Verified and Verifiable Computation with STV Algorithms

Milad K. Ghale

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

October 2019

Except where otherwise indicated, this thesis is my own original work.

Milad K. Ghale
21 October 2019

To my lovely parents, Mehrnaz and Parviz, for having done for me whatever that was in their capacity

Acknowledgements

I am hugely grateful to my lovely parents, Mehrnaz and Parviz. They have done for me whatever that was in their capacity and I shall remember them all until my last breath.

This process could have hardly reached to the current point had Dirk Pattinson and Michael Norrish not been supervising me. I am truly thankful to Dirk for his insightful remarks, ethical and professional manner, cooperativeness and having his door open to me whenever I needed a direction and technical guidance. Also, I am thankful to Michael for offering prompt advice which I always kept before me as a compass for navigating through and also for his guidance on using HOL4.

Life is not great without good friends. I have a few best friends who I keep close to my heart. Two of my best friends, Mohammad Sangtarashan and Sanaz Sheikhi, were the best company in different stages of the PhD process. They were there for me whenever I needed someone who knew me well and with whom I could have a meaningful deep conversation that I always enjoy. The other best friend of mine, Vahid Ashrafian, helped me crucially when I was going through the process of coming to Australia. Also I am grateful to two kind-hearted friends of mine, Hajar Sadeghi and Firouzeh Khoshnoodiparast, who assisted and guided me through the admission process of ANU and upon my arrival to Canberra. Moreover, I thank all of my friends at ANU for the cheerful occasional fun we had together and the warmth of friendship in brutal Canberra winters.

I also am thankful to Ramana Kumar for his help with CakeML and answering all of my questions kindly and patiently. More generally, I am thankful to the CakeML community for replying to my emails or messages in their chatroom which was significantly helpful. Also, I like to thank Laurent Thery for suggesting me on how to overcome a challenging problem I was facing with Coq about two years ago. His help resulted in making the extracted executable vote counting program considerably faster.

Also, I would like to thank anonymous reviewers of my papers and the thesis. Some of their comments gave me insight into aspects of my work and how to communicate it more effectively with readers. I have been attentive to their reviews constantly while composing the final version of my thesis towards creating a more satisfactory end result.

I also like to express my gratitude to ANU, particularly research school of computer science (RSISE). Part of the reason that all of these amazing experiences happened to me during and immediately after PhD is because of the admission and the scholarship that RSISE provided me with.

Last but never the least, I am one of the most fortunate humans on earth for hav-

ing been a student of great mentors before and during my PhD. I am deeply thankful to every one of them for having taught me life lessons. I like to particularly acknowledge a few of those from whom I have learned profoundly; Rumi, Immanuel Kant, Ludwig Wittgenstein, Martin Heidegger, Jürgen Habermas, Thomas Khun, Ferdinand de Saussure, Max Weber, Michel Foucault, Aristotle, Ludwig von Mises, Harry S. Sulivan, Alfred Adler, Martin Seligman, Sigmund Freud, Eric Fromm, Abdulkarim Soroush, and Farhang Holakouie. Most of them have passed away long ago. However, the presence of all of them in my life through their work has been a source of "awakening from slumbers of dogmas". They have "shined on" me when I was in many moments of doubts not having the confidence which direction to go further along.

Milad Ketabi
October 2019, Sydney Australia

Abstract

Single Transferable Vote (STV) is a family of preferential voting systems, different instances of which are used in binding elections throughout the world. Most countries with an STV system rely on archaic manual vote counting or opaque unreliable computerised methods. Although the technology exists to enhance the situation by building significantly more transparent, trustworthy, reliable vote counting tools, in practice these technologies are ignored. We introduce a framework which formalises and verifies the similarities of STV algorithms as an abstract machine and realises differences of various STV algorithms as instantiations into the machine. The framework provides a uniform and modular process of (a) producing tools that carry out verified computation with an STV algorithm and (b) synthesising means for verifying the computation carried out independently of the computation's source code. It also provides flexibility and ease for adapting and extending it to a variety of STV schemes. We minimise the trusted base in the correctness of the tools synthesised by using the Coq and HOL4 theorem provers and the ecosystem of CakeML as the technical basis. Moreover, we automate almost all proofs that we establish in Coq, HOL4 and CakeML so that new instances of verified and verifying tools for computation with a variety of STV algorithms can be created with no (or minimal) extra verification. Finally, our experimental results with executable code demonstrate the feasibility of deploying the framework for verifying real size elections having an STV counting mechanism.

Contents

List of Figures

Chapter 1

Introduction

Most countries with a single transferable vote (STV) [53] system rely on archaic manual vote counting or opaque unreliable computerised methods. Although technology exists to enhance the situation by building significantly more transparent, trustworthy, reliable vote counting tools, in practice these technologies are ignored. This thesis introduces a modular framework for formalisation, verification and construction of provably correct, efficient, independently verifiable tools for counting votes with various STV algorithms. It builds on lessons learned from theoretical computer science, such as automata theory and programming semantics, together with formal engineering and verification methods.

STV is a family of preferential-proportional schemes embraced in several countries, such as Ireland, New Zealand and Australia. STV schemes are used for a range of significant elections including presidential, Senate, and parliamentary. The STV family has risen to such a prominent status because it has survived numerous political challenges [54]. As a result, it has grown more mature over the decades, being successfully adapted and assimilated within divergent political environments [54]. In spite of the maturity and importance of STV schemes, the current practice of vote counting methods with STV algorithms are next to an embarrassment.

History has witnessed tragic performances for conducting harmful methods in counting under STV. In 2013 the authorities in Western Australia (WA) informed the public that mistakes had happened throughout the manual process of counting in the Senate election [95]. They were ashamed that 1370 ballot papers had disappeared. The loss was so striking that it could change the outcome, even who was a winner. Consequently, the initially announced result was aborted. The public embarrassment led to repeating the election, financed by taxpayers, costing some 20 million Australian dollars.

The public vote and trust in the integrity of the electoral system is a cornerstone of modern democracy. One instance of such failures is enough to cast questions on the authenticity and trustworthiness of elections procedure and democratic institutions where they take place in. Mistakes like this subsequently disfigure the delicate face of trust in democracy both as a process and an ideal to desire. To safeguard this fragile trust, and also save resources, electoral authorities have come to rely on computerised vote counting as an alternative.

Adapting an electronic method for operating an election is progress. However,

it does not automatically eliminate every problem. Indeed, some standards such as integrity of the election, guaranteeing the correctness of the outcome, the privacy of both a voter and his/her cast vote, and transparency of process become more challenging as compared to manually operated elections. For example, in a traditionally held election part of the process, such as the physical ballot box is tangibly observable and also a voter can readily know whether or not their vote is included among the cast votes. In the electronic counterpart, however, the voter interacts with a machine for casting and recording their votes and they do not readily know what the machine does with the vote(s) cast and if their vote is recorded to be tallied towards electing winners. Consequently, the integrity and correctness of the outcome together with the transparency of the process and concerns over privacy preservation in an electronic election face question marks to be answered.

Despite the expectations for using provably reliable trustworthy means in electronic elections, the current state of affairs leaves much to be desired. Indeed some electronic voting tools as they are used now may be seen as a step back from traditional paper-based elections where tallying proceeds manually. For example, states of Australia rely on software which essentially resemble magic boxes [142]. No *transparent* information is provided so that observers know how *exactly* the *counting* has proceeded. Nor any form of independently *verifiable* evidence is yielded for the correctness of the computation carried out. This situation results in a low level of transparency and a huge trusted computing base (TCB) required from voters and third-party users to invest in the tools for employing them. The unsatisfactory state of affairs with using many of the electronic voting systems is unfortunately not simply limited to this specific instance.

What makes the situation worse is that, contrary to the election transparency requirements, these programs are mostly black boxes as well. The implementations of the tallying module of many of these voting systems are very often, although not always, kept secret due to legal excuses of confidentiality and protecting the property of the companies that produce them [95]. Therefore one can neither audit the election data and the process of computing winners of the election carried by the software nor they can audit the source code of the implementation of the vote counting algorithm to investigate its correctness.

The notion of end-to-end verifiability [17, 126] has been introduced in the context of election security in order to establish guarantees into the integrity and trustworthiness of an election result and also confidence in the preservation of voter privacy. End-to-end verifiability of an election comprises satisfying several sub-properties including eligibility verifiability, cast-as-intended verifiability, recorded-as-cast verifiability, and tallied-as-recorded verifiability.

- *eligibility Verifiability.* The electronic means employed in the election must verify that only eligible voters are able to cast a vote.

- *cast-as-intended verifiability.* They must provide proof that the vote is cast into the system in accordance with the voter intention.

- *recorded-as-cast verifiability.* Also, they must guarantee that the machine correctly records the cast-as-intended vote into the system.

- *tallied-as-recorded verifiability.* Moreover, it must yield evidence that the tallying progresses correctly according to the vote-counting algorithm used in the election and the ballots recorded as cast.

- *privacy preservation.* Also, the voting system must demonstrate proofs that content of the vote cast and the identity of the voter (as in relation with their vote and its content) is not tractable by either of the machinery employed or any third parties. As a result, the voters cannot be coerced to behave in any specific ways when participating in the election and their vote cannot be bought by any interested third party.

Some knowledgeable scholars together with skilful engineers have drawn on solid academic studies and technology to design, develop and implement a voting system that accommodate STV schemes. As end-to-end verifiability is a highly regarded security standard, it has become a selling point for many such implemented voting systems. However, truth be told, end-to-end verifiability is an ideal attainable only in principle but not in practice. Although the theory behind the design and development of these voting systems is hardly questionable, there is nonetheless a gap from the theory to engineering a concrete provably trustworthy voting system. When it comes to engineering, there are several trusted layers that have to be removed. In particular, assuming that the cast-as-intended and recorded-as-cast properties are verifiably correctly engineered, there are trusted layers encountered while engineering software for tallying votes that must be addressed as well. Currently, the tallying module of every voting system implementing an STV algorithm for counting votes suffers from a considerable TCB.

Every implemented voting system currently in use for STV family of algorithms which claims to satisfy the end-to-end verifiability criteria either does not formally verify the correctness of the tallying component or it relies on the so-called light-weight tools for the verification of the tallying program. Unfortunately, light-weight tools are themselves complex programs which lack proofs of correctness. Consequently, using them requires a significant amount of trust. Moreover, even the theoretical framework behind a light-weight verification software, let alone its implementation, is not proven to be complete. Consequently, the tool may not, and one can never be absolutely sure when and if it does, identify every possible error in the tallying software code. As a result, a light-weight verification tool can only increase the confidence in the correctness of a program. It however does not provide a guarantee into the full correctness of the vote counting software.

Indeed, at the moment, the voting systems accommodating STV, and also other schemes too, mainly focus on the verifiability of the vote casting and recording, e.g. through zero-knowledge proofs or auditability of election data. They treat counting votes as a secondary easily checkable step that everyone can simply participate in once there is proof that the recorded votes reflect the will of the voters. Contrary to

such simplifications, correct implementation of complex algorithms such as an STV scheme is not a straightforward task. For instance, the vote counting software that is used in Australia's most populous state, NSW, was found to contain errors that had an impact in at least one seat that was wrongly filled with high probability. This was reported in specialist publications [18] as well as the national press[112, 24].

To avoid repeating mistakes such as the above in tallying votes with STV schemes, a voting system must provide trustworthy means and techniques to guarantee the correctness of counting votes in a verifiable way. End-to-end verifiability alone is not enough for engineering such a trustworthy voting system for two reasons:

a. end-to-end verifiability, and naturally every existing implementation of an end-to-end verifiable system, basically aims at proving the correctness of the tallying phase by giving every voter the chance to *audit data flow* while counting votes. Unfortunately, auditability can only facilitate but not cultivate trustworthiness of the tallying correctness as mistakes mentioned above have happened in front of the public eyes. Auditability may work for detecting errors while counting with simple voting schemes such as first-past-the-post but not for computation with STV algorithms which are known for their complexities. For detecting possible errors in an instance of vote counting with an STV algorithm, auditability has to be accompanied by second provably trustworthy standalone software that meticulously checks, based on the auditable recorded ballots, if computation performed by the vote counting software has progressed correctly. However, none of the existing implementations of an end-to-end verifiable voting system for STV offers such standalone software.

b. Moreover, auditability that end-to-end verifiability relies on is meant to guarantee that any error in the election process including tallying is detectable. But its aim is not trying to prevent them from occurring in the first place. This stems from the fact that end-to-end verifiability merely examines the voting system from the voters perspective where it is the authority that has to give the voters the confidence in the trustworthiness of the outcome. However, there is another dimension of the trustworthiness in the context of elections as well to which one has to attend. It comes from considering the situation from the viewpoint of democratic institutions which operate an election. These democratic systems themselves also need to have enough confidence in the correctness of software deployed in the election which they run. Such guarantees safeguards the trust found in the institutions that encourages and keeps the people's faithful in the authenticity of their authorities and the system. Otherwise, imagine mistakes such as above happen during tallying votes in important Senate or parliamentary elections. The unleashed torrent of doubts into the public by the media against the democratic system can easily come to raise questions on why the authorities of the system do not employ reliable voting means so that errors do not occur at first instance, although they may be detectable through existence of an end-to-end verifiable voting system in effect for running elections.

Formal verification tools and techniques are now mature enough for creating a provably trustworthy tallying module for satisfying the universal verifiability of tallied-as-recorded property. This thesis focuses (only) on the tallying phase in an electronic election where the vote counting algorithm implemented as a choice function operates on some input ballots and outputs the final end result where the winners are produced. Therefore, we do not question how the input ballots have been obtained and if the security expectations in casting and recording phases have all been met. We do guarantee, as we shall demonstrate, that our framework accomplishes the following milestones.

- **Universal verifiability of tallying with a minimal TCB.** We provide auditability of tallying data and verified tools that have a minimal TCB. Using them allows any voter to satisfy themselves that a vote counting program synthesised from our framework correctly computes the output winners for given input ballots recorded.
- **Software/Computation Correctness.** Counting votes must proceed based on proven correct, hence reliable, tools. We design and develop a framework for formal specification and verification of various STV schemes in a modular way. The framework provides the construction of formally proven correct implementations of a wide range of STV schemes for computing election winners.
- **Transparency of tallying.** The constructed implementations above, upon each execution on an input, produce a printout that consists of all of the necessary information to know how winners are computed. This printout makes the tallying phase auditable which, in turn, allows the verifiability of the tallying result.
- **Practicality.** We are motivated to better our world by resolving the embarrassing situation of counting with STV schemes. Therefore feasibility of deploying the software for real-world elections and ease of adapting it for enhanced usability are factors to which we are attentive.

1.1 Approach

Our quest for achieving the above horizon begins by asking simple but central questions. What is STV really? Why despite conspicuous differences perceived among STV algorithms, they are still categorised as members of one family named STV? What is it that constitutes a family resemblance which unites these diverging schemes as STV?

Through an analysis of various STV algorithms, we come to see common underlying data and algorithmic structure existing in the schemes. This structure forms the core of STV algorithms. We abstract this core concept of STV to frame it as a model of computation, particularly an abstract finite state machine. Concrete instances of STV are then realised as instantiations into the machine. As this novel understanding

is fundamental to the thesis, we shall provide further details to shed light on the way in which we are characterising STV.

To count an election according to STV, one usually computes a quota dependent on the number of ballots cast (often the Droop quota [46] and then proceeds as follows:

1. Count all first preferences on ballot papers;
2. Elect all candidates whose first preferences meet or exceed the quota;
3. Transfer surplus votes, i.e. votes of elected candidates beyond and over the quota are transferred to the next preference;
4. If all transfers are concluded and there are still vacant seats, eliminate the least preferred candidate, and transfer his/her votes to the next preference.

On the other hand, all variants of STV share a large set of similarities. All use the same mechanism (transfer, count, elect, eliminate) to progress the count and, for example, all cease counting once all vacancies are filled. description hides lots of detail, in particular concerning precisely which ballots are to be transferred to the next preference. Indeed, many jurisdictions differ in precisely that detail and stipulate a different subset of ballots be transferred, typically at a fractional weight (the so-called *transfer value*). For example, in the Australian Capital Territory (ACT) lower house STV election scheme, only the *last parcel* of an elected candidate (the ballots attributed to the candidate at the last count) is transferred. In contrast, the STV variant used in the upper house of the The Australian state of Victoria transfers *all* ballots (at a reduced transfer value). Similar differences also exist for the transfer of votes when a candidate is being eliminated.

We abstract the commonalities of all different flavours of STV into a set of minimal requirements that we (consequently) call *minimal* or *generic* STV. It consists of:

- the data (structure) that captures all states of the count
- the requirements that building blocks (transfer, count, ...) must obey.

In particular, we formally understand each single discrete state of counting as a mathematical object which comprises some data. Based on the kind of data that such an object encapsulates, we separate them into three sets: initial states (all ballots uncounted), final states (election winners are declared) and intermediate states. The latter carry seven pieces of information: the list of remaining uncounted ballots which must be dealt with; the current tally of each candidate; the pile of ballots counted in each candidate's favour; the list of elected candidates whose votes await transfer; the list of eliminated candidates whose votes await transfer; the list of elected candidates; and the list of continuing candidates. Basically, they record the current state of the tally computation.

We realise transitions between states, corresponding to acts of counting, eliminating, transferring, electing, and declaring winners, as formal rules that relate a

pre-state and a post-state. These rules are what varies between different flavours of STV, so minimal STV does not define them. Instead, it postulates minimal *conditions* that each rule must satisfy. An *instance* of STV is then given by:

1. *definitions* of the rules for counting, electing, eliminating, and transferring;

2. formal *proofs* that the rules satisfy the respective conditions.

We sometimes refer, somewhat informally, to the conditions the various rules must satisfy as *sanity checks*. They are the formal counterparts of the legislation that informs counting officers which action to perform and when. Each sanity check consists of two parts: the *applicability condition* specifies under what conditions the rule can be applied while the *progress condition* specifies the effect of the rule on the state of the count. For example, the count rule is applicable if there are uncounted ballots and reduces the number of uncounted ballots.

The STV Machine. A marvellous phenomenon emerges from our analysis; we can see STV as a finite state machine. The aforementioned discrete states of computation constitute the states of the machine. The transitions which enable us to move from one state to another are the transition labels of the machine. Moreover, the sanity checks collectively comprise a small-step semantics for the machine.

The machine is conceptualised to have a feature which traditional finite state machines lack. Given an input x to the machine, the model produces as output a value y and a certificate ω. The certificate, also called evidence, is essentially a trace of all of the states and the transitions that the machine visits from the initial state where x was given to the machine all the way through intermediate states of the machine to finally reach the terminal state where the value y is obtained. We refer to this property as certification and name the act of certification as certifying.

We establish three main properties of this generic STV machine. The first, called reducibility property, asserts that each application of any of the STV generic transitions reduces a complexity measure. The second property called applicability states that at any non-final state of the computation, at least one of the generic transitions is applicable because it satisfies its sanity check requirements. Drawing on the reducibility and applicability properties of the machine, we establish a third property stating that the generic STV machine terminates at a final state y, upon each execution on a given input x, where a certificate ω for this instance of computation is constructed as well.

1.2 Solution

We first design and develop in the theorem prover Coq [116] a modular framework [119] for formalisation and verification of various STV algorithms which facilitates producing a provably correct implementation for each of the algorithms formalised. To elaborate more, The framework has a base module which contains formalisation of the generic STV machine with all of its components. In particular,

the transition labels of the machine are formalised as generic parameters to be instantiated later. The machine is specified as the record of the transition labels which satisfy the formal sanity check requirements (small-step [132] constraints). We then prove that the generic machine has the three aforementioned properties, namely reducing a defined complexity measure, the applicability of a transition label at each non-final state, and the termination property.

On the other hand, the framework has several dependent modules each of which is devoted to formalisation and verification of concrete instances of STV. Basically, formalisation of the instances happens through instantiations of the generic transition labels. To instantiate the transitions, one has to specify them in Coq based on the textual description of the counting mechanism of that particular STV. To verify them one only needs to discharge the sanity checks to show that the instance is indeed legitimately an STV algorithm and to therefore automatically extend the verification obtained for the machine to the instance.

Coq has automatic means of extracting program [91] into the Haskell programming language. By using the extraction facility, we extract each dependent module from Coq into Haskell to obtain executable vote-counting programs for a variety of STV algorithms. Thanks to the extraction mechanism of Coq, we are given a high level of guarantee that the executable vote counters behave in accordance with their specification in Coq.

Each extracted program, upon any execution on a given input x, outputs winners of the election y which is accompanied by a run-time generated certificate ω. The run-time certificate is essentially the extracted counterpart of the formally specified certificate notion inside Coq. Consequently, ω comprises the list of machine states visited and transitions applied in order to compute the winners y for the input x. A certificate therefore consists of all of the necessary and sufficient information to know in order to see how the extracted program has computed the final result. Also the certification provides the opportunity for independent validation of the computation carried out without relying on the source program used for computing the election's result and the certificate.

Offering a certificate to voters is fantastic but by itself is not perfect. Suppose some program $\mathcal{P}$, whose source code is confidential, is an implementation of some STV algorithm $\mathcal{S}$. Also suppose that this program is executed on an input x and produces the output y and the certificate ω, where it is claimed that y is the correct output witnessed by the evidence ω. How can one verify or reject the claim without redoing the computation on x? More interestingly, how can one investigate such claims in a way that the method can be modularly extended to work for any STV algorithm rather than merely $\mathcal{S}$ and any implementation of $\mathcal{S}$ instead of only $\mathcal{P}$? Such a method can particularly, but not exclusively, be applied to independently of how extracted vote-counting programs above are produced, check the correctness of their output certificate.

Our method for answering the question is a second standalone modular framework designed for developing verified certificate checkers, also called verifiers, for various STV algorithms. Formalisation and verification of certificate checkers take

place in the environment of the theorem prover HOL4 [131]. By using the proven trustworthy translation tool of the CakeML [136], we obtain equivalent certificate checkers inside the environment of CakeML. We then synthesise machine executable certificate checkers for numerous STV algorithms by relying on the verified compiler of CakeML.

Formalisation, verification and synthesis of certificate checkers proceed modularly. However, each single certificate checker operates for a particular STV instance. A checker for a scheme $\mathcal{S}$ accepts as input a certificate produced based on computing under $\mathcal{S}$. Recall that a certificate is a list of states of computation. The checker recursively consumes two consecutive states in the certificate and checks if the transition from the antecedent state to its succedent has happened in accordance with the counting mechanism of $\mathcal{S}$. This process continues until either all of the steps in the certificate are verified as valid or an error occurs where an invalid step is encountered. In the former case, the certificate checker returns a message communicating validity of the certificate, whereas in the latter an error message of invalidity is output informing us where the error happens.

1.3 Contributions

A good tree bringeth not forth corrupt fruit; neither doth a corrupt tree brings forth good fruit[1]. A new perspective is weighted against the fruits that it bears and the possibilities which it opens. Our work results in immense fecundity.

1. **Correctness.** Every single computational assertion which we rely on for computation is verified. Moreover, in light of the content of the proofs of the theorems which we establish in Coq and HOL4, the counting mechanism with STV algorithms is also proven correct. Additionally, the certificate checkers in their entirety are verified. Hence not only auxiliary components are provably reliable, but also the whole processes of computation are verified as well.

2. **Transparency.** We offer certificates which consist of all of the necessary information to transparently see how winners are obtained. Having a certificate available allows any voter to audit the tallying process independently of any election authorities.

3. **Universal verifiability of tallying.** A certificate, through auditability of data, gives the opportunity to any voter for independently investigating the correctness of the tallying procedure by themselves. They can check a certificate manually or have a certificate checker to machine check its correctness. However, we offer verified means that can effectively carry out this task for voters.

   The trusted base to lay in vote counting authorities and tools which they use should become ideally zero but at least practically as small as possible. By minimising the TCB while engineering a tallying module, we approximate the

[1] New Testament, Luke 6:43

universal verifiability of the tallying phase as much as the currently available state-of-art-technology allows us to. Our certificate checkers are proven correct down to machine code. We obtain this by using CakeML which provides us with end-to-end verification of checkers. Therefore several untrusted verification layers, such as translation of assertions from the theorem prover to the compiler environment or from the compiler to machine executable code are eliminated. We hence minimise the trusted base required to invest in our tools.

4. **Practicality.** The theoretical insight behind the framework together with elegance in employing formal engineering tools brings about several advantages. In particular, we benefit from the followings.

   (a) **Efficiency:** Experimental results on real historical data of elections conducted in the ACT state of Australia witness feasibility of the software produced by our framework. For example, our software computes winners of the biggest electoral district for the legislative assembly elections in Australia in just 22 minutes and checks the certificate produced in simply five minutes.

   (b) **Adaptability:** We achieve a great degree of adaptability of the framework. It is adaptable to various STV algorithms used in parliamentary and Senate elections, and also in institutions and unions across continents for electing their board of trustees or presidents.

   (c) **Usability:** Modularity of the framework's design provides us with satisfactory usability. It brings about two fruits in particular;

      - **Proof automation:** We automate a significant amount of formalisation and verification away so that they are dealt with by us, rather than users, and done once and for all. In the framework developed in Coq, we mainly automate proofs through instantiations into the base module. For the framework in HOL4, we automate proofs by developing the bulk of the formalisation and verification in the base and then simply calling the base inside dependent modules.
      - **Abstraction:** technicalities of formalisation, verification, and discharging tedious complicated proof obligations, such as termination proof, are sufficiently hidden from users who may need not knowing about them. Especially, since CakeML has a challenging learning curve, we fully automate and abstract dealing with CakeML. Therefore when it comes to synthesising machine executable code for checkers, users only need to follow instructions presented to them.

   (d) **Cost-efficiency:** Furthermore, certificate checking software is highly cost-efficient means for verifying the correctness of vote counting programs. Verification of every single implementation of an algorithm is a significant resource taking Herculean task. However, instead of verifying every implementation, we design a checker that can check all of those imple-

mentations for correctness. To guarantee every one of trustworthiness of this procedure, we then verify the certificate checker once and for all.

5. **The Framework.** the unique environment that we have created allows a modular treatment and comparison of various STV algorithms from *a purely syntactic point of view*. This in turn opens the chance for those interested in studying and comparing STV algorithms from a computational social choice theory perspective. They can use our framework for modifying specific parts of one or some STV algorithms, extract an executable program and objectively measure the effect of the modifications. Such analysis of the STV family is subject of our future work.

1.3.1 On Integrating Our Framework into A Voting System

We note that our framework is obviously not meant to be a standalone piece of software for running an entire electronic election. In contrast, for the sake of end-to-end verifiability criteria, tools generated from our framework should be embedded as part of a more encompassing voting system that takes care of the secure vote casting and vote recording as well. At this point, we find it useful to outline how our work can be combined with the existing voting systems for STV schemes such as Prêt à Voter [124] and vVote [5].

The existing voting systems that aim at establishing end-to-end verifiability and accommodate STV algorithms, basically the Prêt à Voter and vVote, can deploy the vote counting programs extracted from our Coq component to compute election winners and output a certificate for the computation performed. These systems can also make the corresponding certificate verifier of the STV algorithm used in the election publicly available so that any voter can run the verifier on the certificate.

We note that every certificate contains the plaintext votes as recorded in the voting system. As it is known [106, 16], STV voting schemes are potentially more vulnerable to the "Italian attack" [106] than non-preferential voting schemes [106]. Therefore an astute reader may object that any voting system wanting to adapt our tools would also suffer from the voter coercion problem. In our defence, we next elaborate more on the current situation with the existing implemented voting systems for STV schemes to see that indeed the problem stems from their technique used for vote anonymisation.

To elaborate more, voting systems currently available that support elections having an STV scheme all rely on mixnet techniques [30] for anonymising votes and eliminating the link between the voter and their vote cast. The output of the last mix in these systems is plaintexts consisting of the votes as cast into the system by voters. They then run the tally function on these plaintext votes to compute winners. On the other hand, it is argued [16], that outputting plaintext votes publically opens the opportunity for a coercer to force a voter to cast the first preference as the coercer wishes and then coerce the voter to choose an oddly occurring ranking of other competing candidates as the rest of preferences. This way, the coercer can check with a

high chance if their votes appear among the plaintext supposedly anonymised votes. This attack is technically referred to as the Italian attack. Consequently, the existing voting systems for STV, and any other emerging ones that rely on the same method of mixing suffer from vulnerability to voter coercion phenomena.

