectably modify the software on the machines, enabling them to manipulate and falsify election results [10].

These problems are by no means limited to one brand of machine. In 2003, software from a Diebold Elections Systems touch screen voting machine was leaked onto the Internet. The code was believed to be from the most common model used in the USA, with over 33,000 machines in circulation. Researchers from John Hopkins and Rice Universities reviewed the code and discovered significant flaws, perhaps the greatest of which being that voters could undetectably cast multiple ballots [16].

It is unsettling that such poor quality software could be used in systems as important as voting machines. The fact that these systems will in all likelihood be used anyway is downright alarming and provides the motivation for this project.

1.2 Aims

This project aims to use software engineering best practices to develop a reliable and trustworthy remote electronic voting system. The existing commercial solutions such as the Diebold system provide a guide for how *not* to go about doing this.

A first requirement for a trustworthy system is that the software be made open to independent scrutiny. Only a few selected reviewers had access to the source code in the CEV review, and the Rice/Hopkins evaluation was only possible due to an inadvertent leak. Making the software **open source** would maximise the opportunity for external review.

A goal of any software system should be **simplicity**. One of the problems identified by the Rice/Hopkins researchers was that the code was overly complex and poorly documented. It is harder for accidental errors or malicious code to remain undetected in a short simple code base.

The main area of interest of this project is that of formal methods. These are techniques used to mathematically verify the correctness of software. They are frequently used, and indeed sometimes legally mandated, in safety and security critical systems such as nuclear power stations and railway signalling. Since the election process is clearly a critical system, electronic voting software should be **formally verified**.

Even if the software could be proved beyond any doubt to be correct, elections could still be tampered with by any number of involved parties. It is therefore necessary that the entire process be **independently verifiable**, i.e., that it is possible to verify that the software behaved correctly through means independent from the software itself.

To summarize, the goal of this project is to develop a remote electronic voting system that is open source; as simple as possible; formally specified and verified; and independently verifiable.

1.2.1 CianVotáil

The original intent of this project was to fulfil these goals by formally verifying an existing system called, "Kiezen op Afstand" (KOA, literally translated from Ducth to "Remote Voting"). KOA is an open source remote voting system that is being presented as a platform for electronic voting research. It was decided, however, that KOA could not be easily adapted to meet the goal of simplicity and that it would be wiser to develop a brand new system.

This new system, CianVotáil (literally translated from Irish to "Remote Voting") is strongly influenced by the design of KOA.

1.3 Outline

This report is divided into two parts.

The first part contains the background for the project. Chapter 2 provides a detailed introduction to the requirements of electronic voting systems. It covers some of the problems inherant to electronic voting and explains some existing solutions. The KOA system, which this project is based on, is also detailed. Chapter 3 details the formal methods techniques used within this project and explains each of them through examples.

The second part is deals with the design and implementation of the CianVotáil system developed in this project. Chapter 4 describes the design of the system and highlights some of the formal specifications created. Chapter 5 describes the implementation of the system. Chapter 6 evaluates the success of CianVotáil and highlights some possibilities for future work. Chapter 7 provides a brief conclusion to the report.

Part I

Background

2

Voting

The process of conducting an election generally involves the following distinct stages.

- **Administration.** The specific format of the election is determined, such as the electoral method and the ballot design.
- **Registration.** A list of eligible voters is compiled.
- **Voting.** Ballots are collected from eligible voters.
- **Tallying.** The cast ballots are counted to produce a result.

These stages can and should be treated as separate concerns to be handled by independent systems. Although the administration and registration stages are extremely important, this paper will assume they are conducted properly and focus on voting and tallying.

2.1 Requirements of a democratic election

Despite the wide variety of democratic voting systems currently in use across the world, there is a generic set of criteria common to each, such as accuracy and voter privacy. Satisfying these criteria is a fundamental prerequisite in

the development of a successful electronic voting system, and indeed variations of these requirements are mentioned frequently in the e-voting literature [9] [5]

Accuracy. To ensure that the preferences of each voter are accurately recorded and counted, the election should ensure the following properties:

- **Cast as intended.** The ballot cast by a voter should match how they intended to vote. The voter should be confident that the ballot they are submitting contains the choices they want to make.
- **Recorded as cast.** The preference recorded must match the preference submitted by the voter. This requires that once a ballot is cast it cannot be altered or invalidated.
- **Counted as cast.** Every valid vote cast by a voter must be included in the tally. The count must not include invalid votes, or votes not actually submitted by a voter.

Democracy.

- **All eligible voters may vote.** Every eligible voter should have equal opportunity to cast their vote and the system should not hinder this ability.
- **Only eligible voters may vote.** In order to cast a ballot a voter must authenticate themselves as an eligible voter.
- **Voters can only vote once.** After an eligible voter has cast their ballot, it should be impossible for them to cast another.

Fairness.

- **Unbiased selection.** The candidates should be presented to the voter in an unbiased manner, for example displayed equally on a randomly ordered list.

- **No partial result during voting.** No information should be made available about the current state of the vote while polling is still under way as partial results could affect voter choice.

Privacy. For a true representation of the preferences of the electorate, it is essential to maintain voter privacy.

- **Secret ballot.** No link between a voter and their ballot may be maintained. It should be impossible for anyone, even an election official, to connect a cast ballot to its source. Even if no explicit pressure exists, the knowledge that their vote may be viewed by friends, relatives or co-workers could affect the choices that a voter declares.
- **Coercion-Free.** A voter should not be able to prove how they voted. If it were possible for a voter to convince a third party that they voted in a particular way, they could be subject to coercion through bribery or extortion.

Verifiability. In order to establish public trust and to ensure that the voting system operates correctly it is imperative that the process be independently verifiable.

- **Individual verifiability.** Each voter can verify that their vote is included correctly in the final tally.
- **Universal verifiability** Anyone can verify that all counted votes have been cast by eligible voters and have been counted correctly.

Some of these properties are, or at least appear to be, in conflict. For example, it would seem that the better a system can assure a voter that it has recorded their ballot correctly, the better that voter can convince others of how they voted. Overcoming these apparent conflicts to achieve the specified requirements is one of the reasons electronic voting is a challenging problem.

2.2 Electoral methods

This section provides a brief overview of some of the most common voting methods in use. These methods define how a voter's preferences are represented on a ballot and how the ballots are tallied to produce a result. They are highlighted here because, while an ideal electronic voting system would be flexible enough to handle a variety of methods, their different properties often influence the design of such systems.

Single winner. These schemes are used where there is only one position available, such as in presidential elections.

- **Plurality.** Often referred to as *first past the post* or *winner takes all*, this is the most common system, used for general elections in the UK and presidential elections in the USA. Voters select one candidate and the winner is the candidate with the largest number of votes.

- **Instant-runoff voting.** IRV is also known as alternate or preferential voting. Voters rank their choices, placing 1 next to their first preference, 2 next to their second preference and so on. Counting proceeds in rounds where the highest ranked candidate on each ballot is counted and the candidate with the majority of votes wins. If no candidate has a majority, the one with the lowest count is eliminated and their ballots are redistributed according to the next preference. The process is repeated until a candidate with a majority emerges. This is the method used to elect the President of Ireland and the Australian House of Representatives.

- **Range Voting.** Voters assign each candidate a score within a given range, e.g. 0-9. The candidate with the highest average score wins. To date no major elections have been conducted using range voting although **approval voting**, which is a form of range voting with a range of 0-1, was used by the IEEE (Institute of Electrical and Electronics Engineers, Inc.) from 1987 to 2002.

Multiple Winner. These methods are used in contests where there are multiple positions to be filled, usually, though not exclusively, to provide proportional representation.

- **Single transferable vote.** Like IRV, voters rank the candidates in an order of preference and there are multiple rounds of counting. At each round a candidate is elected if they reach a defined quota of votes. Their surplus votes, those beyond the quota, are transferred to other candidates based on the ballot's next preference. If no candidate is elected in a round, the candidate with the lowest count is eliminated and all their ballots are transferred according to the next preference. The rounds continue until all the available positions are filled. STV is used for elections in Ireland and thus is the voting method this paper is primarily interested in.

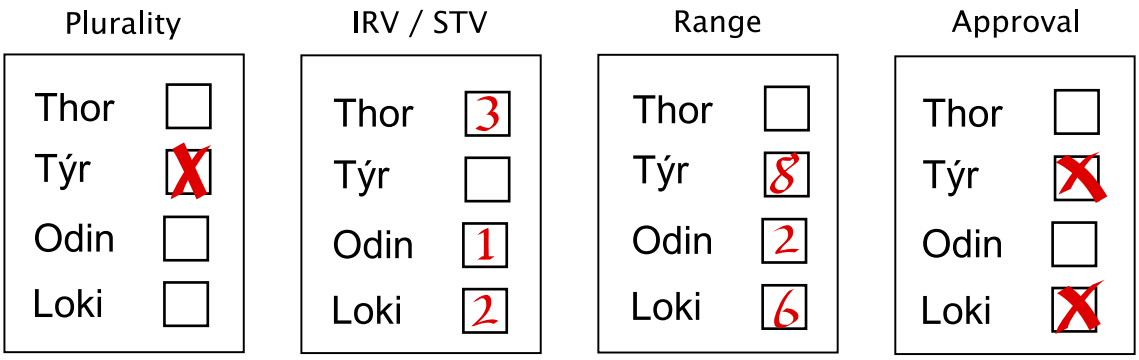

Figure 2.1: Electoral Methods

2.3 Electronic elections

The process of a traditional election should be familiar. The voter visits their local polling station where election officials guide them through the voting process. Their identity is checked and their name is marked off of a list of eligible voters before they are given a ballot slip to fill out. Voters fill out

their ballot slip in a private voting booth and place their completed ballot in a sealed ballot box. The boxes are opened after all official polling has closed and then the ballots returned are counted manually under public scrutiny.

While the general public feel they know and trust this manual system, it does have certain deficiencies such as a lack of individual verifiability and limited accessibility. Computers can be introduced to different degrees and at different stages to improve the voting process. At the most basic level they may be introduced purely at the counting stage by performing the tallying of digitized paper ballots. They may also be adopted at the ballot casting stage through the use of kiosk style direct-recording electronic (DRE) voting machines which store and aggregate electronic ballots. The third prinicpal way, and specifically the method chosen to be researched and developed in this project, is "remote voting". In this case votes are cast remotely over the internet to an election server which stores and aggregates electronic ballots.

The remainder of this section identifies some of the main limitations of the traditional system and outlines how computer based systems may be employed to solve these issues.

Efficiency. Computers are ideal for handling large volumes of data and performing repetitive tasks consistently. Thus, using computers to count ballots and produce election results can be seen as preferable to manual counting which is a slow, tedious and error prone activity.