We do claim that a certificate does not make the hosting voting system any more vulnerable to the voter coercion problem than the mixing technique as currently exercised by the voting systems does. Indeed, our work basically addresses the tallying phase which happens independently of voters interactions that mainly take place during vote casting and vote recording. Privacy matters mainly arise at the vote casting and recording which preceded the tallying stage. Consequently, the level of the voters' privacy is not generically affected by using our tools.

To the best of our knowledge, there exists no end-to-end verifiable voting system currently deployed in real elections that use a homomorphic tallying technique for counting votes based on an STV algorithm. However, assuming there was one, our framework as it stands is more suited with mixnet techniques rather than homomorphic methods. Nonetheless, this is not a generic constraint on using our framework. In the future work, we shall outline how a voting system relying on homomorphic tallying may adapt our tools.

1.4 Publications

The author of this thesis is the main contributor to the following publications, all of which were accomplished during the PhD process.

1. Milad K. Ghale, Rajeev Goré, Dirk Pattinson: A Formally Verified Single Transferable Voting Scheme with Fractional Values Electronic Voting - Second International Joint Conference, E-Vote-ID 2017, Bregenz, Austria, October 24-27, 2017, Proceedings (won the best paper award)

2. Milad K. Ghale, Dirk Pattinson, Ramana Kummar, Michael Norrish: Verified Certificate Checking for Counting Votes, Verified Software. Theories, Tools, and Experiments- 10th Int. Conf. VSTTE 2018, Oxford, UK, LNCS Springer 2018

3. Milad K. Ghale, Rajeev Goré, Dirk Pattinson, Mukesh Tiwari: Modular Formalisation and Verification of STV Algorithms. Electronic Voting - Third International Joint Conference, E-Vote-ID 2018, Bregenz, Austria, October 2-5, 2018, Proceedings

4. Milad K. Ghale: Engineering Software for Modular Formalisation and Verification of STV Algorithms, 20th Int. Conf. on Formal Engineering Methods (ICFEM) 2018, Gold Coast, QLD, Australia, Nov. 12-16, 2018, Proceedings, 459-463.

5. Milad K. Ghale, Dirk Pattinson, Michael Norrish: Modular Synthesis of Verified Verifiers of Computation with STV Algorithms. Forthcoming in FormaliSE: 7th international conference on formal methods in software engineering.

1.5 Thesis Outline

The thesis is organised as follows.

Chapter 2. We begin the chapter by analysing the data structure and the algorithmic vote counting mechanism that is common between STV algorithms. We then discuss how to model these underlying commonalities as an abstract state machine. For pedagogical purposes and also limitations of presenting all of our actual formalisation of the machine, in Chapter 2 we only discuss an informal mathematical, pen-and-paper presentation of the machine components.[2]

Chapter 3 and Chapter 4. These chapters together discuss the first standalone component of our framework which we use to construct verified vote counting programs for various STV schemes. The chapter presents the formalisation and verification of the generic STV machine in Coq. In chapter 4, we demonstrate how a concrete STV algorithm is realised as an instantiation of the formalised STV machine discussed in Chapter 3. We present three such instantiations and discuss technicalities of the implementation and verification of the machine instantiations. Also, we explain how to obtain Haskell executable programs for these instantiations of the machine. Additionally, we demonstrate instances of concrete certificates and present experimental data as evidence for the feasibility of deploying the vote counting tools created in real-size elections.

Chapter 5 and Chapter 6. These chapters discuss the second component of the framework which deal with producing certificate verifiers for checking the correctness of different STV algorithms. Chapter 5 details how we formalise the generic STV machine in HOL4 as a generic tool for verifying the correctness of computation carried out by STV vote counting programs. Chapter 5 also includes implementing a computational counterpart of this generic machine and then verifying the correctness of the implementation against the formalised specification of the machine. In Chapter 6, we discuss what an instantiation of the machine in HOL4 amounts to and illustrate the process of the instantiation for some STV algorithms that are used in real elections. Moreover, we discuss how from an instantiation one can obtain a verifier that checks the correctness of certificates produces for computation with the STV algorithm instantiated in the generic machine. Furthermore, we show how to proceed with translating and compiling executable verifiers using CakeML. Finally, we present experimental results with the executable verifiers on concrete certificates.

Chapter 7 and Chapter 8. We discuss the related work in Chapter 7 and conclude the thesis with an elaboration on future work in Chapter 8.

[2]Throughout the monograph, we reserve the word "formal" specifically referring to formalisation in either Coq, HOL4 or CakeML. The discusion in Chapter 2 is therefore completely informal.

1.6 Tools Used for Constructing the Framework

Trustworthy correct software is built by proven reliable means. We develop both components of the framework in safe provably reliable environments of the theorem provers. A theorem prover itself is software that has a calculus for manipulating computation and provides an interactive interface to users to formalise (implement) their model in a general purpose language given by the theorem prover and then verify the formalised model against its specification. By the term formalisation one simply refers to the practice of implementing the model in a theorem prover. Specification means defining properties in the language of the theorem prover which one expects the formalised model to satisfy. Verifying an implementation then amounts to relying on the calculus used by the theorem prover to prove that the formalised model indeed meets its speciication.

The process of verification proceeds through interaction with the theorem prover environment where the user uses so-called tactics to guide the theorem prover towards finalising the proof of the statement desired to be proved. As the implementation of the calculus (and tactics as well) is proven correct, one can be confident that proofs carried out in these environments are highly reliable. Moreover, theorem provers are excellent at identifying corner cases which a human may fail to recognise. Therefore some trusted computing base is eliminated in light of the verification carried out to demonstrate that the implementation behaves correctly in every possibly occurring situation.

We briefly describe the theorem provers Coq and HOL4 and the CakeML compiler used to build our work on. We develop the first component of the framework in the environment of Coq. We note three characteristics of Coq on which the wrok heavily relys.

- *Dependent types.* Calculus of (Co-)Inductive of Constructions (CIC) [35] is the underlying calculus based on which Coq operates. CIC not only allows defining primitive data types such as lists and trees but it also accommodates constructing types whose definition depend on terms. These later constructs called dependent types can be used, among many other applications, to reason about program properties and behaviour [117, 31]. For example, one can define a function whose input is a list of trees and whose return value is a dependent type where the second value is a proof that the first component consists of only those trees in the input that their height does not exceed some particular number.

- *Proof-as-program.* In CIC similar to other typed lambda-calculi in the lambda cube [12, 73], the notion of a program is expressed through the lambda abstraction rule and its operation is defined by the application rule which determines the effect of applying the program to an input. On the other hand, CIC brings the notion of proof into its syntax through the proposition-as-type isomorphism [98, 97]. In light of the isomorphism, a proposition is true only if it is inhabited by some terms. A term inhabiting a proposition (now viewed

as a type) is called a proof-object. Proof-objects are basically constructed by application of CIC's rules, including the lambda abstraction, for building new terms/types. Therefore in Coq, a theorem or lemma is simply a type and a proof carried out in Coq to establish the correctness of the theorem or lemma is a program that constructs a proof-object inhabiting the type.

- *Extraction of proof carrying code.* Coq has a built-in code extraction mechanism. The extraction facilitates producing functional programs into Haskell, OCaml, or Scheme language. The extracted code is the pretty-printed versions of the entities formalised in Coq. In view of the preceding points on dependent types and proofs operating as programs, the extracted compilable functional code carries its correctness proof within itself (e.g. see Wadler [140]).

We develop the second component of the framework using HOL4 and CakeML. HOL4 is a theorem prover which uses the ML-style of programming for defining data types, functions and other computational entities. It uses a variant of classical Higher Order Logic [4] for declaring predicative assertions and reasoning about declarations specified in the theorem prover. The core calculus of HOL4 is a rewriting system [10] working alongside well-developed libraries of theorems established on a variety of data structures, including lists and tactics developed for automating rewriting definitions to prove desired properties about the formalised model. We particularly benefit from three features that HOL4 offers:

- *Well-developed libraries/tactics.* HOL4 has well-developed libraries comprised of verified assertions of operations on list structure. This motivates developing the data structure of the second software component on top of list structures. By structuring the data on lists, we facilitate us and users with the exploitation of already verified tools to formalise the framework and avoid inventing the unnecessary from the scratch. HOL4 also has well-developed tactics for discharging proof obligations on assertions involving list structure and operations on it. We therefore come to provide us with means for ease of verification of the formalised assertions.

- *Separation of proofs and programs.* In HOL4 one defines a computational implementation and then separately specifies logical declarations that are properties expected from the implementation to do. Then one can proceed to establish proofs that the implementation performs what its specification requires. Thanks to the proofs, whenever one needs understanding what an implementation does, they can simply refer to the specification which is the job description of the computational entity. This separation of programs, specification, and proofs from one another has the advantage of enhancing the degree of modularisation of the software component based on the functionality that each module has in the structure. Moreover, the separation enables users whose backgrounds are distant from functional programming to understand what and how the software component does, which consequently improves the usability of the framework. This ease of understandability can come about because the

specification of a program happens using a syntax similar to first-order logic which many users have already encountered having done courses in mathematics as high school or undergraduate students.

- *Connectability to CakeML.* One of our main motivation for using HOL4 is the proven trustworthy connection that is has with the CakeML through CakeML's verifed proof-translator tool. Once assertions are translated correctly from HOL4 environment into CakeML's, using the CakeML verified compiler, we can then synthesise machine executable code for certificate verifiers that are guaranteed to behave according to their formal specifications. The combination of HOL4 and CakeML thus allows us to minimise the TCB in means used for producing certificate checkers and also in performing computation with the verifiers produced.

CakeML is a terminology used in three different but interrelated senses.

- First of all, the word CakeML refers to a functional programming language comprising a large subset of the ML-style functional language.
- Second, the word also refers to an ecosystem of proven correct programs and proofs, and verified tools including a compiler backend and two compiler front-ends. The first compiler front-end is a PEG parser[3] itself consisting of a parser and a type inferencer that have been proven sound and complete. The second front-end is a proof-translator proven to produce an equivalent abstract syntactic representation (AST) of a computational assertion in HOL4 into an assertion in CakeMl's environment. The CakeML backend compiler mainly transforms an untyped AST to concrete machine code for some target architectures including x86_64, ARMv6 and ARMv8. The compiler backend has been proved to only produce machine code that is compatible with the behaviours of the source programs. Also, CakeML's ecosystem contains libraries of verified functions and definitions for modelling I/O interactions of a program with an operating system. The ecosystem, including the compiler backend and front-ends, are activated and accessed from within HOL4.
- Finally, the word CakeML also refers to the executable compiler of the language which we mentioned above. This executable compiler is obtained by boot-strapping itself. As the compiler backend which has boot-strapped itself is proven correct together with the front-ends, the resulting executable compiler provably correctly compiles the language.

1.6.1 Source Code of the Framework

The interested reader can find the source code of our framework, including the modelling, implementation, and the proofs online at the following address:

[3]PEG stands for Parsing Expression Grammar, which is a type of formal grammar for describing a language [56]. A PEG parser is a parser that parses a grammar whose rules are in PEG form.

https://github.com/MiladKetabGhale

As you see in the above Github account, there are four repositories that host the earlier versions of the framework components and the most recent ones based on which the thesis has been written. More specifically, the source code of the Coq component is at https://github.com/MiladKetabGhale/Modular-STVCalculi and the one for the HOL4-CakeML component is found at https://github.com/MiladKetabGhale/Modular_Checker.

Chapter 2

Specification of the Generic STV Machine

In the first section of this chapter, we provide an explication [96] of the underlying data and algorithmic structure found in STV algorithms. Through this analysis, we come to abstract a general version of STV that we call *generic STV*. We then in the subsequent sections formulate the generic STV as an abstract finite state machine and explain the machine's components in a simple mathematical language. Finally, we discuss some of the main mathematical properties which this model of computation has.[1]

2.1 The Generic STV

STV algorithms are mostly described as texts in natural language. Our quest therefore begins with analysis of the textual specification of these schemes. The objective is to attain an insight into what STV is in a way that realises similarities of instances and also accommodates particularities of each case.

Our thorough examination of different STV algorithms reveals a common underlying data and algorithmic structure which lays the foundation of the STV family. Despite the fact that concrete embodiments of the structure vary, the structure itself invariably remains there. Presence of the structure in each individual STV separates them from other vote counting schemes which are not STV. On the other hand, variations in the embodiment of the structure produces distinguished individual STV schemes. We detail about the structure in the current chapter. We shall discuss how various STV algorithms emerge from the way the structure is materialised in chapters 3 and 4.

To provide a tangible foothold for the discussion, we first present the algorithm used in the Australian Capital Territory (ACT) of Australia for electing representatives of the lower house. Hence the reader comes to a clearer image of the kind of algorithms which are subject of our curiosity. We then sensibly illustrate what our approach is and outline how it unfolds into a solution.

[1] We remind the reader that the content of this chapter is entirely considered as an informal discussion. Formal treatments of the STV are presented in the four subsequent chapters.

2.1.1 A Concrete Instance of STV

The counting protocol determines design of the ballots and the acceptable way of expressing one's vote on the ballot.

Voting. Voters mark preferences for candidates in the order of their choice by using the numbers 1, 2, 3, 4, 5 and so on $\cdots$ If a voter does not fill in the minimum number of squares as instructed, the vote will be counted up to the point where preferences stop, so long as a single first preference is shown.

Ballot Paper. Candidates' names are listed on the ballot papers in columns.

Moreover, the protocol dictates how to perform computations in two sets of instructions. It first gives the general steps to be taken whenever appropriate and then details conditions required for those steps to occur.

Step 1. Count the first preference votes for each candidate.

Step 2. calculate the quota according to the following formula;
(total number of valid votes / (number of vacancies) + 1) + 1

Step 3. Any candidate with votes equal to or greater than the quota is declared elected.

- if all vacancies have been filled, then the election is complete.
- if all vacancies have not been filled, does any candidate have more votes than the quota?
  - if yes, then go to step 4
  - if not then go to step 5

Step 4. Distribute the successful candidate's surplus votes to continuing candidates according to the further preferences shown on the ballot papers by those voters.

Step 5. If there are more continuing candidates than there are vacancies remaining unfilled, exclude the candidate with the fewest votes and distribute this candidate's votes to continuing candidates according to the further preferences shown by those voters. Calculate each continuing candidate's new total votes then go back to step 3. Or, if the number of continuing candidates is equal to the number of vacancies remaining, all of those candidates are declared elected and the election is completed.

The protocol further elaborates on the constraints for and effects of application of counting, electing, transferring, or eliminating candidates;

Counting the first preferences. Count the number of first preference (or number " 1") votes for each candidate. After all the formal first preference votes are counted, the quota can be calculated. Any candidate who has votes equal to or greater than the quota is now elected.

Transferring surplus votes from elected candidates. If a candidate receives a total number of votes equal to or greater than the quota, the candidate is elected. If an elected candidate has received an exact quota of votes, all of those votes are set aside and not counted further. The value of the surplus votes gained by an elected candidate is passed on to other candidates according to the preferences indicated on ballot papers by the voters. If a candidate has received more than a quota of first preference votes, all the ballot papers received by the candidate are distributed at a reduced value called a fractional transfer value (see below). If a candidate has received more votes than the quota following a transfer of votes from another elected candidate or from an excluded candidate, only that " last parcel"of ballot papers that the candidate received are distributed to continuing candidates at a fractional transfer value.

After the surplus votes from an elected candidate have been distributed, the total number of votes which each candidate has received is recalculated. Any further candidates that have votes equal to or greater than the quota are elected. Provided vacancies remain to be filled, the surplus votes of any newly elected candidate are now also distributed one by one.

The fractional transfer value. The fractional transfer value is calculated using the following formula:
(number of surplus votes) / (total number of ballot papers with further preferences shown)

Excluded candidates. If vacancies remain to be filled after all surplus votes from elected candidates have been distributed, the process of excluding the lowest-scoring candidate begins. The candidate with the smallest number of votes is "excluded" and his or her ballot papers are distributed to continuing candidates according to the preferences shown by the voters. Ballot papers from excluded candidates are distributed at the value at which they were received by the excluded candidate. Ballot papers received by the candidate as first preference votes have a value of " 1" , while ballot papers received following the distribution of a surplus will have a fractional transfer value. This will vary depending on the group of surplus votes from which they were received. At each stage after ballot papers have been distributed from an excluded candidate, the total votes received by each continuing candidate are recalculated to determine whether any candidate has received votes equal to or greater than the quota. The process of distributing surplus votes from elected candidates and excluding the candidate with the fewest votes continues until all vacancies are filled.

At this point we have come to an intuition of what STV schemes resemble. This mere intuition could rise to an insight if one searched across STV members to find patterns of similarities.

2.1.2 Data Structure

Conspicuous data structure similarities exist between different STV algorithms. We identify six types of data consistently present in the schemes.

- **Ballot format.** STV belongs to the category of preferential voting system. In preferential voting, voters have the privilege of choosing more than one candidate by ranking them according to an order. As a result ballots are designed to facilitate the preferentiality of the scheme. This usually happens by positioning the list of competing candidates in columns or rows and placing boxes beneath or in front of their names to be filled by numbers.

  Moreover, each ballot has a fractional value which is initially set as $1/1$. The fractional value may reduce to another fractional value as the counting proceeds in accordance with the algorithm used.

- **Valid vote.** STV protocols have restrictions on the legitimate way of expressing preferences on a ballot paper. There are mainly two kinds of constraint. One of them regards the least number of preferences that voters must state on the ballot. The other one concerns ordering candidates on the ballot. If and when a ballot respects the legal way of filling a ballot out, the vote cast is called *valid* or *formal*. However, any ballot filled with disregard to such conditions is set aside as *invalid*, or *informal*.

  The least number of preferences that must appear on a ballot differs among STV protocols. For example the STV of ACT state requires at least five candidates to be ranked on a ballot. But the STV used for the local government elections in Tasmania state of Australia instructs voters to express at least one preference.

  There is nonetheless accordance on the ordering of the candidates on ballots. STV enforces voters to order candidates by using ordinal numbers starting at 1, and continuing until the requirement for stating (at least) a minimum number of preferences dictated is met.

- **Vacancies.** Every election is a decision-making process on how to distribute some vacancies among participating candidates. Hence, there is a parameter for number of seats in STV algorithms as well. Usually STV schemes are employed in constituencies with more than one seat so that the schemes reflect the proportionality of representation. A system is proportional, roughly, when there is a correspondence between the percentage of votes attracted by parties and the percentage of seats won by those parties. In the literature on social choice theory, there is agreement [54] that STV algorithms should be adapted in elections where vacancies equal to or exceed five so that the schemes approximate proportionality to a satisfactory degree. Where STV is used in constituencies with one seat to fill, it degenerates into a scheme called Alternative Vote (AV) [54], which we shall discuss towards the end.

- **Quota.** The least amount of votes needed for a candidate to attract in order to be declared elected, is called quota. Various STV algorithms define this threshold according to different formulae. Regardless of the particular formula chosen, usually definition of the quota depends on the number of valid ballots and vacancies. For example a popular formula for defining the quota called Droop [46], specifies the quota according to the following.

$$\frac{\text{(number of valid ballots)}}{\text{(number of vacancies)} + 1} + 1$$

  However the CADE STV [28] uses a majoritarian quota, which is defined as half of the valid ballots incremented by one, independently of how many the vacancies are.

- **Candidates.** Another significant parameter in STV schemes is the collection of individual candidates who compete for winning. Counting votes in STV systems proceeds based on the preferences for individual candidates rather than preferences for parties, as it is with PR systems [53].

  The fact that under STV candidates precede parties impacts on how we formally encapsulate the notion of ballots and subsequently design the input ballots to implementations of concrete STV algorithms. We therefore wish to exclude any chance for misunderstanding. We acknowledge that ballots in some elections under STV, such as the Senate in Australia, do offer the choice of voting for parties which is called group ticket vote [54]. Nonetheless, the option of voting for party with a group ticket vote does not essentially counter the point that the collection of individual candidates rather than parties is the main subject of interest in STV. The reason is simply that counting mechanism of STV schemes considers ordering of individual candidates on ballots and does not take into account to which party they belong'[54]. Indeed, voting for party only means that the voter has accepted to rank his preferred candidates in accordance with the way that the party to which they belong has listed them to be placed on the ballot papers.

- **Discrete Counting States.** Calculating winners of elections happening under STV protocol reaches an end after going through a *sequence of discrete states* of counting. Each of these states stores local information that informs the tally officers and scrutineers of the current status of the process. Having the necessary data available, the protocol can then instruct officers on how to progress the counting by specifying the way that current information should be updated to move to a new state.

  We catalogue these discrete states into three classes. The first class is the set of *initial* stages of the counting where the process is initiated by readying ballots cast for being tallied. The second one consists of all *intermediate* states of the count that officers encounter before announcing the winners. Each intermediate state encapsulates seven pieces of information:

1. A set of uncounted ballots, which must be counted;
2. A tally function recording the number of votes for each candidate;
3. A pile function recording which ballots are assigned to which candidate;
4. A list of already elected candidates whose votes await transfer;
5. A list of the eliminated candidates whose votes need to be dealt with;
6. A list of elected candidates; and
7. A list of continuing candidates.

The last set of states are the *final* ones where counting has terminated by announcing the end result. We shall demonstrate that knowledge of this much information is necessary and sufficient for any external observer to transparently understand how winners of an instance of any counting is calculated.

2.1.3 Algorithmic Pattern

Every STV algorithm has a mechanism for advancing the counting process. The mechanism instructs on how to move from one state of computation to another until a final state is reached where winners are announced. The mechanism basically comprises a collection of defined actions to take and two sets of constraints, namely applicability and progress conditions defined for each action. Applicability restrictions are the legislative clauses that specify to what states of computation and under what circumstances an action applies. On the other hand, the progress constraints are the legal clauses that determine the immediate consequences of applying the action by specifying the next state to which counting proceeds.

We revisit the STV of ACT presented earlier as an example. Step 4 requires the action of distributing the surplus votes of an elected candidate. In the second part of the protocol, it further explains under what conditions this action comes into effect. For instance, Step 3 delineates two of the constraints on transferring votes of a successful candidate which are (a) there still exist vacancies to fill and (b) some candidate(s) has already exceeded the quota so that there are votes to distribute in the first place. Moreover, it informs us of what an application of such a transfer brings about. For example, continuing candidates receive new votes and their tally subsequently changes. Finally, the Step 3 introduces an ordering on executing actions. If all vacancies have been filled, then an action for finishing the process executes. However, if there are vacancies, then either transferring applies or the action of eliminating.

The exact content of counting actions varies from one STV algorithm to another. For example, STV of ACT requires transferring only a portion, called last parcel, of the surplus votes of an elected candidate. But the STV used for the upper house elections in the Victoria state of Australia transfers all of the surplus votes rather than just the last parcel.

Also, the precise applicability and progress conditions differ among STV algorithms. For instance, the STV of ACT does not allow any action to precede or intervene application of transferring surplus of an elected candidate. In contrast, the STV

of the upper house elections in Tasmania state of Australia permits acts of counting ballots and electing before surplus of an elected candidate has been fully distributed.

There are nonetheless common algorithmic patterns in the STV family. When we survey different STV algorithms some *forms of action* invariably stand out. The forms are generic in nature and need to be materialised with specific substance to be brought into effect for actual counting. The forms are universally present, regardless of how they are embodied with particular definitions given by various schemes so that concrete actions emerge. We recognise eight forms of action.

1. **start** to determine the formal ballots to be dealt with.
2. **count** to deal with the list of uncounted ballots
3. **elect** to elect one or more candidates who have reached or exceeded the quota.
4. **transfer-elected** to distribute surplus votes of one or some of the already elected candidates.
5. **eliminate** to exclude one or some candidates who are weaker than others from the process.
6. **transfer-removed** to distribute votes of one or some removed candidates.
7. **elected-win** to terminate counting by declaring the already elected candidates as winners.
8. **hopeful-win** to terminate counting by announcing the already elected candidates and continuing ones as winners.

We also find commonalities in the applicability clauses among STV protocols. For example, in every STV whenever the vacancies are all filled, applicability condition defined for the finishing action applies and the counting process consequently comes to an end. As another example, whenever all of the uncounted ballots have been dealt with, there are surplus votes to distribute and there exists at least one empty seat, then transferring action applies to advance the procedure. We distil these common clauses to obtain, for each action, a minimal set of conditions that generally exist among the particular applicability constraints defined for actions in specific instances of STV.

Moreover, there are invariant patterns in the progress constraints pertinent to STV schemes. A careful examination of STV protocols shows that applying each action reduces the size of at least one of the following four objects: the list of all of the initially competing candidates minus those who have already been elected[2]; the list of continuing candidates; the number of ballots in the pile of the most recently eliminated candidate; the introduced backlog; and the list of uncounted ballots. Using

[2]To elaborate more on this part, assume the list l consists of all of the candidates who initially participated in the election. Then as tallying proceeds some candidates may be declared elected in which case the list l shortens.

a proper lexicographic ordering, this allows us to define a complexity measure on the set of machine states in such a way that each application of an action reduces the measure. The progress constraints of an action are therefore mainly the minimal conditions that ensure one of the above measure components decreases when applying the action. For instance in every STV scheme a progress condition on eliminating action is that the length of the continuing candidates in the updated post-state which we move to is less than the length of the same list in the pre-state.

The minimal applicability and progress constraints together with the forms of action shape the core algorithmic mechanism of STV. The minimal constraints are universally applicable to each action as defined by specific STV schemes. One can therefore reasonably expect the minimal conditions to apply to the forms of action rather than simply instantiations of the forms. As a result the minimal conditions give a meaning to the forms of action.

The data structure and algorithmic pattern discussed characterise STV as a family. They relate individual members together through an intrinsic resemblance in light of the commonalities which they have in the data content and algorithmic behaviour. An algorithm therefore belongs to the STV family if two requirements are met. First the six underlying data pieces, namely ballots of the preferential format, concept of valid votes, a quota, candidates to compete, and more significantly discrete states as described must inhabit the data structure of that algorithm. Second the forms of action given earlier should be the only forms defined and allowed as steps which advance the states of the counting, and the minimal applicability and progress should be the mechanism behind counting procedure.

A new phenomena results from the previous paragraph. We can introduce a general form of STV simply by considering the common data structure inhabitant in STV schemes to reside as its data structure and having the underlying algorithmic content mentioned to be the main mechanism of its actions and instructions. We shall refer to this broadly specified STV template as *generic* or *minimal* STV, and demonstrate how a wide range of real STV schemes emerge from instantiations into the template.

2.2 The Generic STV Machine

We think of the generic STV as an abstract finite state machine. Our discussion in this section is mainly focused on providing the reader with a simple description of what the machine semantics is comprised of. In chapters 3 and 4 when there are enough technicalities to draw on, we shall detail why we choose to model the machine and its semantics as they are formalised in our system.

As natural language is not suitable for formalistic approaches, we rely on a syntactic system for rigorously representing the machine components. This facilitates us with means of expressing and proving properties about the machine. Our formalism employs the symbols illustrated in figure 2.1.

We next explain some of the notations in the figure. In the subsequent chapter,

C	a set of candidates
$\mathbb{Q}$	the set of rational numbers
$\mathrm{List}(C)$	the set of all possible list of candidates
$\mathrm{List}(C) \times \mathbb{Q}$	the set of all (possible) ballots (with transfer values)
B	shorthand for $\mathrm{List}(C) \times \mathbb{Q}$
$\mathcal{A}_{cand}$	initial list of all of the candidates
st	initial number of vacancies
bs	initial list of ballots cast to be counted
bl	for characterising a backlog
bl_1	the first component of a backlog bl
bl_2	the second component of a backlog bl
b, d	to represent a ballot
ba, nba	list of ballots
c, c'	to represent a candidate
t, nt	list of tally functions each of which maps C into $\mathbb{Q}$
p, np	pile functions mapping C to $\mathrm{List}(B)$
e, ne	for characterising list of elected candidates so far
$[]$	representing an empty list
$l_0 :: ls$	l_0 is the head and ls is the tail of the list $l_0 :: ls$
$l_1 ++ l_2$	list l_2 is appended to the end of list l_1
h, nh	for representing list of continuing candidates in the election
qu	for the quota of the election as a rational number
$\|l\|$	length of the list l

Figure 2.1: Table of Symbols

we shall elaborate on the reasons for choosing to formalise them as they appear.

- **ballots.** A ballot is an ordered pair (l, q) where $l \in \mathrm{List}(C)$ is the preference order and $q \in \mathbb{Q}$ is the fractional value of this ballot.
- **tally.** Upon each round of applying the count action, we define an assignment which allocates to each candidate the amount of votes received up to that state of computation. We create a chronological list of such assignments and refer to the list as tally component.
- **pile.** Upon each round of applying the count action, we assign the ballots to continuing candidates based on the preferences expressed on the ballot. Ballots obtained in each round are placed in a list so that they are separated from the ones received at earlier stages.
- **backlog.** the backlog bl is a pair (l_1, l_2) where l_1 is the list of already elected candidates whose surplus awaits being transferred, and l_2 stands for the list of eliminated candidates whose votes awaits being transferred.

The discrete counting states discussed earlier constitute the machine states. We formally represent an intermediate state by $\mathrm{state}(ba, t, p, bl, e, h)$, where $ba \in \mathsf{List}(\mathcal{B})$, $bl_1, bl_2, h, e, w \in \mathrm{List}(C)$, t is a function from C into $\mathbb{Q}$, and p is a function from $\mathcal{C}$ into $\mathrm{List}(\mathcal{B})$. We use $\mathrm{initial}(ba)$ for the initial state, and use $\mathrm{final}(w)$ for a final one.

Having established terminology and necessary representations, we can mathematically define the states of the generic STV machine.

Definition 2.1 (machine states) *Suppose ba is the initial list of ballots to be counted, and l is the list of all candidates competing in the election. The set $\mathcal{S}$ of states of the generic STV machine is the union of all possible intermediate and final states that can be constructed from ba and l, together with the initial state* $\mathrm{initial}(ba)$.

The eight forms of action comprise the transition labels of the machine. For the moment the transitions are treated as generic labels which relate a pre-state to a post-state. We shall provide them with semantics shortly afterwards.

Definition 2.2 (machine transitions) *The set $\mathcal{T}$ consisting of the labels* ***start***, ***count***, ***elect***, ***transfer-elected***, ***transfer-removed***, ***eliminate***, ***hopeful win***, *and* ***elected win***, *is the set of transition labels of the generic STV machine.*

2.2.1 The Small-Step Semantics

The applicability and progress constraints of each transition label provide semantics for that transition. Hereafter we shall refer to these constraints of a transition as *sanity check*. An application of a transition is legitimate only if it complies with the requirements of its sanity check. The collection of sanity checks constitute a semantics for the generic STV machine.

Recall that a significant part of progress condition of a transition involves reducing a complexity measure. We first define exactly what this measure is and then continue with laying down the mathematical specification of the sanity checks for all transitions. To formulate the complexity measure, we define a lexicographic ordering on the set $\mathbb{N}^6$ which we denote by the symbol $\prec$ and impose it on non-final states of the generic machine.

Definition 2.3 *Let $\{s : \mathcal{S} \mid s \text{ not final}\}$ be the set of non-final machine states. We define a function* $\mathsf{Measure} : \mathcal{S} \to \mathbb{N}^6$ *as follows. We let* $\mathrm{Measure}\ (\mathrm{initial}(ba))$ *= (1,0,0,0,0,0). Suppose $bl = (l_1, l_2)$, for some lists l_1 and l_2, and for a given candidate c,* $\mathrm{concat}\ (p\ c) = l_c$ *where for a list l of lists,* $\mathsf{concat}\ l$ *is the concatenation of all elements of l. Then*

$$\mathrm{Measure}\ (\mathrm{state}\ (ba, t, p, bl, e, h)) = (0,\ |\mathcal{A}_{cand}|\ -|e|,\ |h|,\ \textstyle\sum_{d \in l_2} |l_d|,\ |l_1|,\ |ba|).$$

We naturally face two questions; One wonders why we choose these elements instead of some other for the measure. Also it is not obvious in any ways why we have imposed this particular ordering on the measure components. We answer both questions in more detail but defer it to Section 2.4 in order to avoid disrupting simple flow of the narrative. However, in the discussion that immediately follows, we shed some light on Definition 2.3.

One remark to note at this point is that each component of the measure is mainly targeting one of the generic machine transitions. Also the ordering of the components is carefully chosen to allow proving the termination theorem which we establish for the generic machine to hold for a broad spectrum of machine instantiations with various STV schemes. We next describe each component of the measure in English language and mention their connection with their intended machine transition. Later in Section 2.4 we explain about the ordering of the measure parts.

measure of initial states. As you see, the measure of each non-final state is a 6-tuple of natural numbers. The first element of the 6-tuple of an initial state is defined as 1 and the rest of the element of the 6-tuple are zero. Note that the first component of the co-domain of the measure function simply reduces the measure when moving from the initial state to any intermediate state.

measure of intermediate states. The first element of a given intermediate state is zero. This consequently allows an application of the **start** transition to reduce the measure once we move from an initial input state to an output intermediate state. We next describe what the rest of the components stand for.

- the second component essentially measures how many of the initially competing candidates are not elected at the current intermediate state. This component is essentially designed for reducing the measure upon an application of the **elect** transition. Every application of the **elect** transition decreases the second component of the measure.

- length of the list of continuing candidates at the current state comprises the third component of the measure. This component reduces whenever an application of the **eliminate** transition occurs.

- remember that the backlog $bl = (l_1, l_2)$ is a pair of lists where the second component l_2 of the pair represents the list of candidates who have been already eliminated and whose votes await distribution to the continuing candidates. Also recall that for a given candidate c, the pile of c, namely $p\ (c)$, at the current intermediate state is constituted of a list of ballot lists. We concatenate these ballots lists and consider the length of the resulting list as l_c. The fourth component of the measure is then defined as the sum of all l_c where c belongs to the list l_2. In other words, we add up the number of all of the ballots appearing in the pile of candidates whom have already been eliminated. This component of the measure decreases upon an application of the **transfer-removed** transition.

- the fifth component of the measure is the length of the first element of the backlog which consists of the list of elected candidates whose votes still wait for being transferred. The component reduces whenever a **transfer-elected** transition executes.

- the last component of the measure is the number of the ballots in the list of

uncounted ballots at this stage which decreases upon an execution of the **count** transition.

We subsequently present the sanity checks of every transition.

Start sanity check. The action which determines the formal ballots and thus initiates the process of counting is the start transition. It has to satisfy two criteria. The applicability criterion for the starting transition instructs that if the current state of the computation is an initial state of the form $\mathrm{initial}$ (*ba*) for some list of ballots *ba* cast to be counted, then there must exist an intermediate state of the form state (filter (*ba*),[nty],nas,([],[]),[],$\mathcal{A}_{cand}$) to which we can move by application of the start. Here the function filter decides which ballots are formal and excludes the informal ones from the list of ballots to be counted. The tally of each candidate is set to be zero by the function nty[3] as no votes are yet assigned to any candidate. Since no ballot is allocated to any candidate at this stage, the pile nas[4] assigns the empty list to each candidate. Moreover, because no one is yet elected or eliminated each component of the backlog consists of the empty list of candidates. Additionally, the list of already elected candidates is empty and every candidate participating in the election is currently a continuing candidate.

The progress condition for the start simply asserts that whenever this transition is applicable, the pre-state on which it operates must be an initial state and the post-state must be of an intermediate form where the list of uncounted may change (as a result of informal ballots).

Definition 2.4 (start sanity check) *A rule $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies the* start sanity check *if and only if the following two conditions hold:*

applicability. for all input, output $\in \mathcal{S}$ *and for all ba, if* input $= \mathrm{initial}$*(ba), then there exists* output $= \mathrm{state}(\mathrm{filter}$*(ba),*[$\mathrm{nty}$],$\mathrm{nas}$,*([], []),*[],$\mathcal{A}_{cand}$*) such that (*input,output*) $\in R$.*

progress. for all input *and* output *if (*input,output*)*$\in R$*, then for some ba, ba′, t, p, bl, e, and h,* input $= \mathrm{initial}$*(ba) and* output $= \mathrm{state}$*(ba′,t,p,bl,e,h).*

Count sanity check. To legally apply the count transition the applicability restriction dictates that the pre-state must be an intermediate state such as state (*ba*,*t*,*p*,*bl*,*e*,*h*) where the list *ba* is not empty so that there are ballots to attend to. If so, then there exists a post-state to which the machine transits. The progress condition specifies that this post-state is an intermediate one where the length of the list of uncounted ballots has decreased, meaning that at least some votes have been counted by taking this step, the tally and pile lists are updated in such a way that the third component of the measure remains the same as in the pre-state.

Definition 2.5 (count sanity check) *A transition $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies the sanity check for count if and only if the following criteria hold:*

[3] nty stands for null tally.

[4] nas is an abbreviation for null assignment

applicability. for all input $\in \mathcal{S}$*, for any ba, t, p, bl, e, and h, if* input $= \mathrm{state}$*(ba,t,p,bl,e,h) and* $ba \neq []$*, then there is an* output *state where* $(\mathsf{input}, \mathsf{output}) \in R$*.*

progress. for all input, output $\in \mathcal{S}$ *if* $(\mathsf{input}, \mathsf{output}) \in R$*, then there exist* ba_1*,* ba_2*,* t_0*, ts,* p_1*,* p_2*, bl, e, and h such that*

- input $= \mathrm{state}(ba_1, ts, p_1, bl, e, h)$, output $= \mathrm{state}(ba_2, t_0 :: ts, p_2, bl, e, h)$,
- $|ba_2| < |ba_1|$,
- $\sum_{d \in bl_2} |l_d| = \sum_{c \in bl_2} |l_c|$*, where* $l_d = \mathrm{concat}(p_1\ d)$ *and* $l_c = \mathrm{concat}(p_2\ c)$.

Elect sanity check. The action of electing a candidate is subject to the following constraints: in the pre-state input $= \mathrm{state}([],t,p,bl,e,h)$, where the list of uncounted ballots is empty, some continuing candidate must have reached quota, and there must be a vacant seat. If so, there must exist a post-state output where the length of the elected candidates in the pre-state is shorter than its counterpart in the post-state, there *may* be uncounted ballots to deal with next, and the piles and backlog for candidates may be updated. Mathematically, this takes the following form:

Definition 2.6 (elect sanity check) *A rule* $R \subseteq \mathcal{S} \times \mathcal{S}$ *satisfies the* elect sanity check *if and only if the following two conditions hold:*

applicability: for any state input $= \mathrm{state}([], t, p, bl, e, h)$ *and any continuing candidate* $c \in h$*, if* $t(c) \geq qu$ *(c has reached quota) and* $\mathsf{length}(e) < st$ *(there are vacancies), there exists a post-state* output *such that* $(\mathsf{input}, \mathsf{output}) \in R$*.*

progress: for any states input *and* output*, if* $(\mathsf{input}, \mathsf{output}) \in R$*, then* input *is of the form* $\mathrm{state}([], t, p, bl, e, h)$*,* output *is of the form* $\mathrm{state}(nba, nt, np, nbl, ne, nh)$ *and* $|e| < |ne|$*.*

Transfer-elected sanity check. The *transfer-elected* rule that describes the transfer of surplus votes of an elected candidate must satisfy two conditions. The applicability condition asserts that transfer-elected is applicable to any intermediate machine state input of the form $\mathrm{state}([],t,p,bl,e,h)$, where the list of uncounted ballots is empty, if there are vacancies to fill ($\mathrm{length}(e) < st$), there are surpluses awaiting transfer ($bl_1 \neq []$), no vote of an already eliminated candidate awaits transfer, and no continuing candidate has reached or exceeded the quota. Under these conditions, we stipulate the existence of a post-state output which is reachable from input via a transition labelled *transfer-elected*. The progress condition requires that any application of *transfer-elected* reduces the length of the backlog of the electedbl_1 while elected and continuing candidates remain unchanged. Mathematically, this takes the following form:

Definition 2.7 (transfer-elected sanity check) *A rule* $R \subseteq \mathcal{S} \times \mathcal{S}$ *satisfies the* transfer-elected sanity check *if and only if the following hold:*

applicability: for any state input $= \mathrm{state}([], t, p, bl, e, h)$ *that satisfies*

- $|e| < st$ *(there are still seats to fill),*
- $bl_1 \neq []$ *(there are surplus votes to be transferred),*
- $(bl_2 = []) \vee \exists\, hbl_2\ tbl_2.(\ bl_2 = hbl_2 :: tbl_2) \wedge (\mathrm{concat}(p\ hbl_2) = [])$, *i.e. no vote of an eliminated candidate awaits transfer,*
- $\forall c.\ (c \in h \rightarrow (t\ c < qu))$ *(no continuing candidate has reached the quota),*

there exists a post-state output *such that* $(\mathsf{input}, \mathsf{output}) \in R$.

progress: for any machine states input *and* output, *if* input R output *then* input *is of the form* $\mathrm{state}([],t,p,bl,e,h)$, output *is of the form* $\mathrm{state}(nba,t,np,nbl,e,h)$ *and* $(|nbl_1| < |bl_1|)$, *i.e. the backlog of the elected is reduced.*

Eliminate sanity check. Eliminating a candidate applies only when the current state of computation is an intermediate state of the form $\mathrm{state}([],t,p,([],[]),e,h)$ where there are no ballots to count and no surplus awaits transfer, no votes from an already eliminated candidate awaits distribution, and no continuing candidate has reached or exceeded the quota but there are still vacancies to fill. The progress constraint forces the post-state reached by taking this action to be another intermediate state where some candidate is no longer continuing in the process of counting and the list of uncounted ballots changes to *nba*, which may not be empty.

Definition 2.8 (elimination sanity check) *A transition $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies the sanity check for elimination only if and when the following hold:*

applicability. for any input $\in \mathcal{S}$ *and for all t, p, bl_2, e, and h, if*

- $\mathsf{input} = \mathrm{state}([], t, p, ([], bl_2), e, h)$,
- $|e| + |h| > st$ *(the number of elected and continuing is still more than vacancies), and*
- $(\forall c, (c \in h) \rightarrow (\mathsf{head}(t)\ c) < qu)$, *i.e. no one is electable,*
- $(bl_2 = []) \vee \exists\, hbl_2\ tbl_2.(\ bl_2 = hbl_2 :: tbl_2) \wedge (\mathrm{concat}(p\ hbl_2) = [])$, *i.e. no vote of an eliminated candidate awaits transfer,*

then there is an output *state where* $(\mathsf{input}, \mathsf{output}) \in R$.

progress. for any input *and* output, *if* $(\mathsf{input}, \mathsf{output}) \in R$ *then there exist nba, t, p, np, e, h, nh, bl_2, nbl_2 such that* input $= \mathrm{state}([], t, p, ([], bl_2), e, h)$, $|nh| < |h|$, *and* output $= \mathrm{state}(nba, t, np, ([], nbl_2), e, nh)$

Transfer-removed sanity check. This sanity check specifies when and how to deal with distribution of the votes of an eliminated candidate. In a nutshell, it asserts that transfer-removed transition applies only to an intermediate state of the computation state $([],t,p,(bl_1, bl_2),e,h)$ where there are vacancies to fill, no candidate exceeds or reaches to the quota at this stage, there are votes from an already eliminated candidate awaiting distribution. The progress condition forces the post-state to also be an

intermediate state where the only change in the complexity measure is decrease in the third component.

Definition 2.9 (transfer-removed sanity check) *A transition $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies the sanity check for transferring votes of removed candidates if and only if the following conditions hold:*

applicability. for all $\mathsf{input} \in \mathcal{S}$, t, p, bl_1, bl_2, e, *and* h, *if*

- $\mathsf{input} = \mathrm{state}([], t, p, (bl_1, bl_2), e, h)$
- $(\exists\ hbl_2\ tbl_2.\ (\ bl_2 = hbl_2 :: tbl_2) \wedge (\mathrm{concat}(p\ hbl_2) \neq []))$, *i.e. there are votes from an eliminated candidate awaiting distribution,*
- $|e| < st$, *and* $(\forall c, c \in h \rightarrow \mathsf{head}(t)\ c < qu)$

then there is an output *state such that* $(\mathsf{input}, \mathsf{output}) \in R$.

progress. for any input *and* output, *if* $(\mathsf{input}, \mathsf{output}) \in R$ *then there are* nba, t, p, np, bl, nbl, e, *and* h *such that*

- $\mathsf{input} = \mathrm{state}([], t, p, bl, e, h)$ *and* $\mathsf{output} = \mathrm{state}(nba, t, np, nbl, e, h)$
- $|nbl_1| = |bl_1|$, *i.e. no votes from any already elected candidate is distributed by this transition.*
- $(\sum_{c \in bl_2} |l_c| < \sum_{d \in bl_2} |l_d|$, *where* $l_d = \mathrm{concat}(p_1\ d)$ *and* $l_c = \mathrm{concat}(p_2\ c)$, *i.e. at least some votes of the eliminated candidate at the head position of* bl_2 *are distributed by this transition.*

The sanity checks for elected-win and hopeful-win transitions determine what it means for a transition to legitimately be an instance of a finishing action where a counting process terminates at a final state.

Definition 2.10 (elect-win sanity check) *A transition $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies sanity checks for elect-win if and only if the following hold:*

applicability. for all input *and any* ba, t, p, bl, e, h, *if* $\mathsf{input} = \mathrm{state}(ba, t, p, bl, e, h)$ *and* $\mathsf{length}(e) = st$ *then there is an* output *state where* $(\mathsf{input}, \mathsf{output}) \in R$.

progress. for all input *and* output, *if* $(\mathsf{input}, \mathsf{output}) \in R$ *then for some* ba, t, p, bl, e, *and* h, *we have* $\mathsf{input} = \mathrm{state}(ba, t, p, bl, e, h)$ *and* $\mathsf{output} = \mathrm{winners}(e)$.

Definition 2.11 (hopeful-win sanity check) *A transition $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfies sanity checks for hopeful-win if and only if the following hold:*

applicability. for all input *and any* ba, t, p, bl, e, h, *if* $\mathsf{input} = \mathrm{state}(ba, t, p, bl, e, h)$ *and* $\mathsf{length}(e) + \mathsf{length}(h) \leq st$ *then there is an* output *state where* $(\mathsf{input}, \mathsf{output}) \in R$.

progress. for all input *and* output, *if* $(\mathsf{input}, \mathsf{output}) \in R$ *then for some* ba, t, p, bl, e, *and* h, *we have* $\mathsf{input} = \mathrm{state}(ba, t, p, bl, e, h)$ *and* $\mathsf{output} = \mathrm{winners}(e + +h)$.

The sanity checks constrain the computation that may happen on a given input state if the corresponding rule is applied. A set of rules, each of which satisfies the corresponding sanity check, can therefore be seen as a small-step semantics for STV counting. We capture this mathematically as a generic machine.

Definition 2.12 (The generic STV machine) *Let $\mathcal{S}$ and $\mathcal{T}$ be the sets of STV states (Definition 2.1) and transition labels (Definition 2.2), respectively. The* generic STV machine *is $M = \langle \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ where S_t is the sanity check condition for transition $t \in \mathcal{T}$. An* instance *of M is a tuple $I = \langle \mathcal{T}, (R_t)_{t \in \mathcal{T}} \rangle$, where for each $t \in \mathcal{T}$, $R_t \subseteq \mathcal{S} \times \mathcal{S}$ is a rule that satisfies the sanity check condition S_t.*

2.3 Properties of the Machine

We prove three general properties for the generic STV machine in Definition 2.12. The first property ensures that any legitimate application of a transition reduces the complexity measure. The second theorem ascertains that at any non-final state of the machine, at least one transition is indeed applicable. The two theorems together guarantee termination of the machine upon each execution on any input value.

Theorem 2.1 *Any transition R corresponding to a machine transition in $\mathcal{T}$ that satisfies the corresponding sanity check reduces the complexity measure.*

One specific "sanity check", in fact the one that inspired the very term, is the ability to always "progress" the count. That is, one rule is applicable at every state, so that the count will always progress, and there are no "dead ends", i.e. states of the count that are not final but to which no rule is applicable. The key insight is that if the sanity check conditions (and hence the applicability conditions) are satisfied, we can always progress the count by applying a rule. In a nutshell, the following steps are repeated in order:

- the start rule applies (only) at initial states;
- cease scrutiny if all vacancies are filled by elected candidates ;
- cease scrutiny if all vacancies are filled by elected and continuing candidates;
- uncounted ballots shall be counted;
- candidates that reach or exceed the quota shall be elected;
- the surplus of elected candidates shall be transferred;
- the ballots of eliminated candidates shall be transferred;
- the weakest candidate shall be eliminated.

We realise this order of rule applications in the proof of the rule applicability theorem. We draw upon the local rule applicability property, present in the sanity checks satisfied by the generic STV model, to guide the theorem prover Coq to the proof, according to the pseudo-algorithm above. Hence we formally verify the expectation of STV protocols on the invariant order of transition applications.

Theorem 2.2 (Rule Applicability) *Let* $I = \langle \mathcal{T}, (R_t)_{t \in \mathcal{T}} \rangle$ *be an instance of the generic STV machine. For every non-final state* input *, there is a transition label* $t \in \mathcal{T}$ *and a new state* output *such that* $\mathsf{input}\ R_t\ \mathsf{output}$.

Theorem 2.1 shows that every applicable transition R_t, for $t \in \mathcal{T}$, reduces the complexity measure. Theorem 2.2 shows that for any non-final machine state, a transition from the set $\mathcal{T}$ is applicable. Jointly, they give a termination property that, in the terminology of programming semantics, asserts that every execution of the generic STV model has a meaning which is the sequence of computations taken to eventually terminate, and that each execution produces an output which is the value of that execution.

Theorem 2.3 (Termination) *Each execution of every instance of the generic STV machine on any initial state* input *terminates at a final state* output*, and constructs the sequence of computations (namely a certificate) taken from* input *to reach to* output.

2.4 More Detail on the Machine Semantics

In Section 2.2.1 we mentioned that each component of the complexity measure in Definition 2.3 is associated with a particular generic machine transition. Here we elaborate more on how each part of the measure realises our intention for reducing the overall complexity upon executing a transition. Also we explain about the ordering of the component and how it relates to broadening the usability of the framework.

The reader recalls from our analysis of various STV schemes in Section 2.1 that we have obtained the semantics of the machine as in Section 2.2.1 by analysing different concrete STV algorithms and then identifying patterns of similarity among them which constitutes the generic STV. Moreover, we remind the reader that each instance of a real STV is then realised in our system through instantiation of the general structure referred to as generic STV. In light of this, we shall note that one identifies a measure of complexity with less many components and simpler formulation that can work for one instance of STV but not the other. However, with the right generalisation, we can choose a measure that functions for a wide range of STV algorithms.

To elaborate more tangibly, assume st_1=state $(ba_1, t1, p1, bl_1, e_1, h_1)$ is an intermediate state of the machine (which corresponds to a possible computation stage in an instance of tallying votes of an election). Also suppose by an application of a machine transition $\mathcal{R}$ the machine moves to the next state st_2=state $(ba_2, t_2, p_2, bl_2, e_2, h_2)$. Based on Definition 2.3, the followings are the complexity of two states:

$$\text{Measure}(st_1) = (0,\ |\mathcal{A}_{cand}|\ -|e_1|,\ |h_1|,\ \textstyle\sum_{d\in l_2}|l_d|,\ |l_1|,\ |ba_1|)$$
$$\text{Measure}(st_2) = (0,\ |\mathcal{A}_{cand}|\ -|e_2|,\ |h_2|,\ \textstyle\sum_{d\in l'_2}|l_d|,\ |l'_1|,\ |ba_2|).$$

We consider a few scenarios to describe the choice of each measure component by discussing how the machine makes the transition to st_2 from st_1 depending on the specific STV algorithm used for instantiating $\mathcal{R}$ with.

Elect. Suppose we are in a situation where the generic progress constraints of the elect sanity check are satisfied at state st_1. Therefore, the generic machine and any instantiation of it by an STV scheme must execute the elect transition. Also we know that as a result of this execution, the complexity measure of the next state must reduce compared to st_1. Now let us first consider the scenario when $\mathcal{R}$ is instantiated with the elect action of the ACT STV scheme mentioned in Section 2.1.1. As you see in the algorithm description, once the elect action comes into effect, the length of the elected candidates in the post-state increases because some candidates who have reached the quota are elected and therefore added to the list of elected candidates in the post-state. Consequently, the second component of the measure decreases when moving from st_1 to st_2 because $|\mathcal{A}_{cand}|\ -|e_2| < |\mathcal{A}_{cand}|\ -|e_1|$. As the measure imposed on the set of machine states is a lexicographic ordering, we therefore have $\text{Measure}(st_2) \prec \text{Measure}(st_1)$.

When analysing the ACT STV protocol, we understand that with a simpler measure, every thing else remaining the same, we can have a framework which accommodates STV algorithms whose elect action is the same as ACT STV. More generally, the second component of the measure can be removed without restricting the framework to allow formalisation and verification of the elect transition of an STV algorithm whose elect action is similar to ACT's. The reason is that, the third component of the measure decreases upon an application of an instantiation of the generic elect transition with an elect action such as ACT's and the lexicographic ordering works without any problem.

Nonetheless, such a framework loses its usability for accommodating an STV algorithm whose elect transition behaves like CADE STV. This latter STV uses a radically different mechanism when electing candidates. In short, once a candidate is elected, CADE STV insists on *restarting* the election. In other words, the process of counting votes begins again with the difference that the candidate who has already been elected is no longer running in the counting as if he/she never participated in the election in the first place, and the number of vacancies left decreases by one. The astute reader quickly grasps that removing the second component of the measure in Definition 2.3 causes problems because due to the restarting effect, the length of the continuing candidates in the post-state may increase compared to its counterpart in the pre-state. As a consequence, that measure would not work for CADE STV or any other STV which relies on such a restarting mechanism. This observation leads us to include the second component as in Definition 2.3.

Transfer-removed. We now take into account a scenario where at state st_1 given

above, the progress constraints of the transition transfer-removed are satisfied. Then the generic machine executes transfer-removed branch and moves to the state st_2 where the measure of the post-state must be less than the measure of the pre-state st_1, i.e. $\mathrm{Measure}(st_2) \prec \mathrm{Measure}(st_1)$. As we explained in Section 2.2.1, the fourth component of the measure deals with reducing the complexity when applying transfer-removed transition. This part of the measure has a similar purpose as the second component, which we unfold subsequently. To this end, suppose the transition $\mathcal{R}$ used for moving from st_1 to st_2 is instantiated with the transfer-removed action of the ACT STV. This algorithm transfers the votes of an eliminated candidate in a single application of transfer-removed. As a result, the fourth component of the measure could simply be defined as the length of the second part of the backlog instead of the complicated sum that appears in Definition 2.3. The reason for such a simple alternative being possible is that as far as STV algorithms which resemble ACT STV are concerned their way of distributing votes of eliminated candidates, by every application of an instantiated version of transfer-removed generic transition with the one used in those specific STV schemes, the length of the second component of the backlog reduces. Consequently, for a state st=$\mathrm{state}\ (ba, t, p, bl, e, h)$ we could have defined the measure of the intermediate states to be $\mathrm{Measure'}\ (\mathrm{state}\ (ba, t, p, bl, e, h))$ = $(0, |\mathcal{A}_{cand}| - |e|, |h|, |l_2|, |l_1|, |ba|)$ where $bl = (l_1, l_2)$.

However, such a measure as in the previous paragraph would not work for a case where $\mathcal{R}$ is instantiated with the Victoria STV used in the upper house elections of Australia. In a nutshell, the Victoria STV transfers votes of an eliminated candidate step by step based on the magnitude of the fractional values that they carry. Therefore, votes of an eliminated candidate are piled into distinct groups where every vote in a pile has the same fractional value and then in order of the magnitude of these fractional values, transfer-removed applies in some steps until there is no group of votes left to transfer. In counting votes under this voting scheme, a situation is possible where none of the components of $\mathrm{Measure'}$ given in the preceding paragraph decreases. For example, assume candidate c is the one who has been eliminated and whose votes are to be transferred by executing transfer-removed. Imagine that there are some votes associated with c which have a fractional value distinct from some other votes of his/hers, where these votes have been recorded in more than one parcel of votes. Consequently, an application of transfer-removed can apply for distributing the first group of votes of c while no component of $\mathrm{Measure'}$ decrease. In fact as long as there is more than one group of such votes of c left to transfer, none of the $\mathrm{Measure'}$ component reduce. Hence choosing $\mathrm{Measure'}$ would limit the framework in that Victoria STV can no longer be formalised. To accommodate the Victoria STV and any other STV scheme which uses a stepwise mechanism of distributing votes, we formulate the fourth component as in Definition 2.3.