Accuracy. In some cases, the rules governing the tallying of votes have been specified with a view to simplifying manual counting instead of producing the most accurate result. An example of this is the transfer of surplus votes in STV elections, where for practical reasons only some of the votes, effectively selected at random, are redistributed. This has the implication that counting the same set of ballots twice may produce different results. A computerized counting system could fairly redistribute every vote proportionally according to each ballot's next preference.

Accessibility. Paper based voting systems present every voter with the same

form of ballot. While this could be acceptable for the majority of voters, it could engender serious difficulties for other members of the electorate. Both the DRE and remote voting options present the ballot through a computer interface which could be adapted to the needs of the specific voter. Such a case would be where a voter with visual impairment uses a brail interface or screen reading software while a voter with a physical disability could use an alternate means of input. Another example would be where instructions are presented in the language of the voter's choice.

Fairness. A ballot in an Irish general election lists the candidates in alphabetical order by surname. Studies have shown an inherent bias towards candidates listed near the top of the ballot. A computer interface could increase fairness by presenting each voter with a ballot containing a random ordering of candidates.

Voter intent. The existing paper system gives no feedback to the voter about whether their ballot slip has been correctly completed. A computer interface can warn the voter that their ballot is invalid (e.g. they have *overvoted* on an IRV ballot by selecting more than one candidate).

Availability. To ensure "one person, one vote", a traditional election requires that voters cast their ballots at a designated polling station which holds a single list of eligible voters in that area. This would appear to disenfranchise voters with disabilities or those living abroad and violate the property of equal opportunity to vote. This problem has been mitigated by the introduction of postal votes which allow remote ahead of time voting. Remote electronic voting completely removes this obstacle by allowing voters to vote easily from anywhere they want.

Verification. Once placed in the ballot box, the traditional system provides no direct assurance to the voter that their ballot has been included correctly in the final result. They have to trust that poll workers conduct themselves properly and that official supervision prevents violations like manipulation of ballot boxes, intentional miscounting or simple human error. The ability of electronic voting systems to provide both individual and universal verification

is perhaps the most significant contribution such systems can make to the electoral process.

2.3.1 Public Bulletin Board

Electronically storing ballots presents an interesting possibility for allowing verification. After polling, all cast ballots could be presented in some form on a publicly accessible website (referred to as a Public Bulletin Board in the e-voting literature). This data could allow voters to both check that their ballots are present and correct and to conduct an independent tally of the votes. This would verify that the votes were recorded as intended and also counted as cast.

The problem is deciding what form the published ballots should take. Voters need to identify that their ballot are included in the published list without sacrificing privacy. It would be trivial to ensure verifiability and accuracy if a mapping between voters and their votes were to be published but this would obviously violate the secret ballot requirement. An alternative method would be to provide each voter with a receipt containing a code for their voting "transaction". Subsequently a mapping from codes to ballots could be published that would maintain the secret ballot property but leave voters open to coercion.

Even publishing the plaintext ballots anonymously can lead to coercion. For any ballot type there is a set number of permutations in which it can be completed and in some electoral methods this number is generally much greater than the number of possible voters. A coercer could require that a voter fill out their ballot with a particularly esoteric permutation and check that it is contained within the published list of ballots.

This is not a problem for most single winner electoral methods. Given an election with c candidates, there are only c ways to fill out a valid pluarlity ballot. A range voting ballot with a range of 0-9 can be filled out a large number of ways (namely, 10^c), but since the *score* allocated to each candidate

is independent of the others the ballot can be *debundled* into c different votes, each with only 10 permutations.

This is a significant problem, however, with the IRV/STV method. There are many ways to fill out an STV ballot, specifically $c!$, and it cannot be debundled since the complete rank ordering is required for tallying. The ballot for a constituency in a typical Irish general election might contain 15 candidates, meaning there are 1,307,674,368,000 ways to fill it out, while even the largest constituencies have electorates of less than 100,000.

This means that verification of STV elections cannot involve making the contents of the ballots public. The next section, on cryptographic methods in electonic voting, presents a possible solution to this problem by only publishing encrypted ballots (§2.4.4).

2.4 Cryptography

This section explains some cryptographic concepts and shows how they can be used to solve some of the contradictory requirements of voting systems.

2.4.1 Public-key cryptography

In symmetric cryptography, messages are encrypted using a secret key known only to the sender and receiver. The key needs to be shared with both parties, providing an opportunity for attackers to compromise privacy by intercepting the key during transmission.

Asymmetric, or public-key, cryptography involves creating two keys: a public key which is widely distributed, and a private key which is kept secret. A message encrypted with the public key can only be decrypted by using the private key.

A common public-key algorithm is RSA [22]. It is based on the idea that given

two very large prime numbers, p and q, calculating $N = pq$ is straightforward, finding p and q when given N is prohibitively difficult.

Asymmetric encryption algorithms are computationally intensive and it is only feasible to encrypt small messages. However, large messages can be encrypted with a symmetric key and the secret key can be transmitted to the receiver using asymmetric encryption. This is the protocol used in Secure Socket Layer (SSL) connections.

Asymmetric encryption can also be used to authenticate messages. A message can be digitally *signed* using the private key, and the public key can be used to verify the signature. Since asymmetric encryption is intensive, a hash of the message is often signed rather than the full message.

Asymmetric cryptography can be used in electronic voting to provide secure connections (HTTP over SSL), encrypt ballots and sign receipts.

2.4.2 Blind signature protocol

Blind signatures are a form of digital signature that allow a message to be signed without revealing it to the signer. They were introduced by David Chaum [3] who described them as being analogous to signing a sealed envelope lined with carbon paper. Although the contents are not revealed to the signer, they will bear the signature even after being removed from the envelope. Blind signatures have the following property:

$$\text{unblind}(\text{sign}(\text{blind}(m))) = \text{sign}(m)$$

Chaum realised blind signatures could be used for electronic voting systems and later Fujioka, Okamoto, and Ohta specified a protocol which used it [9]. The system involves multiple voters, a single validator and a single tallier. The validator maintains a list of eligible voters and signs at most one ballot for each. The tallier collects signed ballots and, after voting ends, publishes them and produces the election result.

The steps involved are:

1. A voter creates their ballot, encrypts it using a secret key, blinds it and sends it to the validator along with their authentication details ($blind(encrypt(b))$)
2. The validator checks that the voter has not already voted. If not it marks them as voted then signs and returns the ballot ($sign(blind(encrypt(b)))$)
3. The voter unblinds the ballot and sends it to the tallier ($sign(encrypt(b))$)
4. When voting has ended, the tallier publishes all the signed encrypted ballots
5. The voter checks that their ballot is present and sends the tallier their secret key. The tallier decrypts the ballots and tallies the result
6. The tallier publishes the encrypted ballots and the secret keys to allow independent verification of the results

This protocol provides accuracy and verifiability by allowing the voter to check that their vote is included exactly as they cast it at each stage of the process. It provides ballot secrecy, but does not prevent coercion - the voter can prove how they voted by providing their secret key between stages 4 and 5.

The problem exists that since the tallier knows who has not voted it could submit votes on their behalf. A solution would be to require that every voter vote, with abstaining voters submitting blank ballots. This, however, would be difficult to enforce on a large scale election. Another possibility would be to publish the names of voters who voted and encourage abstaining voters to check that they are not included. Unfortunately while a voter can prove they have voted (a signed ballot) they cannot prove they have not voted. An effective solution would be to have multiple talliers and require that a ballot

be signed by all of them. Voters would have different authentication details for each tallier thus requiring collusion between all the talliers to cast ballots for abstaining voters.

The encryption is not required for ballot secrecy. Imagine removing encryption from the system. The validator knows the identity of the voter but, due to the blinding, not how they are voting. The tallier knows how the voter is voting but not who they are. Even if the validator and tallier collude, it is not possible to link a voter with their ballot. Removing the encryption has the benefit of simplifying the system so that stages 5 and 6 are unnecessary and the voter's responsibility ends at stage 3. This, however, sacrifices fairness as an untrustworthy tallier could leak partial results while polling is still under way. More importantly it sacrifices voter verifiability.

The tallier both collects and counts the ballots. These are separate concerns that could be handled by different entities. The tallier entity in the description above could be renamed "ballot collector" and let an independent entity perform the count with the data published at stage 6.

2.4.3 Zero-Knowledge Proofs

A zero-knowledge proof is a protocol for a *prover* to convince a *verifier* that some statement is true without providing them with any information about the proof. They could be used, for example, to convince a verifier that the prover knows some secret without giving the prover any way to determine the secret themselves.

They are interactive proofs meaning that the active participation of the prover is required. They are also probabilistic in that in each round of the proof there is some small possibility of a false positive, i.e., that a cheating prover could convince the verifier.

Zero-knowledge proofs are of obvious value to electronic voting. For example, they can be used to convince voters that their encrypted ballot contains the

correct data without releasing giving them the ability to convince anyone else of how they voted.

2.4.4 Homomorphic encryption

Homomorphic encryption has the property that an algebraic operation can be performed on the encrypted plaintext without first decrypting it. For example, let $\varepsilon(x)$ be the encryption of x under some cryptosystem. The cryptosystem is homomorphic if algebraic operations $*$ and $\star$ exist such that

$$\varepsilon(a) * \varepsilon(b) = \varepsilon(a \star b)$$

Homomorphic encryption gives us a way to tally an election result without revealing the contents of any ballot. As a very simple example, imagine we a plurality election with two candidates, Týr and Odin, and a homomorphic encryption scheme where the operation $\star$ is addition. Voters vote for Odin by submitting -1, for Týr by submitting 1 and abstain by submitting 0. Odin wins the election if the result is less than 0, Týr wins if the result is greater than 0 and the election is a draw if the result is exactly 0. The voters submit their ballots in encrypted form using the election's public key. Say n voters cast the ballots $\{\varepsilon(b_1), \varepsilon(b_2), \ldots, \varepsilon(b_n)\}$ and these are all published. Anyone can calculate $\varepsilon(b_1) * \varepsilon(b_2) * \ldots * \varepsilon(b_n) = \varepsilon(b_1 + b_2 + \ldots + b_n) = \varepsilon(r)$, the encryption of the election result r. A trusted election authority can then decrypt this to give r and publish a zero-knowledge proof that $\varepsilon(r)$ is indeed an encryption of r.

Homomorphic encryption can solve the problem of coercion when publishing STV ballots as outlined in §2.3.1. Although the process is obviously much more complicated than the simple example presented here, a solution for STV elections is possible. [18] and [23] are two examples of such systems.

2.5 Kiezen op Afstand (KOA)

In 2003 the Dutch government commissioned a remote electronic voting system for use in the June 2004 European elections. It allows both online and telephone voting. The system, named "Kiezen op Afstand" (literally, "Remote Voting"), was evaluated by members of The Security of Systems (SoS) Group at the Radboud University Nijmegen [15]. The SoS group found some problems with the system and made a series of recommendations to the Dutch government.