In the above discussion, we elaborated on why two of the measure components are defined as in Definition 2.3. In the following we explain why the measure components appear in this order. The ordering of the measure components strongly relates to the generic pattern existing in every sequence of machine execution which forms

the structure of the applicability proof explained in Section 2.3. To elaborate more, we described in Subsection 2.1.3 that the progress conditions of the generic STV impose a general pattern of transition executions. This pattern determines on which transition to apply under what circumstances. Consequently, there is an ordering for executing transitions. The ordering restricts the manner according to which the generic machine and any of its instantiations sequentially execute counting actions to compute winners of an election. As a result of the above ordering in machine execution, we must guarantee that any sequential combination of the transitions that stand for a correct machine execution consistently reduces the complexity measure from the initial machine state to a final state where the instance of computation terminates.

The pattern of execution mentioned in the previous paragraph can be reconstructed from the applicability constraints of the sanity checks given in Section 2.2.1. Here we explain only a specific part of the reconstruction so that the reader is guided on how to obtain the full pattern on their own. Recall that in Definition 2.3 the second component of the measure is associated with reducing the measure when applying the elect transition, the third component deals with the elimination and the fourth one takes care of transferring votes of an eliminated candidate. As the applicability of the elimination and transfer-removed shows, neither of the transitions apply if and when there is some candidate who has reached or exceeded the quota. Therefore whenever in an intermediate state of computation the machine reaches a situation where someone is electable, the elect rule has precedence over elimination and transferring votes of an eliminated candidate. On the other hand, for the transfer-removed to apply, there has to exist some votes in the pile of an already eliminated candidate to deal with. Therefore, in situations where the backlog of the eliminated candidates is empty, the elimination transition has to apply ahead of the transfer-removed so that there are votes to distribute as a consequence of having removed someone from the tallying. However, once some candidate has been eliminated and the pile of votes associated with the eliminated candidate have all not yet been transferred, the elimination rule cannot apply prior to transfer-removed until there are no such votes from a removed candidate to distribute.

Observing such existing execution pattern in STV algorithms and generalising it to apply for the generic STV, we need to use a measure that handles these possibly occurring valid execution sequences in an instance of computation. The current ordering of the measure components satisfies our intention. Now if one reconstructs the execution pattern carefully based on the sanity checks, they instantly realise that the ordering of measure components allows realising this phenomena and accommodates formalisation of a wide range of STV schemes. If one replaced the second and third component of the measure, for CADE STV as an example, there can be legitimate computation that can not be realised in the framework any longer because the measure would not decrease in some applications of elect transition. Also changing the order of the third with the fourth component would cause issues because in applications of the elimination it might indeed increase the measure of the post-state compared to its pre-state reached by applying elimination. A consequence of this

latter replacement is losing formalisation of algorithms such as the Victoria STV.

Chapter 3

Formalisation and Verification of the Model in Coq

In the previous chapter, we analysed the family of STV algorithms and conceptualised a core notion of STV which we named generic STV. We then informally formulated it as an abstract finite state machine and mathematically expressed its components. In this chapter, we lay down a formal representation of the machine carried out in the environment of the theorem Coq. We first explain the macro level design decisions made for implementing the machine and the framework as a whole and then continue elaborating on technicalities of the formalisation with constant attention to describing why we performed this way rather than otherwise.

We find spurious tedious details of the formalisation exhaustive and confusing to most readers. Hence we selectively discuss only some parts of the work so that the heart of our message gets across faithfully to the quality of the work performed and the chapter offers a readable narrative to enjoy. We however encourage the reader to visit the Github repository[1] for further information on the formalisation.

3.1 The Architecture of the Coq Framework

Design precedes implementation. Standards exist to guide us on *what* features are desirable for a clever design. Modularity in design and implementation [119] is a standard in which we are interested. Recall that we catalogued objectives for our work to capture one of which is practicality. We wish to produce a framework that enjoys usability so that it can be adapted and extended to the specific needs of users easily. Automating proofs facilitates users with significant relief from dealing with sophisticated formalisation and verification crafts. Also, a thoughtful design keeps an eye on the future so that it can adapt itself to changes that come about as one wishes to thrive on the existing infrastructure rather than building on top of ashes of the old.

The same standards also hint us on *how* and based on what criteria to create the modules. When engineering a framework, it is highly desirable to separate bits and

[1] https://github.com/MiladKetabGhale/Modular-STVCalculi

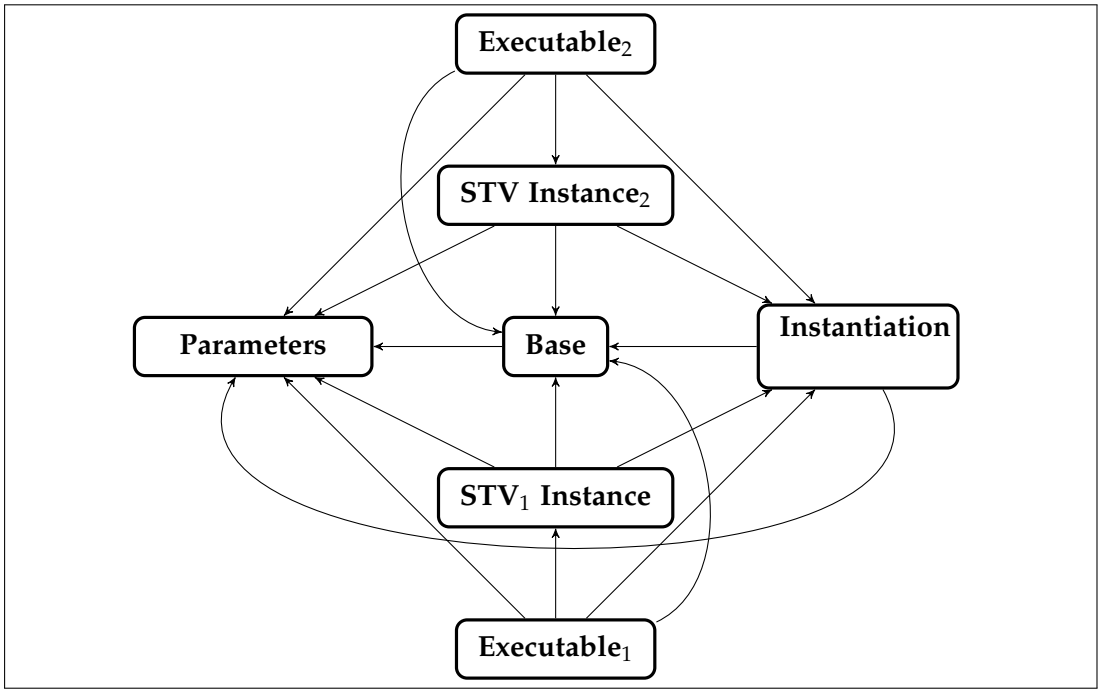

Figure 3.1: Architecture of the Coq Framework

pieces of the encoding into different components in such a way that each component encapsulates parts that are strongly interconnected [119]. It is also an advantage to design each component in a manner that its development minimises the amount of workload required and maximises its functionality and usability to the whole system. All of the above encloses and restricts searching for an error in a considerable smaller chunk of code. It also allows other components whose functionality is independent of a failed one to still continue operating.

Here we refer to the above two paragraphs as the design standards. In light of them, we break implementation of the framework into several purposeful modules schematically illustrated in Figure 3.1. In the figure, the direction of arrows stands for a dependency of the module from which the arrow starts on the one that it points to. We next briefly discuss the content of each module and then explain why we choose this structure.

```
Parameter st : nat.
Parameter quota : Q.
Parameter cand : Type.
Parameter ValidBallot : list cand -> bool.
```

Figure 3.2: Parameters of the Machine

Parameters. It includes the parametric specification of the vacancies, quota, candidates, and the notion of validity of a ballot encoded as Figure 3.2 illustrates. Remem-

```
Parameter cand_all: list cand.
Parameter cand_nodup: NoDup cand_all.
Parameter cand_finite: forall c, In c cand_all.
Parameter cand_eq_dec: forall c d:cand, {c=d} + {c<>d}.
```

Figure 3.3: Constraints on Candidates

ber that the exact formulation of the quota and the notion of valid ballot varies from one STV algorithm to another. Needless to mention, in different elections, there are distinct candidates competing for a varying number of seats. On the other hand, designing an abstract machine which depends on merely specific values for these data parts is insensible simply because it limits the applicability of the machine concept. Therefore we choose to parametrically specify them so that they are generic enough to use for a generic specification of the machine but still can be given specific concrete values through later instantiations so that we can use the instantiated machine for computing real elections.

There are Coq declarations depicted in Figure 3.3 which further specify when candidates and an initial list of competing candidates called cand_all are legitimately acceptable. In every ballot of an election, there is no duplication of any candidate's name (cand_nodup) and candidates are easily distinguished from one another on a ballot by their names (cand_eq_dec). Also, there are only finitely many individuals as candidates and the list of competing candidates given to voters is comprehensive in the sense that it includes all of the individual participating as candidates (cand_finite). Our reasoning in Coq about the machine relies on these simple conditions to express and prove properties about STV schemes.

Instantiation. The entities encoded in the parameters modules are stored as a record. For actual computation, we need to instantiate the formally specified machine with concrete values for each record field. By instantiation, one obtains a verified version of the frame base for the specific values of parameters defined. This simply happens by specifying in the instantiation module who are the candidates, what is the number of the vacancies, the quota, and the exact value for the Validballot function with which we filter formal ballots. As we have automated proof obligations for showing that instantiated values for candidates has no duplication or that the value given for cand_all indeed contains all of the names appearing under the value instantiated for the cand type, users only need to include name of the lemmas in their instantiation for the parameters module.

At this point, we can explain more about some of the constraints imposed on the type cand earlier in the parameters module. As it was mentioned earlier in Section 1.2, we intend to extract and compute with Haskell programs. Each of these programs has an extracted type corresponding to the Coq type cand. If we allowed candidates' names to have duplication, we would get error messages back from the Haskell compiler that name of some constructors is the same. This is another motivation for restricting the type of candidates with such conditions. Figure 3.4 illustrates an example of instantiation where three candidates named Alice, Bob and Charlie

```
Inductive Cand := Alice | Bob | Charlie.
Module Instantiate.
Definition cand := Cand.
Definition cand_all := [Alice;Bob;Charlie].
Definition st := (1)%nat.
Definition quota := (8 # 3)%Q.
Definition ValidBallot (l: list cand) :=
   match l with
     [] => false
   |_ => if nodup_elem l then true else false
   end.
```

Figure 3.4: Example of Instantiation of parameters

are competing in a fictional election for one vacancy and the quota is the fractional value 8/3. Note that only the ballots which are non-empty and duplicate free are considered valid (ValidBallot).

Base. The module base stands at the centre of our structure. It consists of formalisation and verification of the STV machine and also a specification of auxiliary functions used for defining and verifying components of the machine. Some auxiliary assertions that also appear usually in the formalisation of numerous particular STV algorithms are included in this module so that the base functions as a comprehensive library as well.

STV Instance. The structure has a type of modules schematically named STV instances. Instantiation of the STV machine happens in this module in two steps. The first one is to define transition labels of the machine in accordance with the textual specification of the particular STV algorithm in which one is interested. The second step then is to discharge the formally defined small-step semantics of the machine. By proving that transition labels satisfy the semantics, one can then instantiate the machine to obtain a verified formalisation of the STV instance.

Executable. Once an STV instance is verified by discharging the semantics, one can extract a Haskell executable program which is a proven correct implementation for the instance. As parameters have already been specified and the transition labels are also instantiated, the extracted program can compute an election upon each execution on a list of input ballots.

The remainder of this chapter focuses on discussing formalisation and verification of the machine included in the frame base module. In the subsequent chapter, we elaborate on details of auxiliary assertions and the STV instance modules by demonstrating how two STV algorithms used in the upper house elections of Victoria state of Australia and the one used in the lower elections of the ACT state are formally verified.

3.2 Inductive Type of the Machine States

Figure 3.5 illustrates the inductive type whose values formally represent the machine states. There are three constructors each of which is used to construct a different

```
Inductive Machine_States :=
 initial:
    list ballot -> Machine_States
 |state:                                                  (** intermediate states **)
   list ballot                                            (* uncounted votes *)
   * list (X.cand -> Q)                                   (* tally *)
   * (X.cand -> list (list ballot))                       (* pile of ballots *)
   * ((list X.cand) * (list X.cand))                      (* backlog *)
   * {elected: list X.cand | length  elected <= X.st}(* elected candidates *)
   * {hopeful: list X.cand | NoDup hopeful}              (* continuing candidates *)
     -> Machine_States
 |winners:                                                (** final state **)
     list X.cand -> Machine_States.                        (* election winners *)
```

Figure 3.5: Inductive Type for Machine States

group of machine states. The rationale behind initial and final constructors as defined in the figure are straightforward. However, tailoring the intermediate states has an interesting story to tell which we shall unravel shortly afterwards.

Uncounted Ballots. As explained in Subsection 2.1.2 and Section 2.2 a b ballot is a pair (l, q) where l is the list of candidates ranked according to a voter's preferences and q is the fractional value that the ballot b carries. Note that we also explained in Subsection 2.1.2 that there are conditions which determine if and when a ballot is valid. To our best knowledge, STV algorithms agree on two of such conditions, namely that l must be not empty so that the vote is not null and there must be no duplication in the preferences expressed. We therefore decide to declare a ballot to be a value of a pair type whose first component is a Sigma type and the second one is the type of rational numbers in Coq as shown in Figure 3.6. As you see, the second part of the Sigma type demands the first part to be non-empty and duplicate free. This restriction obliges us to demonstrate when performing proofs that our operations do not deviate from these legal conditions so that our computation handle ballots correctly under any circumstances in regard to their validity.

Tally. To formalise some tie-breaking methods used in some STV schemes, we encode tallies into a chronological list so we can trace the amount of votes which each candidate received in previous rounds. This allows us to facilitate one popular tie breaking procedure. In this method, whenever two or more candidates have the least votes, we go backwards stepwise, if need be, to previous states of the machine which we have computed in the same execution until we reach a state where one candidate has fewer votes than the tied candidates. Then we update the current state of the counting by eliminating this candidate. We refer the reader to Subsection 8.1.1 for further discussion on tie-breaking with other methods.

```
Definition ballot :=  ({v : list X.cand | (NoDup v) /\ ( [] <> v)} * Q).
```

Figure 3.6: Type of Ballots

Pile. Some STV schemes, such as lower house ACT and Tasmania STV, employ a notion called last parcel and transfer only ballots included in this parcel according to next preferences. Moreover, they compute the fractional transfer value based on the length of the last parcel. In short, the last parcel of a candidate is the set of votes they received which made them reach or exceed the quota to become elected. As a result, we choose to formalise the pile function to assign a list itself containing some lists of ballots. Every list of ballots as such contains the ballots received by a candidate after each round of application of the count rule. Therefore, we come to identify which exact set of ballots comprise the last parcel of any elected candidate. Consequently, we are able to tailor both the generic transfer and elect rule and instantiations of them in such a way to modularly formalise several STV schemes which use the last parcel effect.

We have another motivation as well for designing codomains of the piles in this way. The Victoria state STV distributes votes of an eliminated candidate step by step, rather than all at once, in order of the magnitude of the fractional value of ballots assigned to the eliminated candidate. It regroups ballots according to their fractional value and distributes each group at a time starting from the ones whose fractional value is the biggest. Therefore, we need to design the codomain of piles to be of type list of list of ballots so that we can place them in separate chunks to manage effects such as Victoria STV elimination style.

The textual specification of STV algorithms and their mechanism of handling counting requires to know the elected candidates, those continuing, and the ones whose votes await transfer because they either have been elected or eliminated. Hence we include data components in our inductive definition for machine states that represents them. However, one may wonder about choosing lists as the underlying data structure instead of, say, sets to realise these parts. First, we note that our formalisation does not make essential use of lists as part of the structure in the sense that no matter how ballots are listed, the winners are not affected by the list representation. Second, theorem provers such as Coq have built-in verified functions and libraries that support constructs for data structures built on top lists. As reducing the workload in constructing, maintaining and extending the framework easily is on our agenda, we therefore avoid inventing the wheel again. Third, the functions in Coq libraries for the list data type are all verified. Therefore we are confident that using them does no harm to the quality of verification and trustworthiness of the framework desired. In light of this confidence and utilisation, we try to rely on Coq libraries for the formalisation as much as they have to offer.

```
Inductive sigT (A:Type) (P:A -> Type) : Type :=
    existT : forall x:A, P x -> sigT P.
```

Figure 3.7: Declaration of Sigma Types

3.3 The Formal Semantics

Recall from the Section 2.2.1 that the semantics of each transition label is comprised of the applicability conditions. These conditions dictate when applying a transition is legal. There are also progress conditions which inform us of the consequences of taking that transition. As discussed, the progress constraints are stipulated by a complexity measure defined according to Definition 2.3. To formally pin down the measure, we take the following steps.

- **Step 1.** we use the notion of dependent pairs constructed by Sigma types of Coq to impose a dependent product construct on the set $\mathbb{N}^6$.
- **Step 2.** then using the well-founded library of Coq, we define a lexicographic ordering on the dependent construct $\mathbb{N}^5$ given in step (1).
- **Step 3.** Finally, we impose a lexicographic ordering on the set of non-final machine states as in Definition 2.3 by drawing on the auxiliary assertions above and the order on dependent $\mathbb{N}^6$.

3.3.1 Complexity Measure on the States

We find explanations based on set theoretic notions simpler for the reader. Therefore to explain what dependent types are and how lexicographic ordering on a dependent type is defined, we invoke a set theoretic terminology. Assume a set A is given. One sometimes wishes to distinguish elements of A based on various properties such as P from one another and place them in a new set. In set theory, this is taken care of by the axiom of comprehension [74]. In type theory, on the other hand, the notion of dependent types enables one to introduce such constructs. It happens by using the type constructors known as Sigma types. A Sigma type constructor accepts a type A and a property P and creates a new type symbolically represented by *sigT A P*.

```
Variable A : Type.
Variable B : A -> Type.
Variable leA : A -> A -> Prop.
Variable leB : forall x:A, B x -> B x -> Prop.

Inductive lexprod : sigT B -> sigT B -> Prop :=
  | left_lex :
    forall (x x':A) (y:B x) (y':B x'),
      leA x x' -> lexprod (existT B x y) (existT B x' y')
  | right_lex :
    forall (x:A) (y y':B x),
      leB x y y' -> lexprod (existT B x y) (existT B x y').
```

Figure 3.8: Dependent lexicographic Ordering

```
Definition DependentNat_Prod2 := sigT (A:= nat) (fun a => nat).
Definition DependentNat_Prod3 := sigT (A:= nat) (fun a => DependentNat_Prod2).
Definition DependentNat_Prod4 := sigT (A:= nat) (fun a => DependentNat_Prod3).
Definition DependentNat_Prod5 := sigT (A:= nat) (fun a => DependentNat_Prod4).
Definition DependentNat_Prod6 := sigT (A:= nat) (fun a => DependentNat_Prod5).
```

Figure 3.9: Dependent pairs on $\mathbb{N}^6$

```
Definition Make_DependentNat2  :
  nat * nat -> DependentNat_Prod2.
 intros (p,q). exists p. exact q.
Defined.

Definition Make_DependentNat3 :
  nat * (nat * nat) -> DependentNat_Prod3.
 intros (n, p_q). exists n. exact (Make_DependentNat2 p_q).
Defined.

Definition Make_DependentNat4 :
  nat * (nat * (nat * nat)) -> DependentNat_Prod4.
 intros (m, n_p_q). exists m. exact (Make_DependentNat3 n_p_q).
Defined.

Definition Make_DependentNat5 :
  nat * (nat * (nat * (nat * nat))) -> DependentNat_Prod5.
 intros (m,n_p_q_r). exists m. exact (Make_DependentNat4 n_p_q_r).
Defined.

Definition Make_DependentNat6 :
   nat * (nat * (nat * (nat * (nat * nat)))) -> DependentNat_Prod6.
 intros (m, n_p_q_r_s). exists m. exact (Make_DependentNat5 n_p_q_r_s).
Defined.
```

Figure 3.10: Imposing Dependent construct on $\mathbb{N}^6$

Figure 3.7 shows how Sigma types are constructed in Coq. A value of a Sigma type is a dependent pair $(x, \phi(x))$ where x belongs to the base set of the Sigma type on which P is defined and $\phi(x)$ is evidence inhabiting the type of $P\ x$. One can think of a Sigma type, when P's codomain is of type Type, as a family of sets indexed by A using P.

Assume $\mathcal{B} = \{B_x\}_{x \in A}$ is a family of sets and leB is a family of relations defined on $\mathcal{B}$ both indexed in A. Then a lexicographic ordering on $\mathcal{B}$ is defined in Coq according to Figure 3.8. There are exactly two ways to create a lexicographic order on a dependent pair, namely either using the constructor left_lex or right_lex. The former asserts that for a given pair (x, y) and (x', y') is x is less than x in terms of the relation leA then the pair (x, y) is lexicographically less than (x', y'). The latter constructor asserts the same but for the second component of a pair.

Now we are in a position to understandably describe how we achieve step (1) above. Since we rely on the Coq library for specifying lexicographic ordering, we have to first impose a dependent structure on $\mathbb{N}^6$.

Figure 3.9 illustrates how we proceed stepwise to make a dependent pair construction isomorphic to the type $\mathbb{N}^6$. First, we define a dependent structure which is

isomorphic to the type $\mathbb{N}^2$. Then using this construction, we introduce a similar one for $\mathbb{N}^3$ and continue in this manner to eventually obtain a dependently constructed type corresponding to $\mathbb{N}^6$. We then establish mappings (Figure 3.10) between each of the dependently constructed DependentNat_Prodi for $i \in \{1, \cdots, 6\}$ and the types $\mathbb{N}^i$ so that the isomorphism stands out.

Figure 3.11 demonstrates how the lexicographic order LexOrdNat is defined on DependentNat_Prod5. In a nutshell, given two dependent pairs $\mathcal{X}$ and $\mathcal{Y}$ in DependentNat_Prod5, the relation LexOrdNat checks if $\mathcal{X}$ is less than or equal to $\mathcal{Y}$ by comparing each components of them using the relation Peano.lt defined in Coq for comparison of two natural numbers.

We prove well-foundedness of the relation LexOrdNat by the lemma wf_LexOrdNat. Then we finally establish the lexicographic measure intended for $\mathbb{N}^5$ (Figure 3.13) and prove its well-foundedness using the lemma wf_LexOrdNat to complete the step (2) above.

```
Definition LexOrdNat_Aux1 :
DependentNat_Prod2 -> DependentNat_Prod2 -> Prop :=
(lexprod nat (fun a => nat) Peano.lt (fun a:nat =>Peano.lt)).

Definition LexOrdNat_Aux2 :
DependentNat_Prod3 -> DependentNat_Prod3 -> Prop :=
(lexprod nat (fun a => DependentNat_Prod2) Peano.lt (fun a:nat =>LexOrdNat_Aux1)).

Definition LexOrdNat_Aux3 :
DependentNat_Prod4 -> DependentNat_Prod4 -> Prop:=
(lexprod nat (fun a => DependentNat_Prod3) Peano.lt (fun (a:nat) => LexOrdNat_Aux2)).

Definition LexOrdNat_Aux4:
DependentNat_Prod5 -> DependentNat_Prod5 -> Prop :=
(lexprod nat (fun a => DependentNat_Prod4) Peano.lt (fun (a:nat) => LexOrdNat_Aux3)).

Definition LexOrdNat:
DependentNat_Prod6 -> DependentNat_Prod6 -> Prop :=
(lexprod nat (fun a => DependentNat_Prod5) Peano.lt (fun (a: nat) => LexOrdNat_Aux4)).
```

Figure 3.11: Lexicographic ordering on DependentNat_Prod5

The definition Measure_States is the formal counterpart of Definition 2.3. We first distinguish between the cases of a state being non-final or final by destructing the type $\{j : machine\_States \mid not\ final\ j\}$. In the former scenario, we then separate an initial state from an intermediate one by destructing j. If j is an initial state, its complexity is set to $(1, 0, 0, 0, 0, 0)$ otherwise we define its measure as $(0, |\mathcal{A}_{cand}| - |e|, |h|, \sum_{d \in l_2} |l_d|, |l_1|, |ba|)$, where for a given candidate c, concat (p

```
Lemma wf_LexOrdNat : well_founded LexOrdNat.
```

Figure 3.12: Well-foundedness of LexOrdNat

```
Definition Order_NatProduct : Product_Five_NatSet -> Product_Five_NatSet -> Prop :=
  (fun x y : nat * (nat * (nat * (nat * nat))) =>
   LexOrdNat (Make_DependentNat5 x) (Make_DependentNat5 y)).
```

Figure 3.13: Lexicographic Ordering on $\mathbb{N}^6$

```
Definition Measure_States: {j:Machine_States |not (State_final j)} -> ProdSixNatSet.
 intro H. destruct H as [j ej]. destruct j.
 split. exact 1.
 split. exact 0. split. exact 0. split. exact 0. split. exact 0. exact 0.
 destruct p as [[[[[ba t] p] bl] e] h].
 split. exact 0.
 split. exact (length (X.cand_all) - length (proj1_sig e)).
 split. exact (length (proj1_sig h)).
 split. exact (Sum_nat (map (fun c => length (concat (p c))) (snd bl)))%nat.
 split. exact (length (fst bl)). exact (length ba).
 contradiction ej. unfold State_final. exists l. reflexivity.
Defined.
```

Figure 3.14: Lexicographic Ordering on $\mathbb{N}^6$

c) $= l_c$, concat is the operation of concatening lists, and l_1 and l_2 are respectively the first and second component of the backlog of the intermediate state j, and $\mathcal{A}_{cand}$ is the list of all of the participating candidates in the election.

The lemma wfo_aux shows that the lexicographic definition of the Measure_States is indeed correct with the standard expecations from a lexicographic relation on 5-tuple entities.

```
Lemma wfo_aux:  forall a b c d a' b' c' d' e e' f f': nat,
    (LexOrdNat (Make_DependentNat6 (a, (b, (c, (d,(e,f))))))
               (Make_DependentNat6 (a', (b', (c',(d',(e',f')))))))) <->
    (a < a' \/
    (a = a' /\ b < b' \/
    (a = a' /\ b = b' /\ c < c' \/
    (a = a' /\ b = b' /\ c = c' /\ d < d' \/
    (a = a' /\ b = b' /\ c = c' /\ d = d' /\ e < e' \/
    (a = a' /\ b = b' /\ c = c' /\ d = d' /\ e = e' /\ f < f')))))).
```

Figure 3.15: Correctness of Measure_States

3.3.2 The Formal STV Model of Computation

Everything is ready to formally present the small-step semantics discussed in Section 2.2.1. We only detail the sanity checks for three transitions, namely elect, transfer-elected and transfer-removed. This provides the reader with a clear picture of the formalisation without exhausting them. In light of the discussion, it is

```
Definition SanityCheck_Elect_App
 (R: Machine_States -> Machine_States -> Prop) :=
  (forall premise, forall t p bl e h, (premise = state ([], t, p, bl, e, h)) ->
   (existsT (c: X.cand),
    (length (proj1_sig e)) + 1 <= X.st /\
    In c (proj1_sig h) /\ ((hd nty t) (c) >= X.quota)%Q) ->
     existsT conc, R premise conc).

Definition SanityCheck_Elect_Red
 (R: Machine_States -> Machine_States -> Prop) :=
  (forall premise conclusion, R premise conclusion ->
   exists nba t p np bl nbl e ne nh h,
    (premise = state ([], t, p, bl, e, h)) /\
    length (X.cand_all) - length (proj1_sig ne) <
       (length (X.cand_all) - length (proj1_sig e)) /\
    (conclusion = state (nba, t, np, nbl, ne, nh))).
```

Figure 3.16: Sanity Check of Elect Transition

easy to construct the rest of the sanity checks from their mathematical formulation in section 2.2.1 for other transitions.

To keep the encoding tidy and avoid tedious proofs, we break the sanity check of each transition into two separate declarations. One of them encapsulates the applicability criteria of the transition and the other stipulates the progress constraints. As we explained in the previous chapter, the constraints which we define as the sanity check for a transition are the minimal conditions required from that transition to satisfy which appear in numerous STV algorithms. Minimality enables us to formally realised a larger class of STV algorithms in our system. It also pins down the core meaning behind each transition so that there is a universal semantics for them across various STV schemes. Therefore, the reader knows why we have these conditions instead of other ones as the semantics of the transitions. Nonetheless, we shall refer to some real STV algorithms to make the formalisation of the semantics more concrete at this stage. Later in the subsequent chapter, we dive into a deep discussion about how STV schemes such the Victoria STV and the ACT STV used for real elections fit into our system.

Figure 3.16 explains what it means for a given transition R to legitimately be an elect transition. SanityCheck_Elect_App asserts that R is legally *applicable* as an elect transition if and when the pre-state premise on which it operates is an intermediate state for which the list of uncounted ballots are empty, there is (at least) one candidate c who has exceeded the quota, that candidate is among the continuing ones so that no one eliminated or already elected is chosen mistakenly as a winner, and the length of the list of elected candidates is less than empty seats cardinal so that by electing c we do not go above the initial vacancies of the election.

At this point, one may wonder how is it possible that some algorithm elects more candidates than there are vacancies? Is not restricting the elect transition by including conditions such as $\mathrm{length}\ \mathrm{proj1\_sig}\ (\mathrm{e}) + 1 \leq \mathrm{st}$ fictional? Not indeed and one shall never underestimate the catastrophe that under-specification can introduce. To give

an example, take an STV algorithm that uses the Droop quota. If a user decides to modify the formulation of the Droop quota and replace it with the following formula to obtain their favourite STV scheme, then there can be scenarios where one vacancy is left however there are more than one electable candidates to fill it.

$$\frac{\text{number of valid ballots}}{\text{number of initial vacancies + 1}}$$

Note that the only difference between the above formula and Droop's is that the latter increments the above fraction by 1. The difference may appear small at first glance. Nonetheless, the consequences of this seemingly minor modification can be significant. Note that our framework is designed to accommodate as many possible STV algorithms as there can be. Therefore, in our framework one is able to formalise such an STV scheme. Indeed our framework allows defining *any quota* that a user desires[2]. Because of such considerations, we build restrictions in the machine's semantics to have control over insensible use of our means and ensure, for instance, that no formalised algorithm elects more than there are empty seats.

We elaborate further on some of the constraints in the SanityCheck_Elect_App. As you see in the figure, the applicability check asserts that there exists some candidate *c* who satisfies such and such conditions one of which is having received at least as much as the quota of the election. Therefore, in our system elect *never* applies unless the ones elected have reached or exceeded the quota. Moreover, the existential quantifier is not limiting us to capture the elect transition of STV algorithms such as Tasmania STV where all of the electable candidates are elected in one step.

SanityCheck_Elect_App does not inform us what happens when one applies the elect transition. It merely dictates existence of some post-state conc[3]. SanityCheck_Elect_Red on the other hand declares the immediate consequences of applying elect. It asserts that if and when a transition *R* applies as an elect transition, there must exist some pieces of data such that they form components of the premise and conclusion in such a way that both states are intermediate, some change *may* happen to the empty list of uncounted ballots by this transition so that consequently there are ballots to deal with later, the tally remains the same which means every candidate's amount of attracted vote is not affected in this step. The pile, backlog, list of elected and list of continuing candidates are updated. SanityCheck_Elect_Red intendedly does not specify how exactly they are updated so that it allows instantiation of various ways of updating this information across the STV family.

Note that by "updated" we do not mean that the witness instantiated for the pieces of information in the post-state necessarily differ from the ones instantiated in the pre-state. It only means that their instantiation *may* differ depending on the STV algorithm with which one instantiates the machine and depending on how they decide doing it as we shall detail in the next chapter.

Furthermore, SanityCheck_Elect_Red demands that the length of the elected candidates *e* in the premise must be shorter than its counterpart *ne* in the conclusion.

[2]Recall that the quota is defined in the parameters modules as a generic variable of rational numbers' type. It can be substituted in the instantiation module with an arbitrary rational value.

[3]conc stands for conclusion.

```
Definition SanityCheck_TransferElected_App
 (R: Machine_States -> Machine_States -> Prop) :=
  (forall premise, forall t p bl1 bl2 e h,
   (premise = state ([], t, p, (bl1, bl2), e, h)) ->
   (length (proj1_sig e) < X.st) /\
   (bl1 <> []) /\
   ((bl2 = []) \/ (exists head tail, (bl2 = head :: tail) /\
    (concat (p head) = []))) /\
   (forall c, In c (proj1_sig h) -> ((hd nty t) c < X.quota)%Q) ->
    existsT conc, R premise conc).

Definition SanityCheck_TransferElected_Red
 (R: Machine_States -> Machine_States -> Prop) :=
  (forall premise conclusion, R premise conclusion ->
   exists nba t p np bl nbl h e,
   (premise = state ([], t, p, bl, e, h)) /\
   (length (fst nbl) < length (fst bl)) /\
   (Sum_nat (map (fun c => length (concat (p c))) (snd bl)) =
     Sum_nat (map (fun c => length (concat (np c))) (snd nbl))) /\
   (conclusion = state (nba, t, np, nbl, e, h))).
```

Figure 3.17: Sanity Check of Transfer-elected Transition

Note that the constraint simply specifies a condition about the length rather than the content of these lists. The reason is variance across the STV family in how they treat electable candidates. As mentioned, some of them may elect only one electable person while some others elect all of them in the same application of the elect transition. But one core aspect of the elect transition in STV is the universal fact that the length of these entities varies according to SanityCheck_Elect_Red.

The sanity check for the transfer-elected transition determines what it means for a given transition R to be legally applicable as a correct instance of transferring votes of some elected candidates (figure 3.17). SanityCheck_TransferElected_App specifies that in order to apply R as a transfer-elect transition, the premise should be an intermediate state where there are neither uncounted ballots left to deal with nor there are any votes from eliminated candidates awaiting transfer. Moreover, there must be someone already elected whose votes can be transferred ($\mathrm{fst}(\mathrm{bl}) \neq []$) and some vacancies still remain to fill. Then there exists a post-state conc to which the machine moves by applying R.

SanityCheck_TransferElected_Red lays down the reducibility criteria of applying R. The pre-state premise must be an intermediate state where the list of uncounted ballots and the list of already eliminated candidates whose votes awaits transfer are empty. By moving to the post-state conclusion which is also an intermediate state, the list of the uncounted ballots, pile, and the list of candidates whose surplus should still be transferred ($\mathrm{bl1}$) are updated in such a way that the length of $\mathrm{bl1}$ reduces. Again, the exact mechanism of updating the information is not a concern at this point but only meeting the conditions of SanityCheck_TransferElected_Red is. Other pieces of data such as tally remain the same.

Finally, Figure 3.18 shows the semantics for transfer-removed transition. As the

```
Definition SanityCheck_TransferRemoved_App
 (R: Machine_States -> Machine_States -> Prop) :=
  forall premise, forall t p bl1 bl2 c e h ,
   (premise = state ([], t, p, (bl1, c::bl2), e, h)) ->
   (length (proj1_sig e) < X.st) /\
   (forall c, In c (proj1_sig h) -> ((hd nty t) c < X.quota)%Q) ->
    existsT conc, R premise conc.

Definition SanityCheck_TransferRemoved_Red
 (R: Machine_States -> Machine_States -> Prop) :=
  (forall premise conclusion, R premise conclusion ->
   exists nba t p np bl nbl h e,
    (premise = state ([], t, p, bl, e, h)) /\
    (Sum_nat (map (fun c => length (concat (np c))) (snd nbl)) <
     Sum_nat (map (fun c => length (concat (p c))) (snd bl))) /\
    (conclusion = state (nba, t, np, nbl, e, h))).
```

Figure 3.18: Semantics of the Transfer-removed Transition

reader must already know how to read the definitions, we do not comment any further on the sanity check declarations.

We define the generic STV machine (figure 3.19) as a record of generic transition labels and evidence that each of the labels satisfies the conditions of the relevant applicability and reducibility sanity checks. We choose to formalise the transitions as generic labels so that our formalisation can extend to a broader class of STV algorithms. But the generic transitions in and by themselves do not specify anything pertinent to STV. We therefore include the evidence of the correctness of the labels with respect to the small-step semantics in the record as well to obtain what represents the generic STV algorithm discussed in the previous chapter.

Including evidence for the correctness of the transition labels in turn results in proof automation. Whenever one wants to instantiate the STV, which is the generic machine, the generic transitions need to be instantiated with specific transitions of some concrete STV counting mechanism and then proof obligations are required to be discharged so that the instantiation is verified as a true instance of the STV. Once the instantiation is successfully carried out, the instance automatically inherits properties established for the machine.

Finally, instantiation into the machine enables designing the system modularly in the manner discussed earlier in this chapter. We can perform instantiation in separate modules so that (a) each module only deals with specification and verification of a specific STV algorithm, (b) each extracted program only includes codes related to one single STV rather than several ones in one extracted module and (c) if instantiation of one module fails due to lack of enough proofs from users side, other successful instantiations can be extracted into executable programs without any problem.

```
Record STV :=
  mkSTV {
   initStep: Machine_States -> Machine_States -> Prop;
    evidence_applicability_initStep: (SanityCheck_Initial_App initStep);
    evidence_reducibility_initStep: (SanityCheck_Initial_Red initStep);
   count: Machine_States -> Machine_States -> Prop;
    evidence_applicability_count: (SanityCheck_Count_App count);
    evidence_reducibility_count: (SanityCheck_Count_Red count);
   transfer_elected: Machine_States -> Machine_States -> Prop;
    evidence_applicability_transferElected:
     (SanityCheck_TransferElected_App transfer_elected);
    evidence_reducibility_transferElected:
     (SanityCheck_TransferElected_Red transfer_elected);
   transfer_removed: Machine_States -> Machine_States -> Prop;
   evidence_applicability_transferRemoved:
     (SanityCheck_TransferRemoved_App transfer_removed);
   evidence_reducibility_transferRemoved:
     (SanityCheck_TransferRemoved_Red transfer_removed);
   elect: Machine_States -> Machine_States -> Prop;
    evidence_applicability_elect: (SanityCheck_Elect_App elect);
    evidence_reducibility_elect: (SanityCheck_Elect_Red elect);
   elim: Machine_States -> Machine_States -> Prop;
    evidence_applicability_elim: (SanityCheck_Elim_App elim);
    evidence_reducibility_elim: (SanityCheck_Elim_Red elim);
   hwin: Machine_States -> Machine_States -> Prop;
    evidence_applicability_hwin: (SanityCheck_Hwin_App hwin);
    evidence_reducibility_hwin: (SanityCheck_Hwin_Red hwin);
   ewin: Machine_States -> Machine_States -> Prop;
    evidence_applicability_ewin: (SanityCheck_Ewin_App ewin);
    evidence_reducibility_ewin: (SanityCheck_Ewin_Red ewin)}.
```

Figure 3.19: The STV Machine

3.4 Verification of the Machine Properties

We demonstrate three main properties for the machine. The first one asserts that whenever the machine is in a non-final state if any of the transition labels apply, the machine state changes in such a way that the complexity measure decreases. The second property guarantees that if and when the machine state is a non-final one, always one of the transition labels applies. Using the two properties we prove that any formal execution of the machine on any non-final machine state terminates at a final one accompanied with a *certificate* for this instance of computation.

3.4.1 Reducibility Proof

Each transition label of the STV reduces the complexity measure Measure_States in Figure 3.15. Drawing on the lemmas which establish the reducibility for each transition, we obtain proof that the STV always moves from a given non-final state s_1 to another state s_2 where the measure of the latter is less than the former. Figure 3.20 illustrates how the reducibility property is declared for individual transitions.

IsNonFinal takes a machine state j and e carrying evidence that j is non-final as input and returns a dependent pair (j, e) as output. In other words, IsNonFinal maps non-final machine states to a corresponding dependent type whose values are

```
Definition IsNonFinal: forall j: Machine_States,
   forall e: not (State_final j),
   {j : Machine_States | not (State_final j)}.
    intros j e. exists j. assumption.
  Defined.

Lemma dec_Elect : forall (s: STV) (p c : Machine_States),
 elect s p c -> forall (ep : ~ State_final p) (ec : ~ State_final c),
  Order_NatProduct (Measure_States (IsNonFinal c ec))
                   (Measure_States (IsNonFinal p ep)).

Lemma dec_TransferElected : forall (s: STV) (p c : Machine_States),
   transfer_elected s p c ->
    forall (ep : ~ State_final p) (ec : ~ State_final c),
     Order_NatProduct (Measure_States (IsNonFinal c ec))
                      (Measure_States (IsNonFinal p ep)).
Lemma dec_TransferRemoved : forall (s: STV) (p c : Machine_States),
  transfer_removed s p c ->
   forall (ep : ~ State_final p) (ec : ~ State_final c),
    Order_NatProduct (Measure_States (IsNonFinal c ec))
                     (Measure_States (IsNonFinal p ep)).
```

Figure 3.20: Measure Decrease for Elect, Transfer-elected and Transfer-removed

comprised of the states and proof for each that they are indeed non-final.

We explain how to read one of the above declarations as an example. The lemma dec_Elect expresses reducibility property for the elect transition. Assume s is instance of the STV and two machine states p and c can be connected through an application of elect transition. If both p and c are (provably) non-final states, then the complexity measure of c is less than the measure of p.

```
Lemma measure_dec : forall (s: STV) (p c: Machine_States),
    (initStep s p c)
 \/ (count s p c)
 \/ (elect s p c)
 \/ (transfer_elected s p c)
 \/ (transfer_removed s p c)
 \/ (elim s p c)
 \/ (hwin s p c) \/ (ewin s p c) ->
   forall (ep : ~ State_final p) (ec: ~ State_final c),
   Order_NatProduct (Measure_States (IsNonFinal c ec))
                    (Measure_States (IsNonFinal p ep)).
```

Figure 3.21: Reducibility Property of the Machine

Note that in dec_Elect we need to know p and c are non-final otherwise the property cannot be proven in the strong form declared above. The reason is that if we exclude evidence ep and ec from the assertion, there is no way to know neither of p and c are non-final. On the other hand, the property says that if elect applies to p then we transit to c but we know from the construction of the semantics for elect that c cannot be a final state. Then once we apply the tactical destruct of Coq to prove the property, two subgoals to discharge are left to deal with which relate to when p and c happen to be final states. Now the problem becomes clear if there are no evidences

```
Lemma Rule_Application : forall (s: STV)
 (j1: Machine_States), ~ (State_final j1) ->
    existsT j2, {initStep s j1 j2} + {count s j1 j2} +
                {elect s j1 j2} + {transfer_elected s j1 j2} +
                {transfer_removed s j1 j2} + {elim s j1 j2} +
                {hwin s j1 j2} + {ewin s j1 j2}.
```

Figure 3.22: Applicability Property of the Machine

that they indeed are not final. Hence, we define IsNonFinal and include witnesses for non-finality of both pre- and post-state in the declaration of dec_Elect.

We also prove similar properties for the rest of the machine transitions. Together they allow us to show reducibility for the machine in general as Figure 3.21 illustrates.

3.4.2 Applicability Proof

The STV machine has no dead state so that it can always move from a non-final state to another state. We achieve this property by proving the applicability lemma (Figure 3.22) that guarantees one of the transitions always applies as long as the machine is in a non-final state, and no transition is applicable once the machine reaches to a terminal state.

As discussed in Section 2.3, the sanity checks impose a universally appearing order of application on the machine transitions. We formally realise this order of execution of transitions in the content of the applicability proof. Figure 3.23 schematically represents the overall structure of the proof. The proof begins by using the tactical destruct to distinguish between three cases, namely when the given state j is an initial state, an intermediate state, or a final one. In the first case, the transition initStep applies and the last case there is nothing to do because we have evidence that j is not a final state.

The interesting case happens when j is $\mathrm{state}(ba, t, p, (bl_1, bl_2), e, h)$ for some uncounted ballots ba, tally t, pile p, bl_1 list of elected candidates whose votes await transfer, bl_2 list of eliminated candidates whose votes awaits transfer, and the list of elected e and continuing candidates h. Then the proof proceeds according to the following.

1. Are all seats filled or no continuing candidate left i.e. $\mathrm{length}(e) = st$ or $h = []$?
• if yes then apply ewin and declare winners
• if not then
2. Is $\mathrm{length}\ (e) + \mathrm{length}(h)$ less than or equal to st?
• if yes then apply hwin and declare winners
• if not then,
3. Is there uncounted ballots, i.e. $ba \neq []$?
• if yes then apply count transition
• if not then,
4. Has any candidate reached or exceeded the quota?

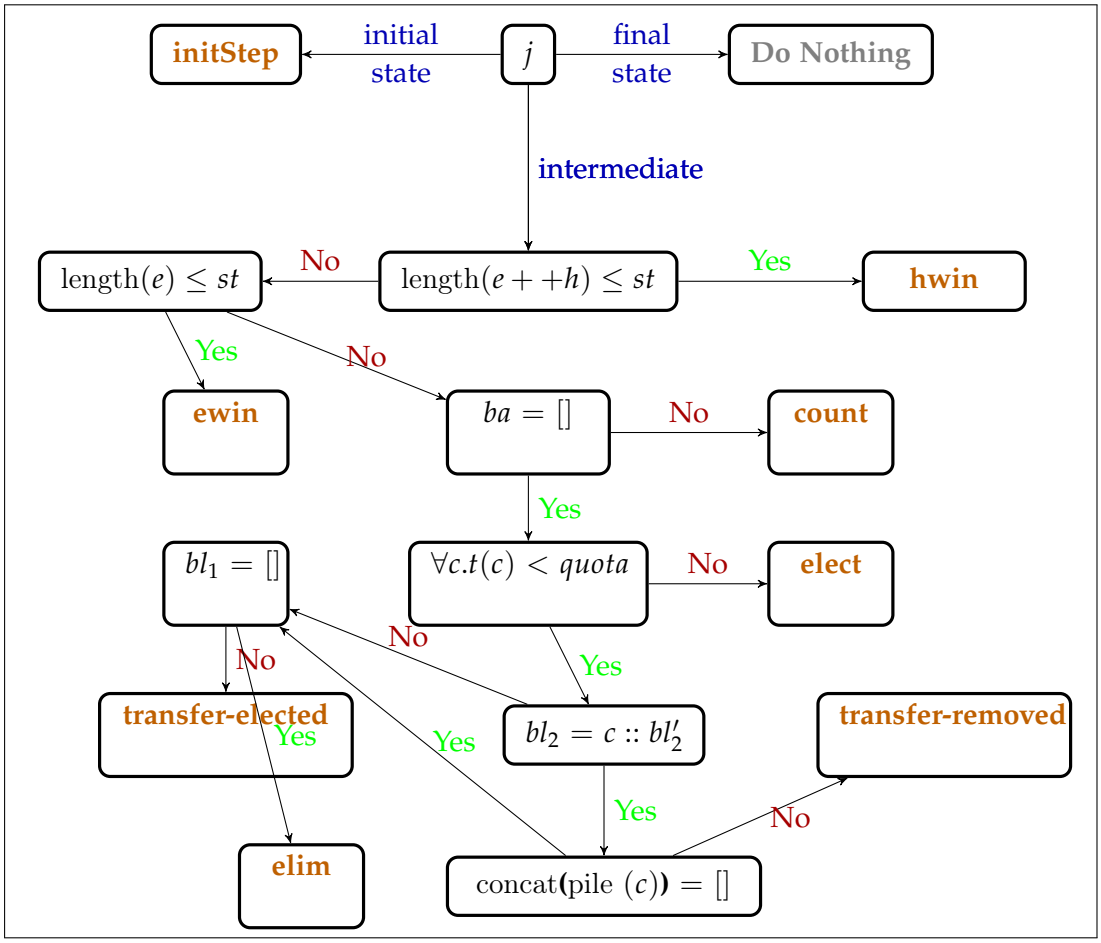

Figure 3.23: Structure of Applicability Proof

• if yes then elect them by elect transition
• if not then,
5. Is the backlog of the eliminated candidates empty?, i.e. $bl_2 = []$?
• if no then,
6. Is any votes from an eliminated candidate at the head position of bl_2 left to transfer? i.e. concat (pile (c)) $= []$?
• if yes then apply transfer-removed
• if not, then
7. Is there any elected candidate whose surplus awaits transfer, i.e $bl_1 = []$?
• if yes then apply transfer-elected
• if no then apply elim

To correctly establish the above structure for the proof we invoke the tactical destruct to visit every possible situation that can occur for the data structure of the machine. Now remember that the applicability sanity checks for each transition ascertain whenever the conditions are satisfied, that transition applies and a post-

state conc exists to which the machine moves using the transition. At each stage of the proof, we instantiate the existential j2 in the Rule_Application lemma with the conc obtained using the sanity check of the transition applied.

Termination The last general property of the STV machine is termination (Figure 3.24). Given a machine state *s*, the content of the applicability proof is repeated so that starting from *s* some sequence of transitions apply where each reduces the complexity measure while moving from its pre-state to the post-state. Eventually, the machine reaches a terminal state where winners of the election are announced and a certificate of this instance of computation is output as well.

```
Theorem Termination : forall (bs: list ballot),
 forall j0 (evj0: ~State_final j0), forall (s: STV),
  existsT j, (State_final j) * (Certificate X.st bs s j0 j).
```

Figure 3.24: Termination Theorem

Put in the context of programming semantics, each execution of the machine has meaning. The sequence of transitions applied along with the machine states visited to reach the final state is the meaning of this execution. The final state resulted from this execution is the value of the execution. Moreover, each value is accompanied by a certificate which carries the meaning of the execution. The certificate in and by itself is meaningful so that it can be investigated for examining the correctness of the value of the execution (with respect to the input of this run) without reference to where and how the certificate is produced.

Chapter 4

Instantiation of the Model

In the preceding chapter, we demonstrated how the generic machine formally realises the *commonalities* in the STV family. In the current chapter, we elaborate on how our framework accommodates formalisation and verification of each individual STV scheme through instantiation of the machine. Recall that the STV machine is a record consisting of generic transition labels and evidence that each generic transition satisfies its sanity checks (see Figure 3.19). Each machine instantiation with an STV algorithm $\mathcal{A}$ requires three steps.

- *specification of transitions.* We want to remove as many trusted layers as possible to minimise the trusted computing base. On the one hand, we must, therefore, verify the correctness of the implementations below against the instructions in $\mathcal{A}$. On the other hand, typically STV algorithms are described in informal environments and languages such as textual documents, which are not suitable for formal reasoning. We hence need a formal counterpart of $\mathcal{A}$'s instructions in order to reason about the implementation's correctness in Coq. We consequently specify declarations in Coq that are the formal specifications of the informal description of $\mathcal{A}$'s transition labels.

- *implementation of transitions.* The machine transitions are merely generic types which do not carry computational content. On the other hand, sanity checks only give a semantics that is common among all STV family members without specifying any meaning particular to individual STV algorithms. In contrast, for computation based on $\mathcal{A}$, we need to implement computational entities that (a) indeed compute an output for a given input and (b) perform the computation as per instructions in $\mathcal{A}$.

- *discharging sanity checks.* Having implemented and formally specified the algorithm $\mathcal{A}$, we proceed to establish proofs that (a) the implementation indeed satisfies the specification so that computational behaviour of the implementation matches with instruction in $\mathcal{A}$, and (b) the specification/implementation successfully discharge the sanity checks so that our specification/implementation is guaranteed to be an authentic STV algorithm. By successful discharge of the semantics requirements, we automatically obtain the guarantee that each instantiated algorithm inherits the machine properties such as termination.

We demonstrate in some detail how the above cycle of specifying, implementing, and discharging the sanity checks develops for elect transition of the ACT STV [**?**]. To this end, we begin by a thorough analysis of the textual description of the ACT STV given in the Subsection 2.1.1 with a focus on the elect transition in order to identify the semantic requirements of the elect that are specific to this scheme. Then, we define a formal specification and an implementation for the informal semantics of the transitions elect, obtained through the analysis. We next discuss how to discharge the elect's sanity checks to complete instantiation of the generic elect transition of the machine with ACT's elect.

To illustrate how instantiated transitions with ACT STV collectively realise the ACT scheme, we briefly discuss formalisation of the ACT's transfer-elect transition as well. We then proceed with a discussion on alternative ways of formalisation of one same STV algorithm by drawing on our detailed analysis and formalisation of the ACT.

We subsequently explain how to extract a Haskell compilable program for each formalised scheme. Each extracted program produces a certificate for each instance of computation performed on an input value. We elaborate on what the certificates are and how they establish transparency and verifiability of the tallying process. Finally, we illustrate some experimental results obtained by executing the extracted Haskell program for the ACT STV on real historical data of elections in the ACT state.

4.1 Analysis of the ACT STV

To pinpoint semantics of ACT STV's transitions, we refer to the textual description of the algorithm given in Subsection 2.1.1 which can also be found at the official webpage of the ACT's government [2]. Our initial effort is to only depend on this document for the algorithm analysis. Nonetheless, we shall see shortly afterwards that the document is occasionally vague on the description of some part of the procedure, or entirely lacks any specification about the subject matter. Therefore, we need to refer to other related resource as well. We shall reference other documents only if more evidence is needed to justify our interpretation, and the text or the ACT's government webpage suggest referring to them for further information.

Recall that the elect transition which corresponds to the action of electing candidates only applies (under circumstances discussed below) to intermediate states of computation. Given an intermediate state $\mathrm{state}(ba, t, p, (bl_1, bl_2), e, h)$, the semantics of the elect transition specifies the post-state $\mathrm{state}(ba', t', p', (bl'_1, bl'_2), e', h')$ to which the computation proceeds by applying this transition. The semantics comprises the pre- and postconditions which impose constraints on when the elect transition can legitimately apply and how to update each piece of information in the post-state.

From a careful examination of the textual description of the ACT STV the following clauses result which comprise an informal semantics for electing a candidate.

- **list of uncounted ballots.**
  - *clause 1.* the list of uncounted ballots in the pre-state of the elect transition is empty.
  - *clause 2.* the list of uncounted ballots in the post-state is also empty.
- **tallies.**
  - *clause 3.* tallies of candidates in the post-state remain the same as in the pre-state.
- **piles.**
  - *clause 4.* pile of any continuing candidate who is not elected remains the same as in the pre-state.
  - *clause 5.* only the ballots in the "last parcel"[1] of an elected candidate consist of their surplus after updating the fractional value that each ballot in the last parcel carries. Update the fractional value of the surplus votes of an elected candidate in the post-state according to the following formula.

$$\text{(formula 4.1)} \qquad \frac{\text{surplus votes of the elected candidate}}{\text{number of ballot papers with further preferences shown}}$$

- **backlog of the elected.**
  - *clause 6.* if a continuing candidate is elected, then they must have reached or exceeded the quota.
  - *clause 7.* update the backlog of the elected candidates in the post-state with the list of newly elected candidates appended to it. The newly elected candidates are themselves ordered according to the amount of votes which each has received.
- **backlog of the eliminated.**
  - *clause 8.* the backlog of the eliminated candidates in the post-state is the same as its counterpart in the pre-state.
- **elected candidates.**
  - *clause 9.* the number of elected candidates should not exceed the number of vacancies left.
  - *clause 10.* elect all continuing candidates who have reached the quota.
  - *clause 11.* The difference of list of elected candidates in the pre- and post-state is exactly the candidates who have newly been elected.
- **continuing candidates.**

[1] We have already explained what the last parcel is in Section 3.2. We also elaborate on it in the next page.

– *clause 12.* neither of the newly elected candidates is a continuing candidate in the post-state.

We continue by analysing the textual document of the ACT STV to illustrate how these clauses emerge. We only elaborate on the non-trivial clauses.

clause 1. The algorithm specifies under Step 3 when the action of electing may come to an effect. It is preceded by the Step 1 which instructs on counting the first preferences in advance of taking any further action. Then in the second sentence under the "Counting the first preferences" clause, the protocol states that "After all the formal first preference votes are counted ... Any candidate who has votes equal to or greater than the quota is now elected". Therefore a precondition for applying the elect action is that there must be no uncounted votes to deal with the pre-state of the action.

clause 5. Under the section "Transferring surplus votes from elected candidates", the protocol dictates to transfer surplus votes of an elected candidate at the fractional value given above in Clause 6. As a result, in the post-state of the elect transition, the fractional value of each ballot in some or all portion of the elected candidate's pile in the pre-state is multiplied at Formula 4.1 to obtain an updated pile in the post-state for the already elected candidate.