One recommendation was that the source be made available to any interested party for review. This resulted in almost all of the system eventually being made open source under the GNU General Public License Version 2 (GPLv2). KOA is now being promoted as an experimental platfrom for electronic voting research and is maintained by the Systems Research Group (SRG) at University College Dublin.

Also on the recommendation of the SoS group, the Dutch government put the development of the vote tallying subsystem of KOA out to tender. The SoS group themselves applied for and won this contract and subsequently developed a formally verified counting system.

2.5.1 Overview

The voter registers in person at a government office with proof of identification. They chose a 5-digit PIN code intended to allow them to authenticate themselves on election day.

Later the voter is mailed an information pack containing a unique voter ID code, their previously chosen PIN and details of the SSL certificate of the election website. It also contains a candidate ID code for each candidate running in the election. Each candidate has multiple unique IDs in the system but only one is sent to the voter. To vote for a candidate the voter must use the ID that they have been given for that candidiate.

On election day, the voter visits the election website. They check the authenticity of the website by comparing the declared SSL certificate to the data from the information pack. Once they have verified the website's identity, the voter enters their voter ID code and PIN. The voter then navigates a series of simple web pages entering the ID codes for the candidates they want to vote for. This provides privacy since even if an attacker were to intercept communication between a voter and the election server they would gain no information about the vote. When completed, the server encrypts and stores the vote and provides the voter with a transaction code.

Once the election ends all the transaction codes are published on a public bulletin board so that voters can check that their codes are included. The ballots are decrypted by the election officials and then counted using the tallying subsystem.

2.5.2 Problems

While the vote tallying subsystem of KOA is formally specified and verified, the core vote collecting system only has a few sparse lightweight specifications. The original intent of this project was to formally specify and verify at least some part of the existing KOA core. After spending much time trying to get to grips with the KOA code base, it was decided that it would in fact be more useful, and straighforward, to build a new system from the ground up. This section provides some of the reasons for this decision.

Design compromises

The requirement that KOA support phone voting appears to have influenced several design choices that would not be considered "best practice" if creating a purely internet based system. For instance, a voter's password is the only mechanism they have to authenticate themselves, a secret known only to them that protects them from impersonation. To allow telephone voters to enter their password through a keypad, KOA limits it to a 5-digit PIN.

Obviously a longer alphanumeric password or passphrase would be much more secure; `aY}U1Xf?O` or `Bluecanary1nthe0utletbythelightswitch` are both much stronger passwords than `23837`.

In addition, the PIN code is sent by mail to the voter prior to the election. Not only does this provide the possibility of password interception, it also implies that, for a time at least, there is a relation stored between voters and their plain text passwords. Passwords should always be stored using a one way hash to keep them secret even from someone with access to the database.

Code complexity

KOA is built using Java 2 Platform, Enterprise Edition (J2EE) which provides support for developing Java server applications with APIs for middleware like database persistence and distribution. J2EE applications are deployed on *application servers* that provide support for things like sessions, security and scalability.

Central to J2EE applications are programmer defined business logic components called Enterprise Java Beans (EJB) which can be distributed and persisted and are managed by the application server.

The version of the platform used in KOA, J2EE 1.4, requires a substantial amount "boilerplate code" for each EJB component. For example, the current state of the KOA system is represented by a string and stored in the database using an EJB called `Koa_state`. Making this EJB persistable requires four classes (`Koa_state`, `Koa_stateBean`, `Koa_stateHome` and `Koa_stateKey`) and two configuration files (`KOAControllerEntityEJB.xml` and `koa-jboss.xml`) to give a total of 240 lines of code and configuration.[1]

The result of this is that the KOA core has nearly 500 classes containing over

[1] J2EE 1.4 has since been superseded by version 1.5. Renamed Java EE, this new version of the platform dramatically simplifies the development of enterprise applications. For comparison, the Java EE 5 version of `Koa_state` would be a single 20 line class with no configuration.

29,000 non-commenting source statements (NCSS). While this is not a very unusual size for a system of this complexity, especially one written in Java, it does make the system difficult to understand. This problem is exacerbated by the fact that the code is sparsely commented and mostly in Dutch.

2.5.3 Votáil

Researchers in the SRG group at UCD have created a formally verified system for tallying Irish STV elections. The system, called Votáil, was formally specified by Dermot Cochran [4] and implemented by Patrick Tierney [24]. It is based on official government specifications of the STV tallying process.

Both the Dutch tally system developed by the SoS group and Votáil are formally specified using JML and verified using ESC/Java2, tools which are explained in the next chapter.

"Software engineers want to be real engineers. Real engineers use mathematics. Formal methods are the mathematics of software engineering. Therefore, software engineers should use formal methods."

Mike Holloway, NASA

3 Formal Methods

3.1 Overview

Formal methods are techniques for assuring software quality through mathematically based specification and verification of software systems. The intent is that formal methods can increase our confidence that a system *does the right thing* by exposing false assumptions and inconsistencies at the design stage and ensure that the implementation correctly matches the specification.

By the standards of computer science, formal methods have a long history[1], but one of the most influential ideas has been Hoare Logic. Introduced by Tony Hoare in the 1969 paper titled "An axiomatic basis for computer programming" [11], the main feature of this logic is the Hoare triple which describes how the state of a program changes during execution. They are of the form

$$\{P\}\ C\ \{Q\}$$

which states that if the boolean precondition P holds, executing the code C will leave the program in a state that satisfies the postcondition Q.

The formal method techniques used in this project are model based methods which make use of these pre and postconditions.

[1] It could be argued they began in the 1840's when Ada Lovelace provided proofs for her programs written for Charles Babbage's analytical engine

3.2 Irish VDM

The Vienna Development Model (VDM) was developed at the IBM laboratory in Vienna in the 1960s and later standardised as VDM-SL (Specification Language). A model of the system state is created using mathematical constructs such as sets and maps. A state invariant is chosen which defines what constitutes a valid system state. Operations are then defined on the model, each of which has a pre and postcondition chosen so that the operation preserves the state invariant.

This report uses a variant of VDM called the Irish School, Irish VDM or *VDM*♣. The Irish school was founded in 1992 by Míchaél Mac an Airchinnigh [17] and Irish VDM is now maintained by the Formal Methods Group (FMG) at Trinity College Dublin. This section introduces Irish VDM through the development of a model based on coursework from the *3BA31 Formal Methods* course at TCD.

Modelling state

Imagine a subsystem of an air traffic control (ATC) system which has the requirement that every aircraft must have exactly one assigned on duty controller at all times while airborne.

We first design a system state, *ATCSys*, which is defined by: $\mathcal{O}$ the set of on duty controllers; $\mathcal{A}$ the set of airborne aircraft; and ψ the mapping from aircraft to their currently assigned controllers.

$$(\mathcal{O}, \mathcal{A}, \psi) \in ATCSys \;\; \widehat{=} \;\; \mathcal{P}\,Controller \times \mathcal{P}\,Aircraft \times (Aircraft \xrightarrow{m} Controller)$$

Aircraft and *Controller* are the *domains* of the system and are token types without any operations defined on them. $\mathcal{P}S$ denotes the powerset of the set S, i.e. the set of all subsets of S. $A \xrightarrow{m} B$ is partial mapping where elements of the set A map to at most one element of the set B.

State invariant

The requirement is that every aircraft must have exactly one on duty controller at all times while airborne. The design of the system ensures that no plane can have more than one controller but we still need to ensure that (i) all airborne aircraft have a controller responsible for them and (ii) that the controller is on duty. We therefore define the system invariant *inv*-ATCSys as

$$\textit{inv}\text{-ATCSys}(\mathcal{O}, \mathcal{A}, \psi) \mathrel{\widehat{=}} \mathcal{A} \subseteq \operatorname{dom} \psi \wedge \operatorname{rng} \psi \subseteq \mathcal{O}$$

$\operatorname{dom} \psi$ returns the domain of ψ, i.e. every aircraft with a controller, and $\operatorname{rng} \psi$ returns the range of ψ, i.e. every controller assigned to an aircraft. $\mathcal{A} \subseteq \operatorname{dom} \psi$ ensures (i) by checking that the set of airborne planes is a subset of planes with controllers. $\operatorname{rng} \psi \subseteq \mathcal{O}$ ensures (ii) by checking that the set of assigned controllers is a subset of the on duty controllers.

Operations

We then define operations for the system. Operations take a system state and, optionally, some arguments and return a new system state. For example the TakeOff operation indicates that an Aircraft a takes off.

$$\begin{aligned} \text{TakeOff} &: \textit{Aircraft} \to \textit{ATCSys} \to \textit{ATCSys} \\ \text{TakeOff}[a](\mathcal{O}, \mathcal{A}, \psi) &\mathrel{\widehat{=}} (\mathcal{O}, \mathcal{A} \cup \{a\}, \psi) \end{aligned}$$

This operation takes an *ATCSys* and an *Aircraft* and returns a new *ATCSys*. The new *ATCSys* has the same set of on duty controllers and the same assigned controllers mapping, but its set of airborne aircraft is the union of the previous set and the aircraft a.

Irish VDM differs from standard VDM in that there is no separate postcondition, the operation itself is an *explicit* prostcondition.

Preconditions

We must ensure that every operation preserves the state invariant. To do this we introduce preconditions on the operations such that given a valid state, if the precondition is true, the state returned by the operation will satisfy the invariant. For example, the precondition TakeOff must ensure that the aircraft a is assigned an on duty controller before it can take off.

$$\begin{aligned} \textit{pre}\text{-TakeOff} &: \textit{Aircraft} \to \textit{ATCSys} \to \mathbb{B} \\ \textit{pre}\text{-TakeOff}[a](\mathcal{O}, \mathcal{A}, \psi) &\triangleq a \in \operatorname{dom} \psi \end{aligned}$$

$a \in \operatorname{dom} \psi$ ensures that a has a controller, while $\operatorname{rng} \psi \subseteq \mathcal{O}$ from *inv*-ATCSys ensures that all assigned controllers are on duty.

Proof obligations

VDM♣ models are verified by performing mathematical proofs. This is referred to as *theorem proving*.