We need to clarify what we mean by "some or all portion of the ... pile". The document specifies two scenarios regarding the surplus of votes of an elected candidate. The first scenario explains that if "a candidate has received more than a quota of first preference votes", then "all the ballot papers received by the candidate are distributed" at a reduced fractional value. The second scenario states that if "a candidate has received more votes than the quota following a transfer of votes from another elected candidate or from an excluded candidate", then "only that "last parcel" of ballot papers that the candidate received are distributed to continuing candidates" at a reduced fractional value. The two cases above verbatim seem significantly different. What the algorithm does is transferring only and always the last parcel of an elected candidate's votes. In the first scenario, the last parcel consists of all of the votes that the elected candidate has received in the election. But in the second scenario, it only contains some of the votes attracted which are the ones that resulted in getting elected.

clause 7. the first sentence in the clause follows from Step 3 and 4 of the protocol. However, the second sentence cannot be inferred from the document. We obtain it by referring to the software used by the ACT Electoral Commission to count votes in the lower house elections. The authorities of the state employ a system called eVACS for casting, recording and tallying votes. The counting module of eVACS system is publicly available [2]. The counting component of eVACS transfers surplus of elected candidates one by one, and based on the magnitude of the elected candidates' tallies. Note that under "Transfer surplus votes from elected candidates", the protocol

specifies to distribute surplus votes of elected candidates one by one. However. the document does not mention in what order to do that.

clause 10. Step 3 of the protocol states that "Any candidate with votes equal to or greater than the quota is declared elected". This specification is ambiguous because there can be two different interpretations of "Any candidate". One understanding is that the text means (a) declaring a candidate elected if they have reached or exceeded the quota and (b) only one among (possibly) many other candidates who satisfy (a). The other interpretation is that every candidate who has reached or exceeded the quota is elected. We shall argue that the latter is the correct understanding.

The reader, especially if they are a native speaker of English would find our discussion on the correct understanding of "Any candidate" misguided. This would be because simply a typical speaker of English would readily express that certainly the protocol means *all candidates*. However, we shall point to another piece of the protocol that makes us hesitate to rush for the usual understanding of "Any". Under the section "Transferring surplus votes from elected candidates" the protocol asserts that "Any further candidates that have votes equal to or greater than the quota are elected". Given that these documents carry legislative content used for executing orders, one expects a reason for using "Any" and "candidate" differently in one same document. Unlike Step 3, here the text uses the plural of the word "candidate", instead of the singular form. Also one can simply conceive of an STV algorithm whose action of electing varies depending on how a candidate reaches or exceeds the quota, i.e. as a result of receiving first preference votes upon the first application of the count transition, or as a consequence of attracting transferred votes from an eliminated or elected candidate. Therefore a discussion on the correct interpretation of "Any candidate" is necessary to justify our formalisation.

The Merriam-Webster dictionary lays down three entries describing the meaning of "any" as an adjective. The first two meanings relate to the current context;

1. one or some indiscriminately of whatever kind:
    a. one or another taken at random
    b. every —used to indicate one selected without restriction
2. one, some, or all indiscriminately of whatever quantity:
    a. one or more —used to indicate an undetermined number or amount
    b. all —used to indicate a maximum or whole
    c. a or some without reference to quantity or extent

We proceed with an affirmative argument for why the item 2.b is the correct interpretation of "any" in this context, instead of negatively excluding the alternatives one by one to reach a conclusion in advantage of our stand.

Step 3 is an important part of the algorithm explaining the act of electing. The first clause of the step asserts that "Any candidate with votes equal or greater than

the quota is declared elected". Then in the second clause, it immediately queries "if all vacancies have been filled then the election is complete". The protocol here *suggests* electing all of the electable candidates. This suggestion sounds reasonable for two reasons. First of all, Step 3 comes after Step 1 which dictates applying the action of counting until every ballot is counted. Second of all, the number of initial vacancies in the ACT electorates is either 5 or 7. This means that it is possible for some candidates to reach or exceed the quota and consequently fill all of them simply after the first application of the counting action. Therefore immediacy of the first clause of Step 3 and the second one querying if all vacancies are filled supports our interpretation of "Any candidate".

The above argument begs for more evidence because there is still room for the alternative interpretation. To reason further, note that in both of the first and second definitions of "Any", Merriam-Webster includes the word "indiscriminately". The protocol in Step 3 states any candidate "with votes greater than or equal to the quota" is declared elected. Therefore the criterion for being elected is to simply reach or exceed the quota. Appealing to any other metrics for electing only one electable candidate instead of others is an act of discrimination and is contrary to the statement of the protocol.

Moreover, as documents of the ACT Electoral Commission attest, authorities declare that the system reflects proportionality of votes. If one understands "Any candidate" as only one among all of those qualified to be elected, then ACT STV loses the proportionality property (see Beckert et al[14][2]).

In view of the argument, we conclude that the ACT STV elects all of the candidates who reach or exceed the quota in a single application of the elect transition.

Having analysed and specified the informal semantics of the ACT's instructions for electing and transferring surplus of elected candidates, we next discuss how we formalise the semantics in Coq.

4.2 Formalisation and Verification of the ACT STV

As you see in the preceding section, we destruct the semantics of each transition, including elect and transfer-elected, into clauses. This destruction bears two fruits.

- *Modular implementation.* The first one is allowing us to implement formalisation of different STV algorithms modularly. Consequently, our framework is modular in not only design but also implementation as well. This means that we can isolate and compartmentalise implementation of entities within each module based on relatedness of their functionality, allow sharing of some parts of the implementation that various STV use without duplication of the process, and

[2]They demonstrate that CADE STV whose electing action only elects one candidate upon each application because of CADE's formulation of the quota. This formulation together with other features of CADE allow a party who is in majority to have a successful strategy in winning all of the vacancies

```
100  Definition ACT_Elect (prem: Machine_States) (conc: Machine_States) : Prop :=
101   exists t p np (bl nbl: (list cand) * (list cand)) nh h (e ne: {l : list cand | length l <= st }),
102      prem = state ([], t, p, bl, e, h) /\
103      exists l,
104       (l <> [] /\
105       length l <= st - length (proj1_sig e) /\
106       (forall c, In c l -> In c (proj1_sig h) /\ (hd nty t (c) >= quota)%Q) /\
107       ordered (hd nty t) l /\
108       Leqe l (proj1_sig nh) (proj1_sig h) /\
109       Leqe l (proj1_sig e) (proj1_sig ne) /\
110       (forall c, In c l -> ((np c) = map (map (fun (b : ballot) =>
111           (fst b, (Qred (snd b * (Update_transVal c p (hd nty t))))%Q))) [(last (p c) [])])) /\
112       (forall c, ~ In c l -> np (c) = p (c)) /\
113       fst nbl = (fst bl) ++ l) /\
114    conc = state ([], t, np, nbl, ne, nh).
```

Figure 4.1: Formal Specification of the ACT's elect transition

verify the shared parts of the implementation once and for all so that users can confidently rely on the implementation.

- *Formal Syntactic Comparison of STV algorithms.* To our best knowledge, there is no existing framework providing a uniform treatment of the STV as a family of algorithms. Our framework is the only structure that facilitates comparison of STV algorithms *from a purely syntactic* perspective, rather than textual interpretative treatments in an informal language.

We demonstrate how we modularly formalise and verify the clauses given for the informal semantics of the elect and transfer-elect transitions. However, we first lay down the formal specification of the elect transition which is composed of the formal counterparts of the informal clauses. We then explain how each part corresponds with a clause in the informal elect's semantics. We subsequently elaborate on some details of the specification and implementation of one of the clauses as an exemplary case.

Figure 4.1 illustrates the formalised ACT elect transition. We describe line by line what each part means and how they match with the informal semantics.

- *Line 100.* we store the definition of the transition under the name ACT_Elect. The definition is parameterised in terms of two machine states called prem which is the pre-state of the elect transition, and conc which is the post-state.[3]. The returned value of this definition is of type Prop.

- *Line 101.* as we explain in Section 1.6, terms in Coq carry computational content with them and proposition are types. In particular, the existential quantifier corresponds to the Sigma type constructor which we describe in Section 3.2.

[3] prem and conc respectively stand for the words premise and conclusion.

Accordingly, values of a Sigma type (therefore an existentially bound expression) are pairs where the second component carries evidence that the first component satisfies the property required by the construction of the Sigma type to which they belong.

Now remember that we eventually want to extract a *provably correct* Haskell compilable implementation of the ACT STV. Using the existential here forces us to meet two expectations:

1. the variable bound by the existential quantifier should be substituted with a *computational* entity in Coq.
2. the computational entity substituted for the quantified variable above must satisfy the properties which the quantified variable appears in.

For an example, line 104 declares that the variable e standing for the list of already elected candidates in the pre-state and the existentially bound variable l standing for the list of candidates which are elected by the current application of the elect transition should together have a length less than the initial number of vacancies. Any computational entity substituted for e and l have to comply with this expectation through proofs. The result of such specification in terms of the existential quantifiers and properties to satisfy is that once we extract these computational components, we are guaranteed in light of the proofs performed that they behave correctly.[4]

- *Line 102.* the premise which is the pre-state that the elect transition applies to is an intermediate state (as the sanity check of the elect in Section 2.2.1 requires) where the list of uncounted ballots is empty (Clause 1).

- *Lines 103, 104, and 105.* there exists a list l of candidates which satisfies some properties that follow after line 103. Two of the properties express that the list is not empty and the length of this list plus the list of already elected candidates in the pre-state is less than the initial number of vacancies st (Clause 9).

- *Line 106.* any member of the list l has reached or exceeded the quota (Clauses 10 and 6).

- *Line 107.* the term ordered is an inductive definition declaring that the list l is ordered with respect to the amount of the tally function (Clause 7).

- *Line 108.* the property Leqe declares that the list of continuing candidates h in the pre-state prem is equal to its counterpart list nh in the post-state conc except that h includes elements in l as well. In other words, the property expresses that concatenation of l and nh is a permutation of h (Clause 12).

[4]These computational entities are instances of what we referred to as proof-carrying code in the introduction as each of them carries their correctness (the second component of the existential with them. Therefore once extracted, an observer can simply inspect the extracted code itself directly without any reference to the Coq source that the functions are truly correct.

- *Line 109.* this line states that concatenation of the l and the list of already elected candidates e in the pre-state prem is a permutation of ne in the post-state conc (Clause 11).

- *Lines 110 and 111.* the fractional value of the last parcel for any newly elected candidate in l should be updated by multiplying it at Formula 4.1 (Clause 5).

- *Line 112.* the pile of any candidate who is not in the list l remains the same as in the pre-state (Clause 4).

- *Line 113.* backlog of the elected candidates $\mathrm{fst}\ (\mathrm{nbl})$ in the post-state should equal to the concatenation of the backlog of the elected candidates $\mathrm{fst}\ (\mathrm{bl})$ with the list of newly elected candidates l (Clause 7).

- *Line 114.* the post-state of the elect transition is also an intermediate state where the list of uncounted ballots is empty (Clause 1), the tally function remains the same as in the pre-state (Clause 3), and the backlog of the eliminated candidates also remains untouched (Clause 8).

We next provide some insight into why line 108 in Figure 4.1 is an appropriate specification of Clause 12 of the elect's informal semantics. Also, we shall discuss how we implement and verify the implementation of line 108.

4.2.1 Formalisation and Verification of Auxiliary Assertions

Clause 12 declares that "neither of the newly elected candidates is a continuing candidate in the post-state [of the elect transition]". Recall that continuing candidates are constructed to have a Sigma type where the second component of each value of this type witnesses that the first component is free of duplication. Also note that the newly elected candidates are indeed continuing candidates in the pre-state of the transition which are consequently duplicate-free as well. Therefore we can rephrase Clause 12 declaring that the concatenation of the list of continuing candidates in the post-state and the new ones is a permutation of the continuing candidates' list in the pre-state.

Specifying Clause 12 in terms of the permutation construct neutralises our STV formalisation from our choice of data structure for recording information. To elaborate more, we build the machine states on top of the list structure. How candidates are listed in, for example, the list of elected candidates should only be influenced by the instruction of STV algorithms rather than restrictions introduced by our way of modelling. On the other hand, recall from our discussion in the previous section that existential quantifiers appearing under the definition of the ACT's formalised elect transition need to be instantiated with functions that compute the updated continuing candidates and the newly elected ones in the post-state. Choosing permutation as a property which these functions must meet makes certain that no part of our formalisation affects the winners of the election.

```
Definition Leqe {A:Type} (k :list A) (l: list A) (nl: list A): Prop:=
 Permutation nl (l++k).
```

Figure 4.2: The formal counterpart of clause 12

We formalise clause 12 by the property Leqe given in Figure 4.2. We then implement the computational assertion Removel used for computing the list of continuing candidates in the post-state appearing under the existential quantifier in Figure 4.1.

```
(*removes every element of the list k which exist in the list l*)
Fixpoint Removel (k :list X.cand) (l :list X.cand) :list X.cand:=
match l with
    [] => []
    |l0::ls => if (X.cand_in_dec l0 k) then (Removel k ls) else (l0::(Removel k ls))
end.
```

Figure 4.3: The implemented function for computing list differences

The function Removel takes two lists of candidates k and l and removes every element of k which appears in l. We prove several statements about the correct behaviour of the function Removel with respect to some desirable properties the most important of which is no_permutation in Figure 4.4.

```
Lemma nodup_permutation:
 forall (k l :list X.cand),
  (forall x, In x k -> In x l) -> NoDup k -> NoDup l -> Leqe (Removel k l) k l.
```

Figure 4.4: The function Removel satisfies the formal counterpart of Clause 12

In a similar manner, we implement other computational assertions which substitute the existentially bound variables in Figure 4.1, and verify them against the formal counterparts of the informal semantic clauses for the elect transition. Later we use Coq's extraction mechanism to compute real election results with the extracted versions of these functions. However, we first need to discharge the sanity checks of the machine so that the framework allows us to proceed with the code extraction.

4.2.2 Instantiation of the Machine with the Formally Verified ACT STV

We define the generic STV machine in Figure 3.19 as a record consisting of three kinds of fields. One kind of field records the generic specification of transition labels. The other two require evidence for the correctness of the generic transition labels with respect to the reducibility and applicability sanity checks of the machine.

An instantiation of the machine with an algorithm $\mathcal{A}$ hence amounts to taking three steps. The first step is to instantiate the generic transitions with a formal definition of $\mathcal{A}$'s specific transitions, such as $\mathrm{ACT\_Elect}$ in Figure 4.1. The second and

```
Lemma ACT_Elect_SanityCheck_App : SanityCheck_Elect_App ACT_Elect.
```

Figure 4.5: ACT STV's Evidence for the Applicaility Sanity Check

third steps are proving that the instantiation of the generic labels satisfies the reducibility and applicability requirements. If and when the two latter succeed, then one can create a value of the record type STV (Figure 3.19) to obtain a verified version of the machine for ACT STV.

The lemma ACT_Elect_SanityCheck_App in Figure 4.5 shows that our formalisation of the ACT's elect transition is correct with respect to the elect applicability sanity check of the machine. We shall discuss what it takes to construct a proof for the lemma as an illustration of how discharging a sanity check happens.

According to the elect sanity check SanityCheck_Elect_App defined in Figure 3.16, we need to instantiate a post-state conc in such a way that the minimal applicability conditions of the generic elect transition hold as well. The reducibility sanity check of the generic elect given in the same figure specifies that this post-state conc is an intermediate machine state. Now we know that each intermediate state has seven components each of which must be specified in order to obtain a witness for conc. Once again the reducibility sanity check hints us in what kind of entities to instantiate for each component of conc. As you see in the reducibility sanity check, the post-state conc is specified as state(nb,t,np,nbl,ne,nh) where the variables nb, np, nbl, ne, and nh are bound by an existential quantifier. Now recall from our earlier discussion in this chapter that existential quantifier in Coq carries computational content with it and that we intend to perform computation on real election data upon code extraction. We therefore need to instantiate these variables with functions that compute an output for a given input. These functions are the ones which we mentioned and then exemplified an instance of, namely Removel, in the previous subsection.

Once we instantiate the post-state conc in the manner explained above, we must carry out two kinds of proof obligations. The first one regards demonstrating that the minimal applicability constraints in SanityCheck_Elect_App hold for the instantiated variables. Discharging these requirements successfully implies that the meaning of each instance of computation with the ACT's elect transition accords with the semantics of the elect transition of the STV family. The second kind of proof obligations is to show that the implementations which we instantiate for the components in conc compute correctly according to the formal specification of the ACT's elect. For establishing this latter kind of proofs we simply draw on the proofs which we already have carried out for the correctness of the auxiliary assertions.

In a similar way, we prove that our specification ACT_Elect satisfies the reducibility sanity checks of the generic elect transition as Figure 4.6 depicts.

```
Lemma ACT_Elect_SanityCheck_Red : SanityCheck_Elect_Red ACT_Elect.
```

Figure 4.6: ACT STV's Evidence for the Reducibility Sanity Check

```
Definition ActSTV :=
 (mkSTV
  (ACT_InitStep)
   (ACTInitStep_SanityCheck_App) (ACTInitStep_SanityCheck_Red)
  (ACT_count)
   (ACTCount_SanityCheck_App) (ACTCount_SanityCheck_Red)
  (ACT_TransferElected)
   (ACT_TransferElected_SanityCheck_App) (ACT_TransferElected_SanityCheck_Red)
  (ACT_TransferElim)
   (ACT_TransferRemoved_SanityCheck_App) (ACT_TransferRemoved_SanityCheck_Red)
  (ACT_Elect)
   (ACT_Elect_SanityCheck_App) (ACT_Elect_SanityCheck_Red)
  (ACT_elim)
   (ACTElim_SanityCheck_App) (ACTElim_SanityCheck_Red)
  (ACT_hwin)
   (ACTHwin_SanityCheck_App) (ACTHwin_SanityCheck_Red)
  (ACT_ewin)
   (ACTEwin_SanityCheck_App) (ACTEwin_SanityCheck_Red)).
```

Figure 4.7: The ACT STV Version of the Machine

When discharging the sanity checks finishes successfully for the rest of ACT's transitions, we are ready to instantiate the STV machine. This gives us a verified version of the machine to use for extraction of a correct Haskell implementation of ACT STV. Figure 4.7 illustrates the record created accordingly for the ACT flavor of the machine.

We next provide a concrete example of counting votes with the ACT STV as a version of the machine. Through this example, we illustrate how individual transitions of the ACT operate on data and how they collectively behave as components of the ACT machine.

4.2.3 A Concrete Example of Counting Votes with the ACT STV

We execute the ACT machine on a small fictional election to exemplify how it manipulates data for computing winners of the election. Figure 4.9 illustrates a transcript which the machine produces upon its execution on the input election parameters. For this election instance, three candidates A, B, and C are competing for two vacancies. The votes recorded as cast by voters consists of eight ballots each of which initially carries a fractional value 1/1. Figure 4.8 depicts the preferences expressed on the ballots.

In the following, we describe step by step how the ACT machine progresses through the states in the Figure 4.9 to terminate at a final state where B and A are declared as the winners of the election. The certificate in the figure comprises two parts. The first part called header records the first three lines from the top to bottom consisting of the quota, number of vacancies and the list of competing candidates. The second part of the certificate encapsulates the machine states visited to compute the winners.

Ballot	Preference	Ballot	Preference
b_1	$[A, B]$	b_5	$[C, B, A]$
b_2	$[A, B, C]$	b_6	$[B]$
b_3	$[B, C, A]$	b_7	$[B, C]$
b_4	$[B, A]$	b_8	$[]$

Figure 4.8: Ballots' Preferences of a Sample Election with ACT STV

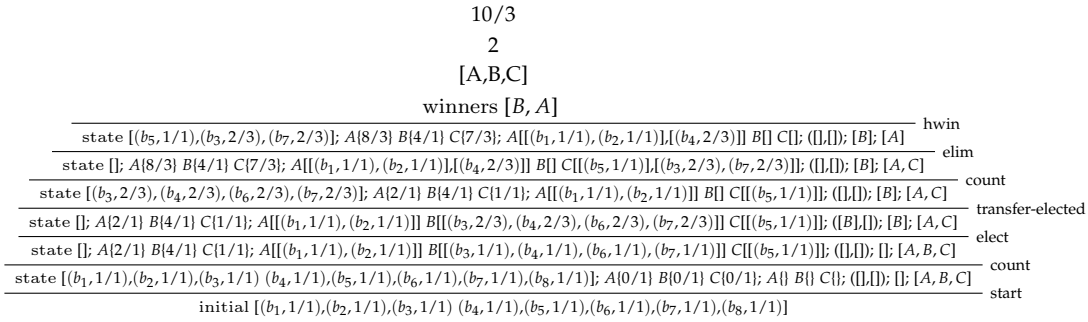

Figure 4.9: A Trace of Machine States Visited to Compute the Winners

Quota. The ACT STV's protocol and subsequently the ACT machine compute the quota of an election based on the following formula.

$$\frac{\text{number of valid ballots}}{(\text{number of initial vacancies}) + 1} + 1$$

The ACT protocol specifies that if a ballot consists of less than five preferences for candidates expressed on the ballot, then that ballot is invalid and is therefore placed aside[5]. Since b_8 does not have any preferences, it is an invalid vote. Hence the quota of the election is (7 / (2 +1)) + 1 = 10/3.

Number of Vacancies and Competing Candidates. The second line from the top consists of the number of vacancies which is 2. The third line comprises the list of competing candidates in the election, namely $[A, B, C]$.

The rest of the lines in the transcript consist of intermediate machine states and the last one which is the final state of the computation. Data of every intermediate state are separated from one another by the semi-colon ";". The order according to which data is presented matches with the order of the abstract representation of the intermediate machine states in Figure 3.5. We shall continue explaining how the computation visits each one the states.

Start. The first machine transition to apply is always the start which filters the valid votes and stages them to be counted next. Recall from our discussion in the Sec-

[5]Note however, to keep the election sample simple and enough illustrative, we only give three candidates and therefore obviously there cannot be an expression of five separate preferences.

tion 3.1 we explain that the parameters modules include a record field generically specifying the function ValidBallot used for deciding which ballots are formal. Also in the same section, we elaborate that this generic function is instantiated in each module with the specific definition of vote validity that the instantiated algorithm uses.[6] According to the ACT STV's protocol, a valid vote includes expression of at least five distinct candidates' names as the vote preference. Based on this, we instantiate the parameter ValidBallot with a function that decides on the validity of the votes accordingly.

Count. The list of uncounted ballots is not empty. Therefore the count transition applies to allocate votes to candidates based on the first preferences expressed on the votes and then update their tally accordingly. Here candidate A receives two votes each of value $1/1$, namely $(b_1, 1/1)$ and $(b_2, 1/1)$. Candidate B attracts four votes which are $(b_3, 1/1)$, $(b_4, 1/1)$, $(b_6, 1/1)$, and $(b_7, 1/1)$. Candidate C receives the vote $(b_5, 1/1)$. The votes attracted are now placed in the pile of each candidate. Note that piles are stored parcel by parcel. This is the reason for two opening and closing brackets, i.e. "[" and "]" respectively, for listing the votes obtained. As all votes are counted, the list of the counted in post-state is empty. Tallies and piles are also updated as explained. Every other component of the post-state remains the same as their counterpart in the pre-state.

Elect. Candidate B exceeds the quota. Therefore they are elected and their name is added to the list of elected candidates and the backlog of the elected in the post-state. Also, the fractional value of every ballot in the last pile of B, which is the is the only one here, is updated by multiplying the values at the Formula 4.1. Since B is elected, their name is removed from the list of continuing candidates and this list is updated accordingly in the post-state.

Transfer-elected. Candidate B has surplus awaiting distribution. The transfer-elected stages the surplus to be transferred next by placing them in the list of uncounted ballots. As the ACT protocol requires transferring surplus of an elected candidate in a single application of the transition, the backlog of the elected in the post-state is updated with B's name removed from it. Also, the pile B is emptied in the post-state. Every other piece of information in the pre-state is preserved and appears in the post-state.

Count. There are ballots in the list of uncounted ballot. So another application of the count transfer applies to distribute the *next* preferences on the ballots. Note that the second preferences are now treated as first preference votes except that their value is reduced. The ballot $(b_4, 2/3)$ goes to candidate A, and the ballots $(b_3, 2/3)$ and $(b_7, 2/3)$ are allocated to candidate C. Consequently, A's tally becomes $8/3$ and C's reaches to $7/3$.

[6] see Figure 3.4 for an example of the ValidBallot instantiation.

Elim. No one reaches or exceeds the quota. There are still empty seats to fill and there are no surplus votes to distribute. Therefore one has to be eliminated. *C*'s tally is less than *A*'s. Hence *C* is excluded and their ballots are placed in the list of uncounted ballots to be counted next. Also, *C*'s pile is emptied for the same reason as we empty pile of an already elected candidate once their surplus is dealt with. Finally, *C*'s name is removed from the list of continuing candidates in the post-state.

Hwin. The number of elected and continuing candidates together equal to the number of vacancies. Therefore candidates in the two lists are declared elected. The computation therefore has reached a final state.

4.3 The Victoria and CADE STV Algorithms

We next discuss the informal and formal semantics of the elect transition of the Victoria and CADE STV schemes. To formalise the elect's semantics, we examine the informal description of the algorithms with the same degree of meticulousness as we did for the ACT STV. The astute reader is already familiar with how that process proceeds. Therefore in the rest of this sequel, we avoid repetition and only comparatively specify the informal semantics of elect. We then lay down the corresponding formal semantics without explaining in detail how the two match.

The semantics of the Victoria STV's elect has many similarities with the ACT's. The former varies from the latter in two aspects. First, the Victoria STV uses a different formula for reducing the fractional value of the surplus votes of an elected candidate. Second, the algorithms differ on what comprises the surplus votes of an elected candidate. This latter difference reflects in the elect's semantics. Despite the fact that the variations appear minor, the end result of their computation with on same input may significantly differ. One reason for such a possibility is the sensitivity of STV algorithms to small changes[7] which influences the meaning of computation.[8]. This is another motivation for a careful formal treatment of STV algorithms.

Clauses of the Victoria's elect semantics are the same as the ACT's except for Clause 5. We replace it with the following.

- *clause 5'.* all ballots in the whole pile of an elected candidate consist of their surplus after updating the fractional value that each ballot in the last parcel carries. Update the fractional value of the surplus votes of an elected candidate in the post-state according to the following formula.

(formula 4.3)
$$\frac{\text{surplus votes of the elected candidate}}{\text{number of ballot papers in the whole pile of the elected candidate}}$$

[7] As we mentioned, STV algorithms have a small margin of victory [18].
[8] see [64] for election examples with and without the last parcel effect.

```
97   Definition VIC_elect (prem: Machine_States) (conc: Machine_States) : Prop :=
98    exists t p np (bl nbl: (list cand) * (list cand)) h nh e ne
99     prem = state ([], t, p, bl, e, h) /\
100    exists l,
101     (l <> [] /\
102     length l <= st - length (proj1_sig e) /\
103     (forall c, In c l -> In c (proj1_sig h) /\ (hd nty t (c) >= quota)%Q) /\
104     ordered (hd nty t) l /\
105     Leqe l (proj1_sig nh) (proj1_sig h) /\
106     Leqe l (proj1_sig e) (proj1_sig ne) /\
107     (forall c, In c l -> ((np c) = map (map (fun (b : ballot) =>
108       (fst b, (Qred (snd b * (Qred ((hd nty t)(c)- quota)/(hd nty t)(c))))%Q))) (p c))) /\
109     (forall c, ~ In c l -> np (c) = p (c)) /\
110    fst nbl = (fst bl) ++ l) /\
111    conc = state ([], t, np, nbl, ne, nh).
```

Figure 4.10: Formal Specification of the Victoria's elect transition

As Clause 5′ specifies, the Victoria STV requires transferring all surplus of the ballots received by a candidate at a reduced value. The fractional value of each ballot is multiplied at the Formula 4.3 to obtain a new transfer value for the vote. The surplus of the elected candidate then is the difference of the aggregation of all these updated fractional values of the ballots and the quota.

Figure 4.10 illustrates the formalised elect's semantics. As you see in the figure, lines 107 and 108 realise Clause 5′.

CADE STV radically differs from "standard" STV algorithms. CADE uses a majoriterian quota computed according to the following.

$$\frac{\text{number of the valid ballots}}{2} + 1$$

Therefore the quota of an election which uses CADE's STV is independent of the number of initial vacancies of the election. This allows a party who is in majority to have a successful strategy to win all of the seats [14], contrary to numerous STV schemes used in Australia which are difficult to manipulate [34]. CADE's odd style of electing becomes more interesting when we come to know that surplus votes, all of the votes received by the elected candidate, are transferred at the full fractional rate 1/1.

The unusual behaviour of CADE STV pervades to the other transitions' semantics as well. Because of this behaviour, CADE is sometimes disputed to be a true member of the STV family. Nonetheless, our framework flexibly accommodates even the extraordinary ones. The semantics of CADE's elect have the same clauses as ACT's except for the following ones.

- *clause 2″.* update the list of the uncounted ballots in the post-state to include all of the election ballots initially recorded to be counted.

- *clauses 4″.* empty the pile of every candidate in the post-state.
- *clause 5″.* the fractional values carried by ballots in the pile of the elected candidate remain as in the pre-state.
- *clause 7″.* the backlog of the elected in the post- and pre-state equals.
- *clause 10″.* elect only one among all of the candidates who have reached or exceeded the quota.
- *clause 12″.* update the list of continuing candidates in the post-state to include all of the competing candidates including those previously eliminated, but none of those who already have been elected.

Figure 4.11 illustrates the formal definition of the CADE's elect transition. Put in a nutshell, CADE STV *restarts* the tallying process once some candidate is elected. The candidates who already have been eliminated are "*resurrected*" to again continue in the process (item (6)). Consequently, all of the ballots are placed in the list of uncounted ballots to deal with (item (9)). However, in the restart, none of the already elected candidates participates as a continuing candidate anymore (items (5,6)). In other words, the election restarts as if the elected have never competed in the election at all.

```
Definition CADE_elect
 (prem: Machine_States) (conc: Machine_States) : Prop :=
  exists nba t p np bl nbl  nh h e ne,
  (1) prem = state ([], t, p, bl, e, h) /\
  (2) exists c,
  (3)  length (proj1_sig e) + 1 <= st /\
  (4)  In c (proj1_sig h) /\ (hd nty t (c) >= quota)%Q /\
  (5)  eqe c (proj1_sig e) (proj1_sig ne) /\
  (6)  (forall d, In d (proj1_sig nh) <->
        In d cand_all /\ (~ In d (proj1_sig ne))) /\
  (7)  (forall d, In d cand_all -> (np d = [])) /\
  (8)  (fst nbl = []) /\
  (9)  (nba = Append_All cand_all p) /\
  (10) conc = state (nba, t, np, nbl, ne, nh).
```

Figure 4.11: Formal Specification of the CADE's elect transition

We give an example election to illustrate how tallying votes with CADE STV progresses. Assume candidates A, B, and C are competing for two vacancies where B and C's party is the same and in competition with A's. Also suppose Figure 4.12 consists of ballots recorded for this election.

Figure 4.13 shows how CADE machine computes the winners. As you see in the figure, despite that candidate C has received fewer first preferences than A, C gets elected. Because of CADE's mechanism of counting votes, voters can only prefer candidates from their favourite party and consequently guarantee all empty seats for them.

Vote	Preference	Value	Vote	Preference	Value
v_1	$[A]$	$1/1$	v_5	$[B, C]$	$1/1$
v_2	$[A]$	$1/1$	v_6	$[B]$	$1/1$
v_3	$[B, C]$	$1/1$	v_7	$[B, C]$	$1/1$
v_4	$[B, C]$	$1/1$	v_8	$[B, C]$	$1/1$

Figure 4.12: Ballots' Preferences of a Sample Election with CADE

$$
\begin{array}{c}
5/1 \\
2 \\
\text{[A,B,C]} \\
\text{winners } [B, C] \\
\hline
\text{state } [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8];\ A[2/1]\ B[6/1]\ C[6/1];\ A[]\ B[]\ C[];\ ([],[]);\ [B, C];\ [A] \quad \text{ewin} \\
\hline
\text{state } [];\ A[2/1]\ B[6/1]\ C[6/1];\ A[[v_1, v_2]]\ B[]\ C[[v_3, v_4, v_5, v_6, v_7, v_8]];\ ([],[]);\ [B];\ [A, C] \quad \text{elect} \\
\hline
\text{state } [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8];\ A[2/1]\ B[6/1]\ C[0/1];\ A[]\ B[]\ C[];\ ([],[]);\ [B];\ [A, C] \quad \text{count} \\
\hline
\text{state } [];\ A[2/1]\ B[6/1]\ C[0/1];\ A[[v_1, v_2]]\ B[[v_3, v_4, v_5, v_6, v_7, v_8]]\ C[];\ ([],[]);\ [];\ [A, B, C] \quad \text{elect} \\
\hline
\text{state } [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8];\ A[0/1]\ B[0/1]\ C[0/1];\ A[]\ B[]\ C[];\ ([],[]);\ [];\ [A, B, C] \quad \text{count} \\
\hline
\text{initial } [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8] \quad \text{start}
\end{array}
$$

Figure 4.13: A Transcript of Computing Winners for the Small Election with CADE

4.4 Alternative Formalisations for One Same STV Scheme

We provided some explanation on the formalisation of the machine semantics in the Subsection 3.3.2. Nonetheless, we decided to delay a full elaboration on the modularity of our way of modelling the machine semantics until we have tangibly illustrated what an instantiation is. The astute reader is now ready for exposure to a climax of our story of the STV family.

The modular implementation of the generic machine semantics accommodates formalising informal semantics of one same STV scheme in various ways that are all extensionally equivalent but not intentionally. In other words, the meaning of instances of computation carried out by each formalisation of the scheme can vary depending on the input data, however, the value of computation output remains the same.

For almost every STV scheme, it is possible to merge the semantics of some transition t with the semantics of another transition t'. However, it depends on the particular scheme instantiated which ones of its transitions's semantics can be merged. The merge process happens mainly by augmenting the informal (formal) clauses of a the transition t into the informal (formal) clauses of the t' semantics. For example, it is feasible to merge the semantics of the transfer elected into the elect's, and merge transfer-removed semantics into the eliminate's. We discuss how merging proceeds by exemplifying the process for integrating the semantics of the transfer-elected into the elect's in the CADE STV case.

- **List of uncounted ballots.**
  - *clause I.* The list of uncounted ballots in the pre-state is empty.

  - *clause II.* update the list of the uncounted ballots in the post-state to include all of the election ballots initially recorded to be counted.

- **Tallies.**
  - *clause III.* tallies in the pre- and post-state are the same.
- **Piles.**
  - *clause IV.* empty the pile of every candidate.
- Backlog of the elected.
  - *clause VI.* the backlog of the elected in the pre-state is empty.
  - *clause VII.* update the backlog of the elected to only consist of the elected candidate.
- **Backlog of the eliminated.**
  - *clause VIII.* the backlog of the excluded candidates in the pre- and post-state are the same.
- **Elected candidates.**
  - *clause IX.* list of the elected candidates in the post-state remains the same as in the pre-state.
- **Continuing candidates.**
  - *clause X.* the list of continuing candidates in the post-state equals to its counterpart in the pre-state.

Essentially, the CADE's transfer-elected moves the ballots from the pile of every candidate into the list of uncounted ballots. We can choose to instantiate the generic machine with CADE's formal semantics to execute the transfer-elected branch for distributing ballots in order to restart the election. Alternatively, we can combine the semantics of this transition with the elect's as we did above so that the elect branch takes care of the computation for electing and restarting the tallying. The former brings about no advantage over the latter. However, in the alternative case, one transition handles the computation for two. This results in saving time for computing winners of an election with the CADE STV. Also, recall that we intend to provide a certificate for each execution of the CADE machine on input data in order to independently verify the correctness of the instance of the computation performed. The alternative formalisation of CADE allows the certificate checker to validate or reject certificates as such more efficiently. It is obvious that extensionally, the two alternatives result in the same computation when being executed on an input. The reason is simply that the content of the semantics is the same is both alternatives. The computational content of each is a formalisation of the exact same informal clauses. This

formalisation only appears under different transition labels. Nonetheless, the meaning of the execution of each alternative may differ because the sequence of transition applications can be different.

The merging process of the transfer-elected with the elect is not complete yet. So far we have simply formalised an alternative for the elect transition. However, we have not finalised the verification of the formalised elect which requires discharging the elect's sanity checks. Actual instantiation of the machine's generic elect transition with the CADE STV's elect occurs in the latter phase where the existentially quantified variables appearing in the CADE's formalised elect are instantiated with (computational) witnesses. Understanding the instantiation of the existentially bound variables plays a crucial role in finalisation of the process. We therefore revisit the generic elect's semantics to elaborate on some more technicalities necessary to explain at this stage.

The definition of the elect semantics in Figure 3.16 specifies that for a given pre-state premise, provided such and such constraints are met, then some machine post-state conc exists to which the computation moves from the premise. The variable conc appears under an existentially bounded variable. Recall that to discharge the sanity checks, we have to instantiate all of the existentially quantified bounded variables with some (computational) entity. However, there is no more specification in the SanityCheck_Elect_App as what kind of machine state conc is and what pieces of information it encapsulates.

On the other hand, the SanityCheck_Elect_Red presented in the Figure 3.16 *hints* at which entity to instantiate for conclusion. It tells us that conclusion has to be an intermediate state. Moreover, the tally function is the same in both premise and conclusion. Every other piece of data in the conclusion may be instantiated with a witness different than its counterpart instantiated in the premise. We say "may" because other pieces of data, namely the list of uncounted ballots, pile, backlogs of the elected and eliminated, and the lists of the elected and continuing candidates in the post-state appear under bounded variables that differ from their twin in the pre-state. For example, the pile in the pre-state premise is bounded by the variable p, but the pile in the post-state conclusion is bounded by the variable np. This allows instantiating the pile in the post-state with some entity that is distinct from what we instantiate for the pile in the pre-state.

According to the above discussion, while discharging the generic machine's sanity check for the applicability of the elect, we instantiate the components of the post-state conc with computational entities that realise the clauses of the CADE's elect. Therefore, for example, we instantiate the empty list of candidates for the backlog nbl1 of the elected candidates in the post-state conc. Instantiating the backlog of the elected this way isolates execution of the transfer-elected so that it is never visited in an instance of computation with the CADE version of the machine. As Figure 3.23 illustrates, the instantiated version of the elect applies instead. The figure shows that once the list of uncounted ballots is not empty, provided there are vacancies to fill, the instantiated count transition applies to allocate the uncounted ballots to continuing candidates. Now if after this, some other candidate reaches or exceeds the quota,

again the elect branch executes followed by an execution of the count transition. Therefore, transfer-elected is completely isolated, meaning it never executes.

The astute reader correctly infers from the previous paragraph that although we opt for merging the semantics of the transfer-elected into the elect's, we still include a formalised version of it in the instantiation of the machine. So they would understandably ask why we keep it after we have merged it with another transition and made sure that it never executes. To answer the question, recall that we formalise the generic STV as a record. Therefore an instance of this record must consist of a value for every single field of which the record is made. As transfer-elected and the reducibility and applicability evidence are three fields in the generic STV, we have to instantiate them as well in order to finalise the instantiation process. To resolve this, we still formalise the transfer-elected according to the CADE instructions and actually discharge the sanity checks as well. However, we ascertain that it never executes by changing the semantics of other transitions in such a way that the pre-conditions for the transfer-elected branch of the machine are never satisfied. As exemplified above, we always keep the backlog of the elected candidates empty to guarantee by construction that transfer-elected remains isolated.

Including a merged transition in the instantiation has another motivation as well. It provides a user with an alternative to opt for separation of the transitions' semantics if they prefer this way of modelling the CADE STV. Therefore, our framework gives more choices to the user when deciding on how to extend the framework to adapt it to their application of the STV family.

From the preceding discussion another question raises, namely why we model the STV family in terms of distinctive generic transitions with separate sanity checks? Why we do not instead formalise it in terms of a monolithic inductive type with seven constructors whose names are the same as the generic transition names and each constructor incorporates the sanity check? Or why we do not model STV as a single gigantic generically encoded program and then verify the whole generic program?

At the beginning of the Section 4.2 we shortly discussed that breaking the STV into separate transitions and then realising each by distinct building blocks called clauses makes the implementation of the framework modular from which several advantages result. The above alternatives for the macro-level implementation decision suffer from some disadvantages some of which we mention as follows.

- With either of a monolithic inductive type and a gigantic generic program for formalising the STV, we lose modular implementation and modular verification of the instances. One difficulty with these two choices is that minor changes to a component of the monolithic inductive type or the gigantic generic program call for re-establishing the whole structure again because many proofs break and have to be fixed as a result of the small change made. Consequently, it lacks isolation of the encoding and verification based on the functionality of the components.

- Another problem relates to the verification process. Establishing proofs of the

instantiations correctness with an STV scheme in the cases of the monolithic inductive type and the gigantic generic program is considerably more challenging. This happens because one cannot destruct the proofs in a straightforward way into separate lemmas to deal with them one at a time. But instead, the whole instantiation has to be proven correct in a single big step.

- The above item, in turn, introduces a secondary issue; usability of the framework reduces. A user has to struggle with significantly bigger proof obligations. This unjustifiable effort may happen simply because the user may decide to use a different formula for updating the fractional value of the ballots in the surplus of an elected candidate.

- We lose easy syntactic comparison of STV algorithms which our framework facilitates. The framework as it stands allows an observer to easily compare separate parts of different STV algorithm together simply by examining the formalised counterparts of the informal clauses. But the monolithic inductive type and the gigantic generic program take this opportunity away because components are all mixed together in one single formal entity.

4.5 Extraction of Certifying Programs

We demonstrated how we carry the task of formalising an STV algorithm in the theorem prover Coq, implement a computational counterpart for the formal specification in Coq, and verify the implementation correct. Coq is an excellent environment for developing and verifying a program. It also can efficiently handle small-size elections such as the one presented in Figure 4.9 for the ACT STV. Nonetheless, Coq is not suitable for efficient computation on large-size inputs to the program. On the other hand, one of our main motivations for constructing the framework is to produce verified *efficient tools for computation* with STV algorithms. To this end, for fast computation on large-size elections, we rely on other means that Coq facilitates. We use the Coq's extraction mechanism for producing Haskell compilable versions of the Coq implementations of the STV algorithms verified in our system.

As Figure 3.1 illustrates, we instantiate the machine in separate modules and then obtain an executable program from each module. Since this process happens modularly and in a uniform manner for every STV algorithm verified in our system, we describe the process of producing the tools by exemplifying the case of the ACT STV.

4.5.1 What to Extract and How

To explain the extraction procedure, we need to revisit the Termination theorem established for the generic machine and discussed in Subsection 3.4.2. Recall from our explanation in Sections 1.5 and 4.2 that propositions are types and that proofs are actually programs. In particular, the Termination theorem for the generic machine is a program. It accepts a list of ballots bs, a non-final machine state j0, and a value s

of the record type STV as input. It then outputs a final state j where the machine terminates at for this instance of execution. It also produces a trace, the Certificate, of the machine states visited and transitions taken to reach to j.

```
Theorem Termination : forall (bs: list ballot),
 forall j0 (evj0: ~State_final j0), forall (s: STV),
  existsT j, (State_final j) * (Certificate X.st bs s j0 j).
```

Figure 4.14: Termination Property of the Machine

The Termination program is the function which we intend to extract into Haskell. We extract this function because its body (proof content) calls the applicability and the reducibility programs. By extracting the Termination we, therefore, guarantee that the general structure of any instance of computation with the machine executes according to the content of proofs established for the applicability and reducibility theorems. Hence, every execution (a) terminates and (b) terminates according to the small-step semantics of the machine.

However, the generic machine can be executed only after it is instantiated with some STV algorithm. This is because values of type STV are generic. Hence the transition labels in s, in the Figure 4.14, need to be instantiated in order to actually compute results. Also the other inputs, namely bs, j0, evj0 consisting of evidence that j0 is indeed a non-final machine state, and the number of vacancies st appearing under the Certificate must be instantiated with concrete values as well.

To obtain an executable version of the Termination theorem for the ACT STV, we instantiate the input s with the record ActSTV given in Figure 4.7. This will substitute the generic labels with ACT's and the evidence required with the ones established for ACT by discharging the sanity checks. This in turn results in obtaining Act_Termination (Figure 4.15) which is the ACT's version of the generic machine's Termination theorem. The program Act_Termination computes based on the refinement of the machine semantics with the ACT's. Therefore the meaning of computation carried out by this instantiated version of the Termination program accords with the formal semantics of the ACT STV. In light of our discussion in Sections 4.1 and 4.2, the formal semantics given for ACT corresponds with the informal description of the algorithm. Hence, the instantiated version of the Termination program with the ACT's semantics computes winners of an election as per instructions of the ACT STV protocol.

```
Variable bs: list ballot.
Lemma init_stages_R_initial : ~ State_final (initial (Filter bs)).
Definition Act_Termination :=
 M.Termination bs (initial (Filter bs)) init_stages_R_initial ActSTV.
```

Figure 4.15: Termination Property of the ACT version of the Machine

Figure 4.15 also contains information about the instantiation of other elements in the termination program. As you see, there is the letter M prefix to the name of

the termination program of the generic machine. This letter stands for the name of the instantiation module discussed in Section 3.1. As we explained in that section, creating such a module M consists of two steps;

1. specifying a value for the generically defined election parameters. The parameters are the candidates participating in the election, the number of vacancies, the quota, the function ValidBallots according to which valid ballots based on the ACT protocol are decided,

2. creating a verified copy of the base module with the specific values instantiated for the parameters[9].

The body of Act_Termination also contains an assertions about performing some filtration on the list of input ballots. The function Filters decides which ballots in the list bs are valid. The function Filter decides on the validity of the ballots based on the instantiated ValidBallots in the item 1 above. As the function Filter is already verified in Coq, we are certain that it behaves correctly so that only those ballots which are valid according to the ACT algorithm influence the end result of the election.

Figure 4.15 also contains an assertion that declares the list of input ballots bs as a variable. By specifying bs as a variable, once we extract the program into Haskell, the extracted version of the Act_Termination will have a corresponding variable for accepting a list of ballots as input. This allows us reading millions of input election ballots from a file and then executing the extracted program to compute the winners of such a large-size election.

We are now ready to extract the function Act_Termination. Figure 4.16 illustrates the two commands that take care of the extraction process. By executing the commands in Coq, the extraction tool [91] produces pretty-printed versions for the Act_Termination and every dependencies of this function.

```
Extraction Language Haskell.
Extraction "Act.hs" Act_Termination.
```

Figure 4.16: Extracting ACT Machine into a Haskell Program

4.5.2 Formal Certificates and Run-Time Certificates

The Termination program in Figure 4.14 carries its correctness proof with itself. The construct Certificate encapsulates a witness for the correctness of each and every instance of the machine execution on an input value. The Certificate is an inductively defined type as in Figure 4.17. An object of the type Certificate records all machine states visited and the transitions taken to move from one state to another in order to reach a final state j where the instance of computation halts for an input state j0 to the machine.

[9]Recall that the base module is parameterised in the record whose fields are the parameters in the item 1 above.

```
Inductive Certificate
 (st: nat) (bs: list ballot) (s: STV)
 (j0: Machine_States): Machine_States -> Type:=
 start:
 forall j, (j = j0) -> Certificate st bs s j0 j
|appInit:
 forall j1 j2, Certificate st bs s j0 j1 -> initStep s j1 j2
               -> Certificate st bs s j0 j2
|appCount:
 forall j1 j2, Certificate st bs s j0 j1 -> count s j1 j2
               -> Certificate st bs s j0 j2
|appElect:
 forall j1 j2, Certificate st bs s j0 j1 -> elect s j1 j2
               -> Certificate st bs s j0 j2
|appTransElected:
 forall j1 j2, Certificate st bs s j0 j1 -> transfer_elected s j1 j2
               -> Certificate st bs s j0 j2
|appTransRemoved:
 forall j1 j2, Certificate st bs s j0 j1 -> transfer_removed s j1 j2
               -> Certificate st bs s j0 j2
| appElim:
 forall j1 j2, Certificate st bs s j0 j1 -> elim s j1 j2
               -> Certificate st bs s j0 j2
| appHwin:
 forall j1 j2, Certificate st bs s j0 j1 -> hwin s j1 j2
               -> Certificate st bs s j0 j2
| appEwin:
 forall j1 j2, Certificate st bs s j0 j1 -> ewin s j1 j2
               -> Certificate st bs s j0 j2.
```

Figure 4.17: The Formal Inductively Defined Certificate Type

As Figure 4.17 shows, Certificate is dependent on values s of the STV type. Whever we instantiate the Termination program with a specific value for s, an object of the type Certificate consequently records, upon each execution, the states visited and transitions taken by the s version of the machine to compute the final result. For example, if we instantiate the generic Termination with the Act_STV, then we obtain the program Act_Termination which consequently creates an ACT version of the Certificate construct for recording a trace of an execution of Act_Termination.

When we extract a Haskell program for an instantiation of the machine with an algorithm s, the extracted program has a counterpart datatype called certificate which is the *concrete* Haskell version of the *formal* Certificate type. We call objects of the concrete Haskell certificate datatype as *concrete certificates* distinguishing them from the ones whose type is Certificate to which we refer as *formal certificates*. We already have seen examples of concrete certificates in Figures 4.9 and 4.13 the first of which is for an ACT STV extracted program and the second for CADE STV's.

A concrete certificate stands as proof of the correctness of the instance of the computation with the extracted Haskell program that outputs it. This is because a concrete certificate simply visualises the content of a formal certificate without tampering with the information that it carries. A formal certificate itself is just a record

of the (symbolic) execution of the instantiated version of the machine with some verified STV algorithm. In light of the previous discussion on machine instantiation, we formally verify that computation with an instantiated version of the machine happens correctly as per instructions of the algorithm formalised. Therefore, a formal certificate witnesses the correctness of a (symbolic) execution of the machine. Consequently, a concrete certificate is evidence for the correctness of the computation performed with the extracted program.

Note that the significance of a concrete certificate is beyond simply witnessing the correctness of an instance of computation. The value of a concrete certificate lies in it being independently verifiable without relying on the program that produces it. A concrete certificate is independently checkable for correctness or rejection because it encapsulates *the meaning of the computation as per the semantics* of the instantiated machine (the formal counterpart of the STV algorithm formalised). Therefore, a concrete certificate is a meaningful computationally informative object in and by itself. Moreover, it provides transparent information sufficient to know how the program has computed winners of an election using the formalised STV algorithm from which the program is synthesised.

4.5.3 Grammar of Certificates

Every certificate produced from an extracted Haskell program regardless of the STV algorithm that the program implements follows the grammar below. Later in Chapter 6 where we synthesise certificate verifiers, we use this grammar for implementing a parser. The parser parses a concrete certificate having this grammar into an abstract syntactic representation which in turn is used for executing the certificate verifier on the parsed certificate for examining the correctness of information given in the concrete certificate. Note that we place literals in between the symbols " and ".

- **Digit :=**

  $''0'' \mid ''1'' \mid ''2'' \mid ''3'' \mid ''4'' \mid ''5'' \mid ''6'' \mid ''7'' \mid ''8'' \mid ''9''$

- **Number :=**

  Digit | (Digit)*

- **Quota :=**

  Number"/"Number

- **Seats :=**

  Number

- **Char :=**

  $''A'' \mid ''B'' \mid ''C'' \mid '' D'' \mid ''E'' \mid ''F'' \mid ''G'' \mid ''H'' \mid ''I'' \mid ''J'' \mid ''K'' \mid ''L'' \mid ''M'' \mid$
  $''N'' \mid ''O'' \mid ''P'' \mid ''Q'' \mid ''R'' \mid ''S'' \mid ''T'' \mid ''U'' \mid ''V'' \mid ''W'' \mid ''X'' \mid ''Y'' \mid$
  $''Z'' \mid ''a'' \mid ''b'' \mid ''c'' \mid ''d'' \mid ''e'' \mid ''f'' \mid ''g'' \mid ''h'' \mid ''i'' \mid ''j'' \mid ''k'' \mid ''l'' \mid ''m'' \mid$
  $''n'' \mid ''o'' \mid ''p'' \mid ''r'' \mid ''s'' \mid ''t'' \mid ''u'' \mid ''v'' \mid ''w'' \mid ''x'' \mid ''y'' \mid ''z''$

- **Candidate :=**

  Char | (Char)*

- **Rational :=**

  Number"/"Number

- **Candidate list :=**

  "[]" | "[" Candidate ("," Candidate)* "]"

- **Ballot :=**

  "(" Candidate "," Rational ")"

- **Ballot list :=**

  "[]" | "[" Ballot ("," Ballot)* "]"

- **(Ballot list) list :=**

  "[]" | "[" Ballot list ("," Ballot list)* "]"

- **Tally :=**

  Candidate "{" Rational "}"

- **Tally list :=**

  " " | Tally (" " Tally)*

- **Pile :=**

  Candidate "{" (Ballot list) list "}"

- **Pile list :=**

  " " | Pile (" " Pile)*

- **Header :=**

  Quota "\n"
  Seats "\n"
  Candidate list

- **Winners :=**

  Candidate list

- **Initial :=**

  Ballot list

- **Judgement :=**

Ballot list	";"	" "
Tally list	";"	" "
Pile list	";"	" "
Candidate list	";"	" "
Candidate list	";"	" "
Candidate list	";"	" "
Candidate list	";"	" "

- **Judgement list :=**

  (Judgement)*

- **Concrete Certificate :=**

  Header "\n"
  Winners "\n"
  Judgement list "\n"
  Initial

We give an example to tangibly illustrate an instance of concrete certificate which follows the grammar. We visualise the data provided in the Figure 4.13 as shown in the Figure 4.18. The reader readily notices differences between how information is represented in the Figure 4.13 and Figure 4.18. Because of the educational purposes and ease of understandability, we earlier presented the examples of certificates given for small elections with CADE and ACT STV in a slightly different format. We choose to deviate from the communicative intentions and represent data of concrete certificates according to the above grammar mainly for the following reasons.

5/1
2
[A,B,C]
$[B, C]$
$[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$; A[2/1] B[6/1] C[6/1]; A[] B[] C[]; ([],[]); $[B, C]$; $[A]$
[]; A[2/1] B[6/1] C[6/1]; A[$[v_1, v_2]$] B[] C[$[v_3, v_4, v_5, v_6, v_7, v_8]$]; ([],[]); $[B]$; $[A, C]$
$[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$; A[2/1] B[6/1] C[0/1]; A[] B[] C[]; ([],[]); $[B]$; $[A, C]$
[]; A[2/1] B[6/1] C[0/1]; A[$[v_1, v_2]$] B[$[v_3, v_4, v_5, v_6, v_7, v_8]$] C[]; ([],[]); []; $[A, B, C]$
$[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$; A[0/1] B[0/1] C[0/1]; A[] B[] C[]; ([],[]); []; $[A, B, C]$
$[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$

Figure 4.18: A Concrete Instance of a Certificate

1. there is no horizontal line between judgement lines. Every judgement (which represents a computation state) is separated from another by the end of line symbol, meaning "\n". Therefore, when parsing a concrete certificate we already know how to distinguish them. This implies that the horizontal line is extra information consuming memory.

2. there are no labels for the transitions between one judgement to another. We have two reasons for dropping the transition labels from a concrete certificate both of which relate to the security concerns involved in the tallying phase. In Chapter 5 where we discuss certificate verifiers, we shall elaborate on this motivation in detail.

3. There is no word such as "state" or "winners" in any judgement. Including these words add no useful information to a certificate. By a careful choice of the grammar, we know that the first three lines of a certificate comprise the header part, and the fourth line is the list of candidates who have won the election and the rest of the lines are the non-final machine states except the last one which is an initial machine state.

4. Every line of the certificate starts, because of saving memory and simplifying the parsing, from the left-most column of a file which records the certificate data, whereas the examples presented earlier are aligned at the center of the page for the sake of pretty printing.

4.5.4 Experimental Results

We have evaluated[10] the extracted Haskell program from the Coq component on real historical data. The data consists of the ACT state elections for electing the representatives of the Legislative Assembly which is available online [1]. Figure 4.19 illustrates the results of executing the extracted Haskell program for ACT STV on three electoral districts of the Legislative Assembly elections held in 2012 and 2008.

electoral	ballots	vacancies	candidates	time (sec)	year
Molonglo	88266	7	40	1395	2008
Brindabella	63562	5	20	205	2012
Ginninderra	66076	5	28	289	2012

Figure 4.19: ACT Legislative Assembly 2008 and 2012

ACT Legislative Assembly has three electoral districts, namely Brindabella, Ginninderra, and Molonglo. Molonglo is the biggest electoral district of the Legislative elections in Australia both in terms of the number of competing candidates and the ballots cast. Most of the districts of the Legislative Assembly election across Australia have a number of ballots cast which is close to Brindabella and Ginninderra. Also, those districts are mainly for electing one winner while all of the ACT electoral districts have five or seven winners. This means that our software computes the winner faster if there is only one candidate to win the election. When there is only one candidate to elect, the elect transition applies just once and the transfer-elected transition never applies as the election terminates by electing one candidate. As a result, costly computation is avoided which reduces time consumed for deciding the winner.