The first proof obligation is to show that there exists an initial state that satisfies the state invariant. This shows that the state invariant is not inconsistent, e.g. $x > 0 \ \wedge \ x < 0$. For the air traffic control system, an initial state could be:

$$\begin{aligned} \Sigma_0 &: \textit{ATCSys} \\ \Sigma_0 &\triangleq (\emptyset, \emptyset, \theta) \end{aligned}$$

i.e. an empty set of on duty controllers, an empty set of airborne aircraft and an empty assigned controllers mapping. We can prove that this satisfies the invariant

$$\begin{aligned} & \textit{inv}\text{-ATCSys}(\Sigma_0) \\ &= \textit{inv}\text{-ATCSys}(\emptyset, \emptyset, \theta) \\ &= \emptyset \subseteq \operatorname{dom} \theta \wedge \operatorname{rng} \theta \subseteq \emptyset \\ &= \emptyset \subseteq \emptyset \wedge \emptyset \subseteq \emptyset \\ &= \textsc{True} \qquad \clubsuit \end{aligned}$$

There is a proof obligation on every state operation. We need to show that, given any system that passes both the state invariant and the operation's precondition, performing the operation preserves the invariant. For example, for the TakeOff operation we are required to prove:

$$\textit{inv}\text{-ATCSys} \wedge \textit{pre}\text{-TakeOff}[a] \Rightarrow \textit{inv}\text{-ATCSys} \circ \text{TakeOff}[a]$$

This is of the form $P \Rightarrow Q$. To prove this we can assume P (since FALSE $\Rightarrow Q$ is TRUE for any Q) and use it to prove Q. In our case this gives us the following assumptions:

$$\begin{aligned}
&\textit{inv}\text{-ATCSys} \wedge \textit{pre}\text{-TakeOff}[a] \equiv \\
&\quad (A1)\ \ \mathcal{A} \subseteq \operatorname{dom} \psi \wedge \\
&\quad (A2)\ \ \operatorname{rng} \psi \subseteq \mathcal{O} \wedge \\
&\quad (A3)\ \ a \notin \mathcal{A} \wedge \\
&\quad (A4)\ \ a \in \operatorname{dom} \mu
\end{aligned}$$

A possible proof is then

$$\begin{aligned}
&\textit{inv}\text{-ATCSys} \circ \text{TakeOff}[a](\mathcal{A}, \mathcal{O}, \psi) \\
\equiv\ &\text{“}\textit{functional composition}\text{”} \\
&\textit{inv}\text{-ATCSys}(\text{TakeOff}[a](\mathcal{A}, \mathcal{O}, \psi)) \\
\equiv\ &\text{“}\textit{def. of}\ \text{TakeOff}\text{”} \\
&\textit{inv}\text{-ATCSys}(\mathcal{O}, \mathcal{A} \cup \{a\}, \psi) \\
\equiv\ &\text{“}\textit{def. of}\ \textit{inv}\text{-ATCSys}\text{”} \\
&\mathcal{A} \cup \{a\} \subseteq \operatorname{dom} \psi \wedge \operatorname{rng} \psi \subseteq \mathcal{O} \\
\equiv\ &\text{“A2”} \\
&\mathcal{A} \cup \{a\} \subseteq \operatorname{dom} \psi \wedge \text{TRUE} \\
\equiv\ &\text{“A1, A4 } \textit{and}\ S \subseteq X \wedge s \in X \Rightarrow S \cup \{s\} \subseteq X\text{”} \\
&\text{TRUE} \qquad \clubsuit
\end{aligned}$$

Combined these constitute an inductive proof that ensures that starting from the initial state it is not possible to enter an invalid state.

Testing with Haskell

Another approach to verifying specifications is to generate models, i.e., instances of the specified system, and check that they satisfy some property about the system. This only proves that the property holds for the models checked but it can still be a useful technique if enough models are included.

Irish VDM models can be checked this way by converting them to Haskell and using the QuickCheck library to generate models and verify properties. This is done using the `ivdm` library[2] which provides functions for creating Irish VDM models in Haskell. For example, $A \xrightarrow{m} B$ is as coded as `Map A B`, $A \subseteq B$ is `A 'subSet' B` and $x \in X$ is `x mOf X`.

The ATC system and its invariant could be written in Haskell as:

```
type Aircraft = Nat
type Controller = Nat
type ATCSys = (Set Controller,
               Set Aircraft,
               Map Aircraft Controller)

invATCSys :: ATCSys -> Bool
invATCSys (onduty, airborne, handlers)
   = airborne 'subSet' dom handlers
    && rng handlers 'subSet' onduty
```

Here the *Aircraft* and *Controller* domains have been represented as natural numbers, but the remainder is an exact duplicate of the *VDM*♣ model. The TakeOff operation and its precondition translate to:

```
takeOff :: Aircraft -> ATCSys -> ATCSys
takeOff (a) (onduty, airborne, handlers)
   = (onduty, airborne union iSet a, handlers)

preTakeOff :: Aircraft -> ATCSys -> Bool
preTakeOff (a) (onduty, airborne, handlers)
   = not (a mOf airborne) && a mOf dom handlers
```

To check that the TakeOff operation preserves the invariant we use QuickCheck to generate valid models and test that the following function returns true for all of them. If it fails QuickCheck will display the constructed model

[2] https://www.cs.tcd.ie/Andrew.Butterfield/IrishVDM/software/IrishVDM-in-Haskell/

that caused the failure as a counterexample to the property. Note that QuickCheck will only generate up to 100 models by default.

```
takeOffPrsInv a atc
   = invATCSys atc && preTakeOff a atc ==> invATCSys(takeOff a atc)
```

3.3 Alloy

Alloy is both a specification language and tool for analysing those specifications. Alloy the language is a relational modelling language based on first-order logic [12]. The Alloy Analyzer is a "model finder" which verifies properties of Alloy specifications by checking them against multiple generated models.

3.3.1 Alloy Specification Language

Alloy is explained here through the translation of the air traffic control system from the previous *VDM*♣ example. The system can be specified in Alloy as

```
sig Controller, Aircraft {}
sig ATCSys {
  onduty: set Controller,
  airborne: set Aircraft,
  handlers: Aircraft -> lone Controller
}
```

The domains of the system are represented by three *signatures*. As before, *Aircraft* and *Controller* are token types and are therefore blank, while the *ATCSys* signature contains three *relations*. The *onduty* field denotes a relation between ATCSys objects and zero or more aircraft, as indicated by the *set* multiplicity keyword. Similarly, *airborne* relates ATC systems to zero or more aircraft. *handlers* maps aircraft to their currently assigned controller; ψ from the *VDM*♣ model. Each aircraft is mapped to at most one controller as denoted by the *lone* keyword.

The invariant for the system can be written using the following predicate, *invATCSsys*. This takes an *ATCSys* and checks that *airborne* is a subset of the domain of *handlers* and that the range of *handlers* is a subset of *onduty*, i.e. every airborne aircraft has exactly one on duty controller.

```
pred invATCSys[s: ATCSys] {
  s.airborne in dom[s.handlers]
  ran[s.handlers] in s.onduty
}
```

The operation *takeOff* is written as a predicate which takes two *ATCSys* objects and an aircraft a. *ATCSys* s represents the system before the plane takes off, and s' represents the system afterwards. The operation states that the *onduty* and *handlers* relation is unchanged and that a is added to the set of airborne planes.

```
pred takeOff(s, s': ATCSys, a: Aircraft) {
  s'.onduty = s.onduty
  s'.airborne = s.airborne + a
  s'.handlers = s.handlers
}
```

The precondition of the operation specifies that a must have an assigned controller. Note that the *in* keyword works for both elements and sets.

```
pred preTakeOff(s: ATCSys, a: Aircraft) {
  a in dom[s.handlers]
}
```

In addition to predicates, Alloy supports functions. For example the function *handler* takes an aircraft and returns that aircraft's currently assigned controller. This function makes use of the *univ* relation that contains all other relations in the system.

```
fun handler(a: Aircraft) {
  univ.handlers[a]
}
```

3.3.2 The Alloy Analyzer

The Alloy Analyzer is a tool which can be used to check specifications written in Alloy. It operates by generating models, i.e. instantiations of the specification, and checking that they satisfy some given constraint. This is similar to the use of QuickCheck for Haskell implementations of Irish VDM models with the major difference that rather than checking around 100 randomly generated models, it checks *every* possible model within some scope.

For example, the analyser can be used to check constraint consistency, i.e. that at least one model can be generated that satisfies the constraint. The instruction "**run** invATCSys **for** 4" tells the analyzer to find a model that satisfies the state invariant predicate. It searches for a suitable model with a scope of four, meaning that if necessary it will generate every possible combination of relations with up to four instances of each signature. If it finds such a model it can present it graphically for the user.

The analyzer can also be used to check assertions by searching for counterexamples to them. For example, to verify that the *takeOff* operation preserves the invariant we can use the assertion

```
assert takeOffPrsCns {
  all s: ATCSys, a: Aircraft |
    invATCSys[s] && preTakeOff[s,a] && takeOff[s,s',a]
      implies invATCSys[s']
}
check TakeOffPrsCns for 7 but 2 ATCSys
```

which claims that if a valid system satisfies the precondition of *takeOff*, applying *takeOff* results in another valid system. The line of code beginning "**check**" instructs the analyzer to try find a model that violates the assertion. It does this using an exhaustive search space of up to seven Aircraft and Controller 'atoms", but only two air traffic control systems. If it finds such a counterexample it can display it graphically.

Thus the analyzer does not prove assertions, merely refutes them with counterexamples if they exist within a given scope. Daniel Jackson asserts that a small number of instances are sufficient to uncover "pathological" errors

and claims the *small scope hypothesis*: "Most bugs have small counterexamples" [14].

It is important to note that even with a small number of possible atoms, the search space required for models is very large. A specification with only one signature and one relation in a scope of 4 has 65536 possible models. Indeed a specification with r binary relations and scope s has a total of 2^{rs^2} possible models [13]. To manage this problem the analyzer converts the first order relational logic to and from boolean formulas which are then handled by a boolean satisfiability problem (SAT) solver.

3.4 JML

In the 1980's Bertrand Meyer introduced the concept of "Design by Contract" for object oriented programming. The idea is that each method of a class has a set of preconditions that callers must satisfy when calling it. In return the method has a set of postconditions that it guarantees to fulfil. Additionally a class may define invariants that all methods guarantee to maintain. The preconditions, postconditions and invariants define the *contract* that the class exposes.

Design by contract constructs were integrated into the programming language Eiffel that Meyer created. Eiffel influenced the design of Java but these idioms were not included. The Java Modelling Language (JML) solves this problem by introducing formal specification annotations to the language.

JML is introduced by implementing the air traffic control example.

```
public class ATCSystem {
  private final /*@ spec_public non_null @*/ Set airborne = new HashSet();
  private final /*@ spec_public non_null @*/ Set onduty = new HashSet();
  private final /*@ spec_public non_null @*/ Map handlers = new HashMap();
```

Here the class `ATCSystem` is defined, containing the three fields `airborne`, `onduty` and `handlers` as seen in the Alloy example. They are all annotated with `spec_public` which means that although the Java fields have been de-

clared `private`, the fields can be used in publicly visible specifications. They are also annotated with `non_null` which ensures that they may never be null.

State invariants are specified using the `invariant` annotation. These define class level constraints that must be true at all times.

```
//@ public invariant handlers.keySet().containsAll(airborne);
//@ public invariant onduty.containsAll(handlers.values());
```

The `TakeOff` method takes an Aircraft and adds it to the set of airborne aircraft.

```
//@ requires handlers.keySet().contains(aircraft);
//@ ensures airborne.contains(aircraft);
public void takeOff(/*@ non_null @*/ Aircraft aircraft) {
  airborne.add(aircraft);
}
```

The `requires` annotation denotes a precondition, a requirement that callers of this method must satisfy. Here it requires that the passed aircraft is a member of the domain of the handlers mapping. There is an additional `non_null` annotation on the aircraft argument that requires that it is not null; this is equivalent to adding `requires aircraft != null` to the method signature. The `ensures` annotation specifies a predicate that the method must fulfil once execution is complete. Here it ensures that the passed aircraft has been added to the set of airborne aircraft.

The preceding code shows how easily VDM style specifications can be added to Java programs. JML provides further features that can be shown by a different example. A simple `BankAccount` class is defined that contains a non-negative balance, i.e. this bank account can never be overdrawn

```
public class BankAccount {
  private /*@ spec_public @*/ double balance;
  //@ public invariant balance >= 0;
```

A `withdraw` method is then defined that deducts a given amount of money from the account

```
/*@ public normal_behavior
  @   requires amount > 0;
  @   requires amount <= balance;
```

```
  @   assignable balance;
  @   ensures balance == \old(balance) - amount;
  @ also
  @ public exceptional_behavior
  @   requires amount > balance;
  @   signals (BankException e) balance == \old(balance);
  @*/
public void withdraw(double amount) throws BankException {
  if (amount <= balance) {
    balance = balance - amount;
  } else {
    throw new BankException("Insufficient funds");
  }
}
```

The `normal_behavior` of the method defines how it should normally operate. Here the normal behaviour is defined such that if a positive amount of money is requested and the balance of the account is greater than or equal to the amount, then the balance is decremented by this amount. The `\old` keyword is used to refer to the value of fields *prior* to the execution of the code. The `assignable` annotation declares exactly those fields that the method alters; no other field may be modified.

The `exceptional_behavior` of a method defines circumstances where it will throw an exception. Here this will happen if the amount requested is greater than the balance of the account. The `signals` annotation specifies that a `BankException` will be thrown but that the balance will remain unchanged.

The bank account class also provides a retrieval function `getBalance` which returns the current balance of the account. This method is `pure` meaning that it has no side effects - it does not modify any values.

```
/*@ public normal_behavior
  @   ensures \result == balance;
  @*/
public /*@ pure @*/ double getBalance() {
  return balance;
}
```

3.4.1 ESC/Java2

The Extended Static Checker for Java 2 (ESC/Java2) is be used to automatically verify JML specified programs. It translates the code and specifications into *verification conditions* which are then verified using *Simplify*, a first-order logic theorem prover [6]. If ESC/Java2 finds an error in the code, for example an ensures postcondition not being met by a piece of code, it can highlight the line of code in issue and present a counterexample.

Part II

CianVotáil

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."

C. A. R. Hoare

4 Design

This chapter explains the design of *CianVotáil*, the remote electronic voting system developed for this project. The main component of this system is a server application that collects and stores ballots and maintains a list of eligible voters. This *Ballot Collector* server attempts to satisfy the appropriate election requirements (§2.1) by using asymmetric cryptography (§2.4.1) to encrypt and sign ballots which are subsequently published (§2.3.1) to enable individual voter verification.

A formal specification for the Ballot Collector software is presented in this chapter using Irish VDM (§3.2), Alloy (§3.3) and JML (§3.4).

4.1 Overview

Before the election, authorities generate a list of eligible voters and define the terms of the election. This information is used to configure a *Ballot Collector* (BC) server. The election authorities also generate an asymmetric key pair and provide the collector with the public key for use when encrypting ballots. The private key is kept secret by the administration.

When polling opens, voters send ballots along with their identification to the collector using a *Voter Agent*[1]. When the collector receives a ballot, it checks

[1]Adapted from the term "user agent" used to describe standard network client appli-

that the identification provided matches the credentials of an eligible voter; marks the voter as voted; encrypts and signs the ballot; stores the encrypted and signed ballot; and returns a copy of the encrypted and signed ballot to the voter as a receipt.

When polling closes, the ballot collector makes public all the encrypted ballots. Voters may check that the encrypted ballot receipt they received when voting is contained in the published set of ballots. The list of voters is also published so that abstaining voters can check that no vote has been cast illegally on their behalf. The only information published about voters is their potentially anonymous username and a boolean flag stating whether or not they voted, thus preserving privacy.

The election authorities use their private key to decrypt the set of encrypetd ballots and publish them in their plaintext form. A *Tallier* application then counts the published plaintext ballots and the election result is declared.

4.1.1 Software modules

There are three software modules in the system: the Voter Agent, the Ballot Collector (BC) and the Tallier.

The Voter Agent is merely an intermediary interface between voters and the Ballot Collector and requires only publicly available information to operate. There may therefore be many Voter Agent implementations offered by various third parties for different purposes, e.g. an accesibile interface for the visually impaired or an application for mobile phones. It is vitally important for the voter, however, that the agent they use operates correctly and the election authorities may therefore officially endorse only particular implementations.

There will necessarily be one authoritative Tallier, i.e. the software used by the election authority to declare the results, but since it uses only publicly available information, anyone can create their own Tallier to count the votes

cations like web browsers.

and confirm the result.

The critical entity is the Ballot Collector (BC). There is only one BC that all voters must communicate with. It must operate correctly in order to maintain the election requirements, e.g. only eligible voters may vote and they may only vote once. This is the part of the system that this project primarily focuses on and the rest of this chapter is dedicated to its design.

4.2 Ballot Collector Specification

The Ballot Collector has a number of states where some operations are allowed and others are prevented. They are inspired by similar states of the KOA system (§2.5) though there are fewer here since the the BC does not deal with decrypting or tallying ballots as KOA does.

1. **Prepared.** When first initialized, the BC requires a list of eligible voters, the election definition and a public key to encrypt ballots with. It immediately enters the *prepared* state. In this state voters can check that they are registered to vote and view the election definition but cannot vote.

2. **Open.** To start polling, an administrator *opens* the BC. In this state it will accept ballots but not allow anyone to view cast ballots.

3. **Suspended.** During polling, an administrator can halt the BC by moving it to the *suspended* state. Here the BC will not accept any new ballots while still denying access to cast ballots. In order to resume polling, an administrator must move the BC back to the *open* state.

4. **Blocked.** The BC will automatically enter the *blocked* state if it determines that an invariant has somehow been violated (e.g. there are somehow more stored ballots that voters who have voted). As with the suspended state, no ballots may be cast and an administrator must re-open the BC to resume polling.

5. **Closed.** When *closed* the BC will not accept any further ballots and will finally allow access to the cast ballots. The BC is now in an immutable state and cannot be re-opened.

4.2.1 Irish VDM

The Irish VDM model only specifies the behaviour of the BC in the *open* state. This is mainly for simplicity, but the open state is also the most interesting and important since it is the only one where the contents of the system can be changed.

The only domains of the system are *Voter* and *Ballot*. Election details such as the set of running candidates are ommitted as they are secondary to the operation of the Ballot Collector. A constraint is made on ballots than that they constitue a valid vote for the current election (e.g. an STV ballot containing only running candidates in a unique sequence), but the details of this validity are left to implementation. Each ballot contains a unique ID string.

The system state, *BC* (Ballot Collector), may be defined by three sets: the eligble voters $\mathcal{E}$; the cast ballots $\mathcal{B}$; and the voters who have voted $\mathcal{V}$.

$$(\mathcal{E}, \mathcal{B}, \mathcal{V}) \in BC \quad \widehat{=} \quad \mathcal{P}\,Voter \times \mathcal{P}\,Ballot \times \mathcal{P}\,Voter$$

The system is subject to the constraint *inv*-BC which ensures that only eligible voters may vote and that the correct number of ballots have been cast (i.e. the total number of cast ballots is equal to the number of voters who have voted). This is a very weak invariant it it does not guarantee anything about the actual stored ballots other than that there is the correct number.

$$\begin{aligned} \textit{inv}\text{-BC}(\mathcal{E}, \mathcal{B}, \mathcal{V}) \mathrel{\widehat{=}} \quad & \mathcal{V} \subseteq \mathcal{E} \\ \wedge \quad & \#\mathcal{B} = \#\mathcal{V} \end{aligned}$$

The important operation is Vote which casts a Ballot b on behalf of Voter v and marks v as voted.

$$\text{Vote}[v, b](\mathcal{E}, \mathcal{B}, \mathcal{V}) \mathrel{\widehat{=}} (\mathcal{E}, \mathcal{B} \cup \{b\}, \mathcal{V} \cup v)$$

The precondition of Vote ensures that v is an eligible voter, that v has not yet voted that it b has not already been cast.

$$\begin{aligned} \textit{pre}\text{-Vote}[v, b](\mathcal{E}, \mathcal{C}, \mathcal{B}, \mathcal{V}) \mathrel{\widehat{=}} \quad & v \in \mathcal{E} \ \wedge \ v \notin \mathcal{V} \\ \wedge \ & b \notin \mathcal{B} \end{aligned}$$

4.2.2 Alloy

The Irish VDM specification is very simple and a direct mapping to Alloy produced no counterexamples. A second Alloy specification was made that included the concept of the ballot reciept by using a relation between Voters and Ballots. This relation allows for the modelling of the voter verifiability concept.

The specification requires a *fact* block that ensures that no two voters have the same receipt

```
sig Ballot {}
sig Voter {
  receipt: lone Ballot
}

fact oneReceiptPerVoter {
  all v, v': Voter |
    not v = v'
      implies not v.receipt = v'.receipt
}

sig BC {
  voters: set Voter,
  ballots: set Ballot,
  voted: set Voter
}
```

The receipt relation allows a more accurate state invariant than was used in

the Irish VDM specification. This invariant ensures that only eligible voters may vote; that if a voter has voted they have a receipt; that the voter's receipt is contained in the set of cast ballots; and that every cast ballot is the receipt of some voted voter.

```
pred invBC[bc: BC] {
  all v: bc.voted |
    v in bc.voters and
    some v.receipt and
    v.receipt in bc.ballots

  all b: bc.ballots |
    some v: bc.voted |
      v.receipt = b
}
```

This strengthened invariant enables assertions to be made about the system. For example, we can assert that if the invariant is preserved, then correct number of ballots have been cast.

```
assert correctNumberOfBallots {
  all bc: BC | invBC[bc] implies #bc.ballots = #bc.voted
}
```

This Alloy specification is obviously not implementable in a real system (the link between voter and ballot would violate privacy) but it can be used to reason about the actual distributed implementation of the Ballot Collector. For example, we can determine that if the number of ballots is *not* correct then the invariant has been violated.

4.2.3 JML Specification

The JML specification is completed exactly as in Irish VDM, with a set of eligible voters, a set of voted voters and a set of cast ballots.

```
private /*@ spec_public non_null @*/ final Set castBallots;
private /*@ spec_public non_null @*/ final Set eligbleVoters;
private /*@ spec_public non_null @*/ final Set votedVoters;
```

The JML `constraint` annotation is used to specify the properties that must hold while the collector is in a certain state. For example, the following two

constraints specify that the set of eligible voters is immutable and that when the ballot collector is suspended, no new ballots may be cast

```
//@ constraint \old(eligbleVoters).equals(eligbleVoters);
//@ constraint state == State.SUSPENDED ==>
//@          \old(castBallots).equals(castBallots);
```

Again the interesting method is `vote(Voter, Ballot)`, which requires that the voter is eligble, that they haven't already voted, and that the ballot hasn't already been cast. Its postcondition ensures that the ballot has been cast and that the voter is marked as voted.

```
/*@ ...
  @  requires state == State.OPEN;
  @  requires eligbleVoters.contains(voter);
  @  requires !votedVoters.contains(voter);
  @  requires !castBallots.contains(ballot);
  @  requires isValid(ballot);
  @  ensures castBallots.contains(ballot);
  @  ensures votedVoters.contains(voter);
  @*/
public void vote(Voter voter, Ballot ballot) {
    castBallots.add(ballot);
    votedVoters.add(voter);
}
```

4.3 Ballot Collector Server

The Ballot Collector is exposed as a *service* through a web server. It provides an application programming interface (API) for both Voter Agents and election administrators. This section describes the Ballot Collector API and introduces the REST architectural style which was used in its design.

4.3.1 REST

Representational State Tranfer (REST) is an architectural style for "distributed hypermedia systems" (i.e., Web applications). It emerged from the

design decisions made during the drafting of the HTTP specification. It was introduced by one of HTTP's principal authors, Roy Fielding [7].

REST systems consist of a set of *resources*, which can be thought of as objects from object orientated languages. Each resource can be referred to by its unique *resource identifier*. Resources are transferred between the client and server as *representations* of the current state of the resource in some media type. The resources expose a *uniform interface* for the transfer of state representations, i.e. there is a constrained set of operations that have semantics common to all resources.

REST using HTTP

HTTP is usually the protocol used for REST systems, where the resource identifier is a Uniform Resource Identifier (URI), the representations are HTTP messages, typically containing HTML or XML media types, and the uniform interface is the set of HTTP methods GET, POST, PUT and DELETE.

HTTP requests are sent from clients to servers and may include a message body containing a resource representation. GET returns a representation of a resource from a URI. PUT sends a representation to a URI to either replace that URI's existing resource or create a new resource there. POST sends a representation of a new resource to be added to a collection. DELETE, unsurprisingly, deletes the resource at the requested URI.

The server responds with a HTTP response, which may or may not contain a resource representation, but will contain a status code. The status code indicates how the server dealt with the request. For example, 200 OK indicates the request was fulfiled successfully while 401 Unauthorized indicates that authentication details are required to complete the request.

Benefits

REST systems are stateless, meaning that each request contains all the information required to fulfil it. There is no concept of server side session data maintained between calls providing further context to the request. This can improve performance and scalability through load balancing since any request can be handled by any server.

Unique URIs for each resource provide the ability for efficient caching. Clients can perform conditional GETs, which only return the representation of a resource if it has changed since the last time the client requested it.

For this project, the main advantage of REST over other architectures is simplicity. For example, by using HTTP as the underlying platform, the well supported HTTP basic and digest authentication schemes can be used.

4.3.2 Ballot Collector API

Table 4.1 outlines the API that the Ballot Collector presents to Voter Agents. The *URI Template* and *Method* columns show the types of requests that the server will accept[2]. The *Code* and *Response/Cause* columns show how the server may respond to these requests. For successful responses (200 and 201) the contents of the response body is detailed. For error codes (400+) the cause of the failure is explained.

The remainder of this section explains the use of the API.

Voting. Voters cast their ballots by making a POST request to `/ballots/`. The request contains a valid ballot, encoded as XML, and includes an Authorization header containing their username and password. If the ballot is cast successfully, the server returns a 201 Created response. This response contains an XML document specifying the encrypted value of the ballot, a

[2]The URI template is relative to be base URI of the server. For example, if the Ballot Collector server was located the URI `http://example.net/bc`, the URI Template `/ballots/` denotes the absolute URI `http://example.net/bc/ballots/`

URI Template	Method	Code	Response/Cause
`/ballots/`	GET	200	All the encrypted ballots
		403	Collector not closed
	POST	201	Newly created encrypted ballot
		403	Collector not open
		401	Not authenticated as an eligible voter
		400	Invalid ballot
		403	Voter has already voted
`/ballots/{id}`	GET	200	The encrypted ballot with ID equal to {id}
		400	Collector not closed
		404	No such ballot
`/state`	GET	200	Current state of the collector
	PUT	200	State changed
		401	Not authenticated as an administrator
		400	Invalid state
		403	Invalid state transition
`/voters/`	GET	200	All eligible voters
		403	Collector not closed
`/voters/{user}`	GET	200	The voter with username {user}
		401	Not authenticated as {user} while collector not closed
		404	No such voter
`/election`	GET	200	Details of the election

Table 4.1: REST Ballot Collector API

signature for the encrypted data and a reciept ID. The response also includes a Location header that specifies the URI of the created ballot. An example of such a request and response pair is included in §C.1.

The request may fail for several reasons. A 403 Forbidden response will be generated if the Ballot Collector is not in the *open* state or if the voter has already voted. 401 Unauthorized means that the valid credentials for an eligible voter have not been provided. 400 Bad Request will be returned if the ballot sent does not match the expected XML schema, or if it is not valid

by the terms of the election.

Verification. Once polling ends and the BC enters the closed state, the encrypted ballots and list of voters are made public. Voters can check that their ballot is included correctly by making a GET request to `/ballots/{id}`, where `{id}` is their receipt ID. Abstaining voters can check that no one has voted on their behalf by checking their details at `/voters/{username}`.

A complete list of encrypted ballots and voters is available from `/ballot/` and `/voters/` respectively and may be used to check that the number of cast ballots is equal to the number of people who voted.

Administration. Election administrators can change the state of the election by sending a PUT request containing the new state to `/state/`. This will fail with a 401 Unauthorized if the administrator does not provide the appropriate username and password, 400 Bad Request if the required state is not a valid state (i.e. not one of OPEN, SUSPENDED, CLOSED), or 403 Forbidden if the current state of the system does not allow moving into the new state.

"The function of good software is to make the complex appear to be simple."

Grady Booch

5 Implementation

This chapter explains how the CianVotáil system was implemented from the design outlined in Chapter 4. A major goal of this project was to create an implementation that was as simple and small as possible. To this end, Java 5 was chosen as the implementation language instead of Java 1.4. This more recent version of Java enabled the use of libraries that greatly reduced the amount of "boilerplate" code required for object persistence, XML binding and web services.

Unfortunately JML and ESC/Java2 are not yet compatible with Java 5 and so could not be used in the final implementation. The workaround used for this problem was to convert JML preconditions into Java assert statements that would ideally provide the same effect at runtime. Appendix A explains in detail the rationale behind the language choice and describes the libraries used.

5.1 Data classes

Many of the classes in the system are simple immutable data holding classes, annotated to facilitate database storage and XML serialization.

- `Election` - the title of the election and the set of running candidates

- `Candidate` - the ID, name and party of a candidate
- `Voter` - the username and password of a voter, and a mutable boolean `hasVoted` field
- `Ballot` - an array of candidate IDs
- `EncryptedBallot` - an ID, encrypted array of candidate IDs and a signature of the encrypted data

5.2 DBBallotCollector

`DBBallotCollector` is the main class in the system. It is an implementation of `BallotCollector` that stores data in a relational database. The following is the `vote` method from the class. As previously stated, assertions are used in place of JML preconditions. The Ballot is encypted and signed, stored in the database and returned to the user.

```
public EncryptedBallot vote(Voter voter, Ballot ballot) {
    assert state.equals(OPEN);
    assert isEligble(voter);
    assert !hasVoted(voter);
    assert isValid(ballot);

    EncryptedBallot encryptedBallot = encrypt(ballot);
    database.store(encryptedBallot);

    Voter dbVoter = database.findById(Voter.class, voter.getUsername());
    dbVoter.setVoted(true);
    database.update(dbVoter);

    return encryptedBallot;
}
```

5.3 Web Service

5.3.1 BallotCollectorResource

The `BallotCollectorResource` class allows remote access to a `BallotCollector` as outlined in §4.2.3. It is implemented using Jersey, the reference implementation of the Java API for RESTful Web Services (JAX-RS) (§A.3).

Normal Java methods are annotated to specify what HTTP method and URL they respond to and the content-types that they accept and return. JAX-RS also deals with marshalling and unmarshalling between Java objects and the specified content types.

The `BallotCollectorResource` class is annotated with `@Path("/election/")` to indicate it reponds to URLs beginning with `/election/`

The `vote(Ballot)` method responds to POST requests to the URL `/election/ballots/`, requires a request body containing the XML representation of a `Ballot` and responds with an XML document (the default content type) containing an `EncryptedBallot`. If the ballot is not valid or the collector is not open, the server responds with 400 Bad Request. It requires authentication and will respond with 401 Unauthorized if the voter does not send valid credentials. It will respond with 403 Forbidden if the voter has already voted.

```
@POST
@Path("/ballots/")
@ConsumeMime("application/xml")
public EncryptedBallot vote(Ballot ballot) {
    requireState(OPEN); // throws 400 if not
    if (!collector.isValid(ballot)) {
        throw new WebApplicationException(400);
    }
    Voter voter = requireAuthenticatedVoter();
    if (collector.hasVoted(voter)) {
        throw new WebApplicationException(403);
    }
    return collector.vote(voter, ballot);
}
```

The `getBallots()` method responds to exactly the same URL as `vote` (`/election/ballot/`), but to GET requests instead of POST requests.

```
@GET
@Path("/ballots/")
public Set<Ballot> getBallots() { ... }
```

The `getVoter(Voter)` method responds to the URL `/election/voters/{name}` where `{name}` is the username of an eligble voter. It returns an XML document containing the status of the requested voter, or 404 Not Found if the voter does not exist (i.e. is not eligible).

```
@GET
@Path("/voters/{name}")
@ProduceMime
public Voter getVoter(@PathParam("name") String username) { ... }
```

Web server

In line with the goal of simplicity, the system can work with the embedded HTTP server bundled with Java 6. It can also work within servlet containers such as Jetty and Tomcat and in application servers such as JBoss and Glassfish. Using an application server provides additional features such as stability and scalability.

Basic access authentication

The HTTP basic access authentication method [8] is used to authenticate voters and administrators. This method has the major weakness that the username and password are transmitted in plaintext, but since we are assuming a secure connection exists between the voter and the server this can be ignored. Digest access authentication does not have this problem.

Servlet containers and application servers provide HTTP authentication support but they require an amount of configuration to allow them to access the database. Basic authentication has therefore been implemented indepen-

dently in the system in the form of a utility class called `BasicAuthentication`.

5.4 Database

The system uses the Java Persistence API (JPA) to store objects transparently in a relational database. JPA is explained in more detail in §A.1.

The backend Relation Database Management System (RDBMS) used is HSQLDB, a lightweight (< 1Mb) RDBMS written in Java. It is trivial, however, to change the RDBMS to any that supports a JDBC (Java Database Connectivity) driver, e.g. MySQL or PostgreSQL.

It was found that the majority of actions in the system involved at most one database request, but even the simplest database access required a reasonable amount of JPA boilerplate code. The `Database` class was created as a facade to the JPA implementation that abstracted some of this complexity. For example, `Database.store(Object)` allows an object to be persisted in one line of code instead of 15 lines of JPA code (it would not be wise to use this function to store multiple objects since it creates a new transaction each time).

5.5 Configuration

When the application starts, the system checks for an existing database. If a database is found, it is loaded and the `DBBallotCollector` is resumed. If the database is not present, the system attempts to load the configuration files and uses them to create the initial database.

The election authority the Ballot Collector with following configuration files

- `voters.xml` contains a list of voters with their usernames and hashed

passwords

- `administrators.xml` contains a list of authorized election officials with their usernames and hashed passwords
- `election.xml` contains the description of the election - the title of the election and the list of running candidates
- `encryptionkey` contains the RSA public key that the Ballot Collector should use to encrypt the ballots (the authority keeps the private key)

The `*.xml` files contain XML that directly map to objects in the system. They are parsed using JAXB (§A.2). `encryptionkey` contains the Base64 encoded binary data representation of the RSA public key.

Examples of the XML files can be found in Appendix C.

6

Evaluation

This chapter provides a critical evaluation of CianVotáil, discussing how well the system succeeded in fulfilling the goals of the project. Where problems still exist, possible solutions are presented as future work.

6.1 Election requirements

The system developed largely satisfies the outlined election requirements (§2.1). For *individual verification*, voters can check that their encrypted ballot receipt is included in the list of published ballots, while for *universal verification*, anyone can tally the published unencrypted ballots and confirm the election result. The requirement for a *secret ballot* is fulfilled through the use of a secure connection (HTTPS) and encryption of the ballot. The *democracy* requirement that only eligible voters may vote and do so only once is sucessfully maintained by the specification of the ballot collector software.

One of the main remaining problems is the need for trust in the election systems. The voter must trust the ballot collector to both encrypt the ballot correctly and to discard their plaintext ballot. They must also trust the election authority to correctly decrypt the ballots.

An additional problem is that of coercion. All remote voting systems are susceptible to coercion since a coercer could monitor the voter while they

submit their ballot, or simply demand the voter's identification credentials[1]. Unfortunately CianVotáil makes this problem worse by publishing STV ballots which can lead to coercion through identifiable ballots as outlined in §2.3.1.

Both the problems of trust and coercion could be fixed by using homomorphic encryption as explained in §2.4.4. With this method, the election result can be tallied using only the encrypted ballots, removing both the possibility of coercion and the need to trust the election authority. This could also allow voters to encrypt their ballot with the election's public key *before* submitting it, removing the need for trust in the ballot collector.

6.2 Formal Verification

The core of the system, the Ballot Collector, has been formally specified using Irish VDM, Alloy and JML. Unfortunately, JML does not currently support Java 5, meaning that the actual implementation could not make use of it. The workaround chosen for this problem was to convert the JML preconditions into Java assert statements. As explained in §A.1, this is not an ideal solution since it meant the implementation could not be formally verified using ESC/Java2.

Future work to verify the core should be reasonably straightforward given the existing specification and the small amount of code involved. Ideally, however, to be confident of the system's behaviour, the libraries that the implementation makes use of (see Appendix A) should also be verified. This would involve a considerable amount of work.

[1] A possible solution to this was used in a the 2007 Estonian parliamentary election. Voters could cast multiple ballots online, with only the most recent ballot being included in the count. They could also override their online vote by voting in person on election day.

6.3 Simplicity

The implementation exceeded expectations for its simplicity. As shown in table 6.1, CianVotáil contains over 37 times less non-commenting source statements (NCSS) than KOA.

	CianVotáil	KOA
Classes	22	489
Methods	117	2693
NCSS	778	29053

Table 6.1: Code Metrics

6.4 Future Work

As already mentioned, the actual implementation needs to be formally verified and the system could be improved by being adapted to use homomorphic encryption. There are further areas where the system could be enhanced and some are presented here as possiblities for future work.

6.4.1 Tamper Proof Storage

The specification of the BallotCollector (§4.2) includes two mutable collections: the voters who have voted and the cast ballots. These are represented using the *set* abstract data type, but they actually have some additional requirements beyond those of a normal set.

Once a ballot or voter is added to these sets it should be impossible to remove or edit them; they should be *write once*. If an element *is* removed or edited, the violation should be easily discoverable; they should be *tamper evident*. Since ballots and voters are added to the sets simultaneously, any latent

ordering of elements would provide a link between voters and their ballots; they should be *history independent* [20].

CianVotáil implements these data structures using a relational database that does not provide these properties. The write-once property is maintained through the specification of the program but a malicious program installed on the server by a corrupt election official could alter the contents of the database. The tamper-evident property is not maintained directly, but through the use of encryption and voter verification. When the collections are accessed, they are sorted by username or a randomly assigned ID string, but the actual database implementation may store some history through the order of transactions.

The development of a write-once, tamper-evident, history-independent data store for use with election servers is therefore an interesting area for research. This has been previously pursued by [19] which presents a history independent data stucture and by [20] which presents a PROM based data store with all these properties.

6.4.2 Alternative Voter Agents

The voter agent implemented for this project is adequate for testing the system but would not be appropriate for an actual election. Developing alternative voter agents is made possible by the modular design of CianVotáil and the existence of the Ballot Collector API (§4.3.2) and would be an interesting area for future work.

One possibility is a voter agent designed to assist voters with poor eyesight. Another would be a website based agent for use on small screens such as mobile phones. Perhaps most usefully, a formally verified voter agent client library could be developed using JML and ESC/Java2.

6.4.3 Tallier

Integration with the formally verified STV tallier *Votáil* (§2.5.3) was an intended goal of this project but was not completed due to time constraints. Completing this would simply require conversion between the XML ballot format produced by this system and the format expected by *Votáil*. If the system was converted to use homomorphic encryption a new tallier would need to be written or *Votáil* adapted.

6.4.4 Logging

While it is important not to store any information which could link a voter to their ballot, some logging is essential for auditing and maintaining security. The KOA system, for example, records login attempts and blocks access to a user after a certain number of failed attempts. This helps prevent an attacker from guessing a voter's password. The implementation in this project has minimal logging of this kind and could be improved with more detailed audit logs.

7

Conclusion

> "We have to correct the software, which will cost 500,000 and try to move forward. Otherwise, this country will move into the 21st century being a laughing stock with our stupid old pencils."
>
> Taoiseach Bertie Ahern

This quote comes from the same Dáil Éireann debate mentioned in the introduction. It hints that the reasons for the adoption of electronic voting systems are largely political rather than technical. Unfortunately this would seem to imply that they will be become increasingly common despite their many unsolved problems and against the recommendations of many computer science experts. The need for further research in the field is evident.

This project has developed CianVotáil, a formally specified remote electronic voting system that provides independent verification. Its remarkable simplicity and modular design mean that it could provide an ideal basis for such future work.

The core of the CianVotáil has been formally specified, but formal verification of the implementation was unfortunately not possible. This project shows how starting with a formal specification can remove the danger of *accidental complexity* in software and produce simple and clear implementations.

Bibliography

[1] Dáil Éireann, Leaders Questions. October 2006.

[2] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[3] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.

[4] Dermot Cochran. Internet Voting in Ireland Using the Open Source Kiezen op Afstand (KOA) Remote Voting System. Master's thesis, University College Dublin, March 2006.

[5] L. Cranor and R. Cytron. Sensus: A security-conscious electronic polling system for the internet, 1997.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Richard N. Taylor.

[8] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, Internet Engineering Task Force, June 1999.

[9] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *ASIACRYPT '92: Proceed-*

ings of the Workshop on the Theory and Application of Cryptographic Techniques, 1992.

[10] Rop Gonggrijp and Willem-Jan Hengeveld. Studying the Nedap/Groenendaal ES3B voting computer: a computer security perspective. In *USENIX/Accurate Electronic Voting Technology Workshop*, 2007.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[12] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM.

[13] Daniel Jackson. Alloy models and visualizations from demo. In *Keynote, 3rd International Conference of B and Z Users (ZB2003), Turku, Finland*, June 2003.

[14] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The koa remote voting system: A summary of work to date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2006.

[16] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. *sp*, 00:27, 2004.

[17] Mac an Airchinnigh, M. *Conceptual Models and Computing*. PhD thesis, University of Dublin, Trinity College, 1990.

[18] Michael McMahon. Verification of preferential voting system elections without publishing plain-text ballots. 2008.

[19] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine (extended abstract). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 365–370, Washington, DC, USA, 2006. IEEE Computer Society.

[20] M. Naor and V. Teague. Anti-persistence: History independent data structures, 2001.

[21] Commission on Electronic Voting. Interim resport on the secrecy, accuracy and testing of the chosen electronic voting system.

[22] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.

[23] Vanessa Teague, Kim Ramchen, and Lee Naish. Coercion-resistant tallying for stv voting. 2008.

[24] Patrick Tierney. Implementing the irish voting system, May 2007. Final Year Undergraduate Project Thesis.

A

Tools

One of the main aims of this project was that the implementation of Cian-Votáil should be as simple as possible. This chapter decribes the tools and libraries used to develop the system and explains how they help attain this goal.

A.1 Java 5

Java 5 was chosen as the implementation language instead of the older Java 1.4. Unfortunately at the time of writing JML and ESC/Java2 (§3.4) are still in the process of being re-written to handle the new Java 5 syntax meaning that the final implementation could not be formally specified and verified.

This section outlines some of the language features that made the choice of Java 5 compelling even at the expense of formal verification. It also presents a workaround for the lack of JML annotations.

Generics

Java generics are a useful, if slightly verbose, solution to many problems. The most apparent benefit is when using *collections*. The Java Collections Framework provides interfaces for common abstract data types like List, Set

and Map. Prior to Java 5 these were not *typesafe*, meaning objects of any type could be put into them. In addition, all their retrieval functions returned only the base type Object, meaning typecasts were required whenever they were used. For example, this is a Java 1.4 method that takes a set of Employee objects and returns the average of their salaries

```
int averageSalary(Set employees) {
    int total = 0;
    Iterator i = people.iterator();
    while (i.hashNext()) {
        Employee employee = (Employee) employees.next();
        total += employee.getSalary());
    }
    return total / employees.size();
}
```

This code is not typesafe, i.e. there is no guarantee that `employees` actually contains only Employee objects and if it does contain any incompatable types, the required typecast will cause the program to fail.

A possible solution is provided with JML. The JML specification of the Java collection types include a ghost field `elementType`. Ghost fields exist only with the specification and are not realised by any implementing code. Every method which adds something to the collection has a precondition that ensures that the type of the object being inserted is compatible with the `elementType` of the collection. This prevents mixing types within collections but does not remove the need for the typecast.

Java 5 generics provide a way to specify that a collection is of a particular type. The follow code shows the generic version of averageSalary and also uses the new *for each* loop construct

```
int averageSalary(Set<Employee> employees) {
    int total = 0;
    for (Employee employee : employees) {
      total += employee.getSalary();
    }
    return total / employees.size();
}
```

Enumerations

Enumerated types are often used when a field may be equal to one of a defined set of distinct values. For example, if we were to model a playing card there would likely be a field called "suit" which may be one of Club, Diamond, Heart or Spade.

Prior to Java 5, these were typically implemented by using a primitive type, such as integer, and the field would be set to one of a set of constants. For example the first few lines of a playing card class could be:

```
public class Card {
  public static final int HEARTS = 0;
  public static final int DIAMONDS = 1;
  public static final int CLUBS = 2;
  public static final int SPADES = 3;

  private int suit;
```

The problem with this is that `suit` is not typesafe. There is nothing stopping `suit` from being set to 8, -42 or 34525, none of which are valid suits.

Joshua Bloch, the designer of the Java Collections Framework, introduced the typesafe enum pattern to help tackle this [2]. The idea is to create a new class for the enum type with a private constructor so that the values can only be declared within that class. Unfortunately this becomes quite verbose and is rarely implemented correctly.

```
public class Suit {
    private final String name;
    private Suit(String name) { this.name = name; }

    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
```

Again JML can also be used to solve the problem. An invariant can be introduced to ensure that `suit` is one of the declared constants, but this too is very verbose - to be done correctly requires additional invariants to ensure no two of the declared constants are equal.

Java 5 introduces a new language construct for enumerated types, allowing the original class to be written much more succinctly as

```
public class Card {
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private Suit suit;
```

Annotations

Annotations are a way of adding metadata to code, i.e. information about code explaining what it is or how it should be used. For example, in previous versions of the testing framework JUnit, test classes were required to subclass a base class and the methods had to start with the word "test"

```
public class StringTest extends TestCase {
     public void testSubstring() {
```

Java only permits single inheritance so this meant that test classes could never form a type hierarchy. The most recent version of JUnit uses annotations and only requires that test methods be annotated

```
public class StringTest {
    @Test public void substring() {
```

Annotations simplify code by removing the need for excessive configuration. They are used extensively in the database, XML and web service tools used in this project.

Preconditions as assertions

The use of Java 5 meant sacrificing JML specification and ESC/Java verification from the final implementation. As a workaround the preconditions were replaced with assertions in the body of the method. An assertion is a guard that throws an exception if a given boolean expression is false. The idea behind this is that the correct behaviour should be preserved - the body

of the method remains as before and is only executed if the precondition holds.

For example, a method specified as

```
/*@ public normal_behavior
  @   requires amount > 0;
  @   requires amount <= balance;
  @   assignable balance;
  @   ensures balance == \old(balance) - amount;
  @*/
public void withdraw(int amount) {
    balance = balance - amount;
}
```

would be replaced with the code

```
public void withdraw(int amount) {
    assert amount > 0;
    assert amount <= balance;

    balance = balance - amount;
}
```

This is by no means an ideal solution - there is no way to perform static checking on this code, for example.

A.2 Java Persistence API (JPA)

The Java Persistence API (JPA) was used for the database backend to the system. JPA is an Object Relational Mapping (ORM) style database API which allows Java objects to be stored directly in relational databases.

Entities

JPA handles the conversion between objects and the relational database tables, using annotations as the guide for the mapping. Very little is required to make a class "persistable" - they must be annotated with "@Entity", they

must contain an ID field annotated with "@Id", and they must contain a non-private no-argument constructor. For example:

```
@Entity
public class Candidate {
    @Id private byte id;
    private String name;
    private String party;

    protected Voter() { }

    public Voter(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public byte getID(){ return id; }
    public String getName() { return name; }
    public String getParty() { return party; }
}
```

This results in a database table called CANDIDATE with the three columns ID, NAME and PARTY where ID is the primary key. The default settings used by JPA are usually acceptable, but they can be overriden to allow interoperability with existing databases. For example an alternate column name can be specified using `@Column(name="whatever")` and fields can be excluded from the database by annotating them with `@Transient`.

JPA Query API

The JPA Query Language (JPAQL) can be used to retrieve elements from the database.

```
Query q = em.createQuery("select v from Voter where v.hasVoted == true");
List voters = q.getResultList();
```

A.3 Java Architecture for XML Binding (JAXB)

The Java Architecture for XML Binding (JAXB) does for XML what JPA does for relational databases. It provides a simple way to marshal objects into XML format and to unmarshal XML back into Java objects.

For example, the following class

```
@XmlRootElement
public class Candidate {
    @XmlAttribute private byte id;
    @XmlElement private String name;
    @XmlElement private String party;

    protected Voter() { }

    public Voter(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public byte getID(){ return id; }
    public String getName() { return name; }
    public String getParty() { return party; }
}
```

Will be translated into the following XML

```
<candidate id="4">
  <name>Thor</name>
  <party>Aesir</party>
</candidate>
```

A.4 Java API for RESTful Web Services (JAX-RS)

The Ballot Collector web serice was implemented using XML Rest Services (JAX-RS) which is defined by JSR-311 and implemented by Jersey. Service

methods are annotated with the URI template that they map to, the HTTP method they respond to and the content types that they produce. JAX-RS is still being specified and as a result has a constantly changing API and a occasionally buggy implementation.

```
@Path("/election/")
public class BallotCollectorResource {
    ...

    @POST
    @Path("/ballots/")
    @ConsumeMime("application/xml")
    @ProduceMime("application/xml")
    public EncryptedBallot vote(Ballot ballot) {
    ...

    @GET
    @Path("/ballots/")
    @ProduceMime("application/xml")
    public Set<Ballot> getBallots() {
    ...
}
```

B

Voter Agent

The Voter Agent implemented for CianVotáil is a simple website built using HTML and javascript. This combination was chosen to allow the web browser to handle the majority of the required grunt work such as dealing with HTTPS connections and authentication. It has the added advantage that the code is small and instantly accessible through the web browser to allow voters to verify its behavior.

Although it exists on the same web server as the Ballot Collector, it does not derive any special benefit from this and uses the same Ballot Collector API that any voter agent could.

C

Sample output

C.1 Ballot Collector server

This is an example of a voter casting a ballot as described by the REST Ballot Collector API (§4.3.2). First the request from the voter:

```
POST /ballots/ HTTP/1.1
Host: example.org
Content-Type: application/xml; charset=UTF-8
Authorization: Basic anVsaWU6anVsaWU=

<ballot>
  <choice id="5" />
  <choice id="8" />
  <choice id="3" />
  <choice id="11" />
</ballot>
```

Reply from server:

```
HTTP/1.1 201 CREATED
Content-Type: application/xml; charset=UTF-8
Location: http://example.org/ballots/wtk9ms

<encryptedBallot id="wtk9ms"
  data="at1qWETGrGmpPo4u2gKZJdIBn5MOgSRcVY1gr0l5S9B9IWmzpKnkOx3
hfrzG0XxdmSd3HXh/TD3CQdZVZad+NGIyz5iluEUzSqsGGwHwI2Qgf6CtlIZEUo
JhTh/4zvr2Jj4cvT9rudqmK2cfhPAbLN6pMwyz6HdO3bMuPh8BXHw="
  signature="UifZF1+uEEebv/tPYGeyX5Lf/sUD+2Xdr3aMqJuaWgOTxGLN6m
w//C23SHet7snMB7B3TzB9i6veOQkiBn7rGYGjZdPpW1DZUJYBVDFn/iVzGDE+x
dIBP7F1giDfmLjvpUkVRyK7Mdh7yD7WvknOYuwIUSmvhl6p2wvDJATc0yU="/>
```

C.2 Configuration files

These are some, very small, examples of the types of configuration files that are required to start a Ballot Collector, as explained in §5.5

voters.xml defines the set of eligible voters along with their credentials

```
<voters>
  <voter username="jneill" password="N/Ek7N7UtBqUP3g0+lLd+DWtCAw=" />
  <voter username="younggx" password="0UTdttfv66FwBdnFp3y8R2EuH/I=" />
  <voter username="dromeyl" password="licBxWuTn+mJLkHuYqMAovuegyM=" />
  <voter username="redmondbd" password="coOHOqzoUWu5KphCOdWR72n53Bs=" />
  <voter username="toalc" password="lb5PoS04WQ03jxqlYayMVH7AlNg=" />
</voters>
```

administrators.xml defines the set of authorized election officials who may alter the state of the system

```
<administrators>
  <admnistrator username="admin6" password="pgyo5Di/uxXte+aN9UqGPMzpvhE=" />
  <admnistrator username="admin5" password="ZbdWMjLua9tJyE9szQlQX6zjrAw=" />
  <admnistrator username="admin4" password="W83SvCdOETqrYp1s/z/tesQ/3DM=" />
  <admnistrator username="admin3" password="wLtXYX+EHFAwP7qCJoLFtvyBqH0=" />
  <admnistrator username="admin2" password="mTnmK1cy+QsCXE3Q7VP4ByNw0a0=" />
  <admnistrator username="admin1" password="zLgyieCpErI+VgFTiNQORC8PjTY=" />
</administrator>
```

election.xml defines the candidates running in an election and the election's title

```
<election>
  <title>Asgard Presidential Election 2008</title>
  <candidates>
    <candidate id="1" name="Odin" party="Aesir" />
    <candidate id="2" name="Tyr" party="Aesir" />
    <candidate id="3" name="Freyja" party="Vanir" />
    <candidate id="4" name="Thor" party="Aesir" />
    <candidate id="5" name="Loki" party="Aesir" />
    <candidate id="6" name="Frigg" party="Asynjur" />
    <candidate id="7" name="Idunn" party="Asynjur" />
  </candidates>
</election>
```