[10]The evaluation has proceeded using an Intel Core i7-7500U CPU 2.70 GHz×4

In fact, Molonglo district for the election held in 2008 had the largest participating candidates among other instances of the Legislative Assembly held in the same district in other years. Therefore, we have included this one instead of the smaller ones, such as Molonglo 2012 for a better representation of our software capacity. As you see in Figure 4.19, the extracted Haskell program for ACT STV takes only about 22 minutes to compute the winners of the biggest Legislative Assembly district in Australia. This stands in sharp contrast to the folklore that theorem proving means are good only for verification but not for efficient computation.

We must note that we have intentionally not presented any data on how the program performs on randomly generated ballots. Randomly generated data are uniformly distributed so that no candidate has a meaningful margin of victory over any other. Consequently, in an instance of computation with such data, the elect transition almost never applies. Instead, candidates are removed so that the remaining continuing candidates are declared elected by an application of hopeful-win transition. On the other hand, elect is one of the costliest transitions because computing the transfer value that each ballot in the surplus votes of an elected draws on Coq library functions for handling rational arithmetic. The problem with this arithmetic is that all of its functions are not tail recursive. Especially, the part of the arithmetic for managing multiplication is significantly time-consuming. As a result, using them slows the computation down. Moreover, elect transition requires calling nested `map` function, also for updating the transfer value of each ballot in the surplus, which is not tail recursive. However, when the elect transition does not apply, and instead the elimination transition occurs, only the addition function is called to sum 1/1 numbers and there is no need to use nested calls to functions like `map`.

In light of the above paragraph, we cannot discuss the complexity of the whole extracted program based on randomly produced data. This is partly because of the randomly generated data is biased and therefore unreliable for analysis of our programs and also we lack enough divergent real election data. The only available data is the Legislative Assembly elections of ACT whose variations does not add any meaningful difference to what we have presented in Figure 4.19. Moreover, recall that we use the proofs-as-programs paradigm. Therefore the structure and the content of proofs also plays a significant role in the overall analysis of the program complexity. Unfortunately, use of dependent types in proof contents considerably complicates evaluation of the extracted program's complexity in a rigorous way. Since our work promotes formal reasoning as opposed to informal pen-and-paper arguments, we therefore believe that the complexity side of our programs should be left for another formal future endeavour. In Subsection 8.1.2, we discus the possibilities of improving the performance of the extracted Haskell programs.

Remark 4.5.1 *We shall note that the purpose behind our experimental result is and has never been examining whether or not any possible error has occurred in the past historical elections. We claimed that software produced from our framework is capable for deployment in real elections. The experiments given above substantiate the claim which was an obligation on our side. If a reader is interested in conducting examinations for "unveiling possible errors" in*

some past real elections whose data is accessible to them, they can simply visit the GitHub repository hosting our source code. The README file in the repository clearly instructs how to obtain executable Haskell programs which can be subsequently used for the experiments.

Chapter 5

The Generic STV Verifier

This chapter and the subsequent one together discuss the second standalone component of the framework. The component is an environment constructed for modular formalisation, verification and synthesis of tools for *verifying computation* carried out by executions of *any implementation* of an STV algorithm. We call tools produced from this component *verifiers*.

The current chapter elaborates on the following parts of the second component's story of synthesising formally verified verifiers.

1. We elaborate on the problem that the component is addressing.

2. We then present our approach and solution in an informal mathematical language. The intention for a mathematical description is to provide the reader with a clear picture of what a verifier technically resembles. This is important as the reader has to wait until they reach Chapter 6 to see the formalised version of the verifier in HOL4.

   In a nutshell, at this stage, we use our analysis of the generic STV given in Subsections 2.1.2 and 2.1.3 to formalise the generic STV as a machine for verifying computation carried out with an STV algorithm. From this newly formulated abstract machine, we devise a generic notion of verifier of computation with the generic STV. We then proceed to define what an instantiation of a machine is and use it to mathematically pin down what a specific verifier of computation for a particular STV algorithm is.

3. We then present the architecture of the framework's component to provide the reader with a holistic image of the structure of the component and the functionality of each sub-component of the structure. This also gives a roadmap of what is happening throughout the current chapter and the subsequent two chapters.

4. Having provided a vision of how the trilogy is unravelling, we continue the chapter by formally specifying in the theorem prover HOL4 a generic notion for the informally defined verifier in item 2 above.

5. We also explain how we implement a decision procedure inside HOL4 which is the computational counterpart of the formally defined generic verifier.

6. We then show in HOL4 how the specification and the implementation of the generic verifier match. We therefore establish a guarantee that the implementation is a proven correct computational realisation of its specification.

Chapter 6 details instantiations of the generic verifier and synthesis of the instantiated generic verifier to obtain machine executable specific verifiers of computation carried out based on various STV algorithms.

5.1 Generic STV for Verifying Computation

We are motivated to tackle the following problem. How can one verify that any execution of any implementation $\mathcal{P}$, whose source code may be secret, of an arbitrary known STV algorithm $\mathcal{A}$ correctly computes the end result according to $\mathcal{A}$?

Before presenting our approach, we explore some possible alternatives based on which one may hope for resolving the problem. We shall weigh the alternatives against the four general framework requirements that we discussed in the introduction of the thesis, namely correctness, verifiability of tallying, transparency and practicality.

Full static verification approach. Verifying that an implementation always returns the correct result on any input establishes a significantly high degree of guarantee in the correctness of the program and subsequently promotes its trustworthiness. However, there is a serious generic disadvantage with this approach for verifying vote-counting programs. This disadvantage exists regardless of the fact that most often the source code of the programs are kept secret, and therefore not available for full static verification, due to commercial in confidence excuses.

The problem with this approach is impracticality. It is not practical to conduct full static verification for all programs implementing different STV algorithms. Full static verification of one single implementation in a programming language alone is a Herculean task because it requires formalisation and establishing some verification for the underlying semantics of the programming language used for implementing the vote counting program. This latter by itself is a significant resource taking a process which becomes close to impossible if one intends repeating the process for verifying other programs written in other programming languages.

Also, the implementations even if happening in one same programming language can considerably vary in technicalities of the encoding such as the choice of data structure for recording data which in turn affects the mechanism of encoding the algorithm as well. This adds to the challenge of formalisation and verification of the programming language used for encoding the implementations.

Finally, all of the above has to be repeated simply by varying the STV algorithm used for counting votes. Putting the pieces together one instantly realises that despite the initial charm, the full static verification method is entirely a waste of resources.

Recomputing the tallying. Instead of the impractical full static verification of the source code of an implementation $\mathcal{P}$, one may think verifying the correctness of *instances of computation* with $\mathcal{P}$ by *recomputing* the end result using a second implementation $\mathcal{P}'$ of an STV algorithm $\mathcal{A}$. In this approach, one implements another program $\mathcal{P}'$ and executes it on the same input election parameters and the input recorded votes. Then one examines if the returned value of $\mathcal{P}'$ is the same as $\mathcal{P}$'s used for counting votes in the first place. If $\mathcal{P}$ has computed the same winners as the implementation $\mathcal{P}'$ in which one trusts, then the instance of computation with $\mathcal{P}$ has happened correctly. Otherwise, one rejects the output of this particular execution of $\mathcal{P}$ as erroneous.

Although there is more than one way of exercising the recomputation approach, however, all of them suffer from the absence of two properties, namely transparency and verifiability of the *process* of tallying with $\mathcal{P}$:

- The approach lacks *transparency* of the process through which $\mathcal{P}$ outputs the end result regardless of the correctness or incorrectness of the computation performed. With recomputing the tallying, one at best can come to attest to the correctness of the output of $\mathcal{P}$ which is the list of winners. However, the approach methodologically fails to provide *transparency in how* the program $\mathcal{P}$ has computed the, considering the best scenario, correct result.

- Also, the process of tallying is unverifiable because one can only examine the end result of the computation. There is no access to how the program $\mathcal{P}$ has computed the outcome so that one inspects the process as well. Therefore, at best, the end result of the tallying *may* be verifiable depending on how one practices the recomputation approach, but the process of computation with $\mathcal{P}$ remains unverifiable. Also, in the case when program $\mathcal{P}$ and $\mathcal{P}'$ output different winners for one same election input parameters, there is no way to know where the error in computation with the program $\mathcal{P}$ resides.

Due to lack of transparency and unverifiability of the tallying process, the recomputation approach cannot handle non-deterministic nature of STV algorithms. STV algorithms in some corner cases may use randomness to decide on how to break a tie between two equally weak candidates for eliminating one. Consequently, in two different executions of the program implementing an STV scheme, in such a situation, different votes may be transferred to continuing candidates. Hence, the set of election winners may vary. The recomputation approach inherently is unable to verify the correctness of the end result of such instances of computation.

To begin addressing the challenge in our way, assume $\mathcal{P}$ is an implementation of an STV algorithm $\mathcal{A}$. We reasonably demand each execution of $\mathcal{P}$ on a given input x consisting of votes recorded in the election to output the winners y and evidence ω as a claim for the correctness of the execution. Having evidence available for each execution of $\mathcal{P}$ facilitates checking the correctness of both the process and the end result of the computation carried out *independently* of the (source code of) $\mathcal{P}$. Therefore the original problem boils down to how one can verify the evidence of any *instance of computation* with any algorithm $\mathcal{A}$.

We verify an instance of computation by examining the evidence produced upon performing the computation. Inspecting evidence of computation for correctness happens by a decision procedure which we call *verifier*. A verifier accepts as input evidence generated for an instance of computation and checks the evidence for the correctness. If the verifier decides that the evidence is authentic, it certifies the correctness of the computation carried out to produce the evidence. In case, the verifier identifies an error in the evidence, it signals invalidity of the evidence and thereof the instance of computation performed. Figure 5.1 schematically illustrates our approach for solving the problem of verifying the computation with a program $\mathcal{P}$.

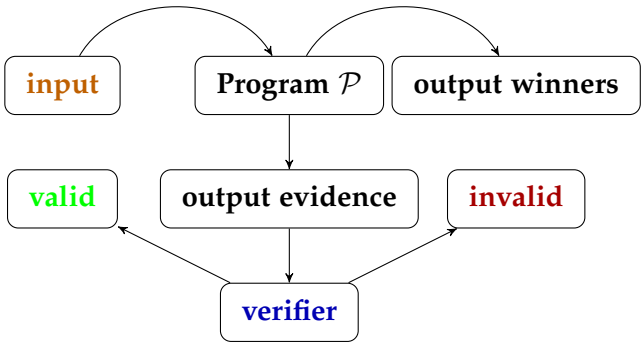

Figure 5.1: Our Approach in a Picture

There are still difficulties to overcome in order to achieve a satisfactory solution to the problem. The evidence as such must be enough informative to provide transparency of tallying and also allow voters, or at least a large pool of scrutineers, to verify themselves that the tally is authentically processed as per instructions of the counting scheme. On the other hand, we wish to verify the correctness of computation with various STV algorithms and not simply one specific STV scheme. Also, we want to verify any implementation of an STV algorithm instead of merely a particular one. We consequently face two challenges:

1. We should choose an abstract data type whose values formally represent evidence of computation with any STV algorithm, and implement a data structure that encapsulates the abstract data type. Note that the evidence must be produced for an instance of computation for a variety of STV algorithms. Therefore one challenge is deciding on a structure that is enough general for transparently accommodating data of evidence for the whole STV family without leaving any necessary information out. Also, data has to be represented according to a (formal) format. Consequently, another question regards the choice of format for representing the evidence.

2. Suppose we succeed in item 1 above and find out how to record data given in evidence. For validating or rejecting the correctness of evidence by a verifier, we need more than knowing how to store the data. We need an operational semantics to manipulate data for inspecting the evidence by the verifier. On

one hand, the semantics must be flexible enough to accommodate variations existing in the counting mechanism of STV algorithms so that we can produce verifiers specifically operating for validation of evidence output by their associated scheme. On the other hand, it must facilitate designing and developing the framework modularly while offering automation of the synthesising process of verifiers.

The real voyage of discovery consists not in seeking new landscapes, but in having new eyes[1]. Indeed, we have already come to see a perfect solution for the item 1 above. Recall that in Subsection 2.1.2, we discovered an underlying data structure for the STV family. As we explained there, an STV algorithm has a quota, some initial number of vacancies, some competing candidates. More importantly, we discussed that every instance of computation with an STV algorithm proceeds through *discrete states of computation*. The discrete states consist of all the necessary and sufficient information to know in order to transparently observe the tallying process. The computation states also provide a chance for verifiability of the process and outcome of the tallying.

It should not then come as a surprise that we already know how to answer challenges in the item 2 above as well. Remember that in Subsection 2.1.3, we detailed a general algorithmic pattern that identifies the computational commonalities among various members of the STV family. As we elaborated in the subsection, this universal algorithmic content lays the foundation of a generic vote-counting mechanism. It can be used as the semantics of an abstract machine which realises the core algorithmic characteristics of the STV family. Also, it is capable of being easily extended to capture variations existing among different STV algorithms through *instantiations* into the machine.

From the above realisation, the generic STV as a machine reemerges. However this time, it appears in disguise of a generic machine for verifying computation with the STV family. Using the machine, we formally realise the semantics of an STV algorithm $\mathcal{A}$ through an instantiation into the generic STV machine. From this discussion, we then obtain a thorough formulation of evidence for an instance of computation with $\mathcal{A}$ and also mathematically specify what a verifier for computation with $\mathcal{A}$ is.

Definition 5.1 (Machine States) *Suppose* $C \in \mathbb{S}$ *is the list of competing candidates in an election. A non-final state* nfs *is a 7-tuple construct* $(ba, ta, p, bl_1, bl_2, e, h)$ *where* $ba \in \mathcal{B}_a$, $ta \in \mathcal{T}_a$, $p \in \mathcal{P}$, *and* bl_1, bl_2, e, $h \in \mathrm{List}\ (C)$. *Moreover, a final state* fs *is an element in* $\mathrm{List}\ (C)$. *The collection of non-fnal and final states comprise all of the possible computation states which we denote by* $\mathcal{S}$.

The forms of action explained in Subsection 2.1.3 comprise the transition labels of the generic STV.

Definition 5.2 (Transition Labels) *The set of transition labels* $\mathcal{T}$ *comprises the elements* **count**, **elect**, **eliminate**, **transfer** $-$ **elected**, **transfer** $-$ **removed**, **hwin**, *and* **ewin**.

[1] Marcel Proust

Symbol	Description
Q	a rational number in the set $\mathbb{Q}$
s	a natural number in the set $\mathbb{N}$
$\mathbb{S}$	the set of all finite strings from alphabetic characters
C	a list of strings in the set $\mathbb{S}$
List (X)	the set of all finite lists of elements from X
$\mathcal{T}_a$	for representing the set List $(\mathbb{S} \times \mathbb{Q})$
$\mathcal{B}_a$	for representing the set List (List $(\mathbb{S}) \times \mathbb{Q}$)
$\mathcal{P}$	for representing the set List (List $(\mathcal{B}_a)$)

Figure 5.2: List of Symbols for characterising the Generic Verifier

Also, the common pre- and postconditions described in Subsection 2.1.3 comprise a semantics for the transition labels of the machine. They determine if and when a transition taken to move from a pre-state to a post-state happens correctly. We can mathematically express these pre- and postconditions in the syntax of Higher-order Logic to obtain a syntactic formulation for the machine as in the subsequent definition.

Definition 5.3 (The Generic STV) *Let $\mathcal{S}$ and $\mathcal{T}$ be respectively the set of machine states and transition labels. Also assume for $t \in \mathcal{T}$, $S_t = \bigwedge_{i \leq j_t} \psi_i$ where ψ_i is the formal specification (in Higher-order Logic) of a pre- or postcondition of t. Then the generic STV is the triple $M_{spec} = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$.*

As we have already seen, there are variations in the pre- and postconditions of STV algorithms. We recognise these existing differences in the counting mechanisms of STV schemes by separate *instantiations* into the generic machine as in Definition 5.4. Recall that the semantics of the machine is formed by conjunctions of formally specified universally appearing pre- and postconditions in STV algorithms. An instantiation of the machine with an STV algorithm $\mathcal{A}$ happens by enriching the generic semantics by adding formal specifications of the legal clauses that are specific to $\mathcal{A}$. We hence come to define an operational semantics functioning in accordance with instructions of $\mathcal{A}$.

Definition 5.4 (Machine Instantiation) *Assume the generic machine $M = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ in Definition 5.3. A machine instantiation with the algorithm $\mathcal{A}$ is the triple $\hat{\mathcal{A}} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t \in \mathcal{T}} \rangle$, where for each $t \in \mathcal{T}$, $(S'_t) = S_t \wedge \bigwedge_{i \leq j'_t} \phi_i$. Each ϕ_i is the formal specification of a pre- or postcondition specific to $\mathcal{A}$.*

We are now ready to mathematically define what evidence of an instance of computation with an STV algorithm $\mathcal{A}$ is. Informally speaking, such evidence records the (discrete) states visited during execution of an instantiation of the generic machine by $\mathcal{A}$ in order to compute the winners of an election which uses $\mathcal{A}$ as its vote-counting scheme.

Definition 5.5 (Formal Evidence) *Suppose $\mathcal{A}$ is the vote-counting scheme used in an election where the $Q \in \mathbb{Q}$ is the quota, $s \in \mathbb{N}$ is the number of initial vacancies, and $C \in \mathbb{S}$ is the list of competing candidates. Also assume $\hat{\mathcal{A}} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t \in \mathcal{T}} \rangle$ is the instantiation of $\mathcal{A}$ in the generic machine as defined in Definition 5.4. Finally, assume $y \in$ List (C) is the output winners of an (symbolic) execution of $\hat{\mathcal{A}}$ on an input value $x \in \mathcal{B}_a$. By formal evidence for this instance of computation with $\hat{\mathcal{A}}$, we mean the quadruple $\hat{\omega} = (Q, s, C, \hat{\Omega})$, where $\hat{\Omega} \in$ List $(\mathcal{S})$. In other words, $\hat{\Omega}$ is a list of machine states visited in order to reach form the state where x is registered to the final state where y is resulted.*

The definition of evidence of an instance of computation with an algorithm $\mathcal{A}$ in Definition 5.5 is a mathematical concept. In contrast, an implementation $\mathcal{P}$ of the algorithm $\mathcal{A}$ cannot produce an object representing such a mathematical construct. Therefore, our characterisation of the notion of evidence has to become *concrete*. Representation of information in concrete evidence has to follow a (formal) format otherwise we will not be able to devise a uniform modular process for validating them. To this end, we demand that every instance of evidence should follow the grammar given in Subsection 4.5.3 except that no evidence needs to have a line for recording an initial machine state. In Chapter 6 when we present the formalised definition of a verifier, we discuss the reasons behind this exception.

Remark 5.1.1 *The above definition encapsulates the notion of evidence regardless of whether or not it is a valid witness for the instance of computation c carried out with the algorithm $\mathcal{A}$. Validity or invalidity of evidence is decided by the verifier of the algorithm $\mathcal{A}$.*

Remark 5.1.2 *The astute reader may have already attended to a resemblance, namely that our identification of evidence as in Definition 5.5 indeed matches with what we earlier called* certificate *in the preceding chapters. The only difference is that we used the word "certificate" in relation to the programs extracted from the Coq component of our framework. As our motivation for developing the second framework's component is to provide transparency of computation and independently verifiability of tallying with* any *program implementing an STV algorithm, we chose the word "evidence" instead of "certificate". This avoids creating the wrong impression that our discussion in the current chapter and the subsequent ones are just limited to verifying computation carried out by programs produced from the first component.*

Having eliminated the possible misunderstanding of our intentions, from this point on we shall use the word "certificate" interchangeably with the word "evidence". We adopt this terminology because the noun "evidence" is uncountable. Using the word "evidence" may cause confusion for the reader when the text is employing the word as a singular or as a plural noun. However, with "certificate" we can easily make the distinction stand out as the plural of this word is simply "certificates".

The operational semantics of each instantiation allows performing operations such as validation on evidence. Definition 5.7 lays down what verifying evidence of an instance of computation amounts to. Informally speaking, to check if evidence is

valid one simply needs to inspect if transitions between every two consecutive states of computation appearing in the evidence occur by a legitimate application of a counting action of the scheme used. In other words, a verifier is simply a Boolean-valued decision procedure which recursively accepts a pre-state and a consecutive post-state of an instance of computation and checks if the piece of information in the post-state have been computed based on the data in the pre-state and in accordance with the instructions of the algorithm used to generate the evidence.

Definition 5.6 *Assume* $\hat{\mathcal{A}} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t\in\mathcal{T}} \rangle$ *is an instantiation of the STV algorithm* $\mathcal{A}$. *The function* ValidStep *is a mapping from* $\mathcal{S} \times \mathcal{S}$ *into {*true, false*} where for given* $\hat{\Omega}_0$, $\hat{\Omega}_1 \in$ List $(\mathcal{S})$,

$$\text{ValidStep } (\hat{\Omega}_0, \hat{\Omega}_1) = \bigvee\nolimits_{t\in\mathcal{T}} (\vdash \mathcal{S}'_t[\hat{\Omega}_0, \hat{\Omega}_1])^2.$$

where $\bigvee_{t\in\mathcal{T}} (\vdash \mathcal{S}'_t[\hat{\Omega}_0, \hat{\Omega}_1])$ *means that the transition from the pre-state* $\hat{\Omega}_0$ *to the post-state* $\hat{\Omega}_1$ *happens only when the semantics' requirements of some transition label t is logically true.*

Definition 5.7 (verifier) *Assume* $\hat{\mathcal{A}} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t\in\mathcal{T}} \rangle$ *is an instantiation of the STV algorithm* $\mathcal{A}$. *A verifier for instances of computation with* $\mathcal{A}$ *is a function* $\hat{\mathcal{V}}$ *mapping from* $\mathbb{Q} \times \mathbb{N} \times$ List $(C) \times$ List$(\mathcal{S})$ *to {*true, false*} such that for any evidence* $\hat{\omega} = (Q, s, C, \hat{\Omega})$ *where* $\hat{\Omega} = \langle \hat{\Omega}_1, \ldots, \hat{\Omega}_n \rangle$, $\hat{\mathcal{V}}(\hat{\omega}) =$ true *if and only if the following holds:*

$$\forall i \in \{1, \ldots, n-1\}.\ \text{ValidStep } (\hat{\Omega}_{i-1}, \hat{\Omega}_i)$$

$\hat{\Omega}_{i-1}$ *is a pre-state and* $\hat{\Omega}_i$ *is a post-state visited in the execution.*

From the above, we see why the generic STV appears this time as a machine for verifying computation performed by an STV algorithm, instead of a machine for carrying out the computation. The machine is here used in implementing verifiers only for inspecting the authenticity of the information given in the evidence of an instance of computation, but not regenerating the evidence or the computation. A consequence of this different role that the generic STV is playing is that we do not need to prove general properties about the generic STV as a machine for verifying computation or any particular STV algorithm. For example, there is no need to prove the termination property because the evidence is a finite piece of data and the fact that it is already output means the instance of computation has already terminated. However, what we need to establish is a guarantee that the implementation of a verifier as a decision procedure, checks the correctness of the information given evidence as per instructions of the algorithm used.

Note that the definition of a verifier of an algorithm given in Definition 5.7 is also an abstract notion. We need an implementation of this abstract construct that can accept as input a concrete certificate and perform the validation process on it. In the next section, we shall elaborate on the whole structure of this framework and also provide a general picture informing the reader how using CakeML we synthesise a machine executable verifier from the formalised version of a verifier in the theorem prover HOL4.

[2]The symbol $\vdash$ stands for logical truth.

5.2 The Architecture of the Second Component

We progress through three macro-level phases to develop the framework.

- *Step 1.* This step itself develops according to the following cycle.
  - *specifications.* We formalise the content of the Definitions 5.3, 5.4, and 5.7. This amounts to specifying the pre- and postconditions of the generic STV and also the semantic constraints that are particular to an STV algorithm being instantiated into the machine. Also, we give a specification of verifiers for an instantiated algorithm. Moreover, we specify every auxiliary assertion needed for the formalisation of the generic STV's semantics and that of its instantiations with various STV schemes.
  - *implementations.* In the theorem prover HOL4, for each of the specifications above, we implement a decision procedure that decides whether or not a given input value is valid. We assemble these implementations together in order to first obtain an implementation of the machine instantiation with an algorithm $\mathcal{A}$ and then synthesise a certificate verifier for $\mathcal{A}$.
    In particular, we implement a decision procedure for the generic machine. Technically, an implementation of the generic machine is a triple $M_{dec} = \langle \mathcal{S}, \mathcal{T}, (S_t)^{dec}_{t \in \mathcal{T}} \rangle$ where $(S_t)^{dec}$ is the implementation of a decision procedure for the specification of the transition t's semantics.
  - *verification.* We demonstrate that every decision procedure given above correctly implements its specification. For example, we formally prove in HOL4 that for every generic transition, the implementation of t's semantics matches with its implementation, i.e. $\forall\ t \in \mathcal{T},\ (S_t)^{dec} \Leftrightarrow\ (S_t)^{spec}$.
- *Step 2.* we use the verified proof translation tool of CakeML to provably correctly translate implementations of the first phase into equivalent CakeML implementations.
- *Step 3.* The last phase uses the ecosystem of CakeML and its verified compiler to generate machine executable evidence verifiers from the translated implementations in the preceding step.

We break the implementation of the above steps into several modules. Figure 5.3 is a simplified schematic illustration of the framework modules and their dependencies. We explain the purpose of each and then describe how they collectively function.

- **Auxiliary.** The module contains specification, implementation and verification of the generic STV. It consists of subdivisions for the formal specification of the machine and its components, implementation of functions meant to be the computational counterparts of the specifications and verification of the implementations by proving that they logically satisfy their respective specifications. Moreover, we include specification, implementation and verification of helper

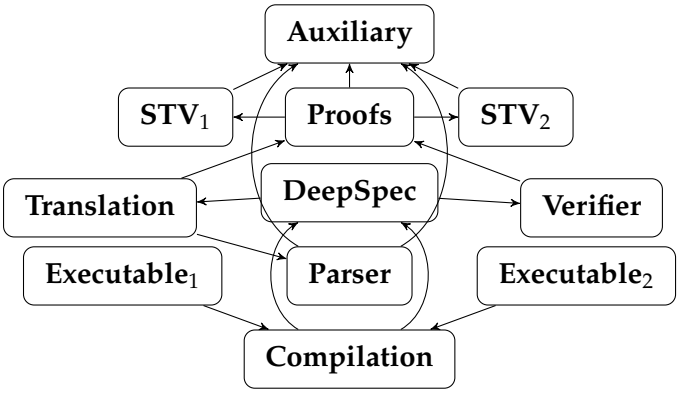

Figure 5.3: Architecture of the Framework, note that direction of arrow represents module dependencies

assertions which appear in many well-known STV schemes here so that Auxiliary also serves as a comprehensive library for STV.

- **Instantiation.** Instantiations of the generic STV happen in separate modules which we have schematically shown only two, namely **STV**$_1$ and **STV**$_2$. An instantiation module **STV**$_i$ for an algorithm $\mathcal{A}$ consists of a subdivision for the specification $\hat{\mathcal{A}}_{spec}$ of the scheme and another one its implementation $\hat{\mathcal{A}}_{dec}$.

  Mathematically speaking, an implementation of a machine instantiation with $\mathcal{A}$ is also a triple $\hat{\mathcal{A}}_{dec} = \langle \mathcal{S}, \mathcal{T}, (S'_t)^{dec}_{t \in \mathcal{T}} \rangle$, where $(S'_t)^{dec}$ is the implementation of a decision procedure for the instantiation of the transition t's semantics with pre- or postconditions particular to $\mathcal{A}$.

- **Proofs.** We prove that for an algorithm $\mathcal{A}$ in the above modules, $\hat{\mathcal{A}}_{dec}$ is logically equivalent to $\hat{\mathcal{A}}_{spec}$, i.e. $\forall\ t \in \mathcal{T}.\ (S'_t)^{spec} \Leftrightarrow (S'_t)^{dec}$. We automate proofs so that they work for various instantiations.

- **Verifier.** This module formally realises the notion of the verifier in Definition 5.7. However, it is developed in a way that automatically operates regardless of which STV algorithm is instantiated into the machine. It has three subdivisions. A specification $\hat{\mathcal{V}}_{spec}$ defining what a verifier is, the implementation $\hat{\mathcal{V}}_{dec}$. We also prove that the implementation of the verifier satisfies the logical assertions in the specification of the verifier.

- **Translation.** We translate each implementation $\hat{f}_{dec}$ in HOL4 to a proven equivalent implementation f_τ in CakeML's environment by using the verified proof translator of CakeML. For example, an implementation $\hat{\mathcal{A}}_{dec}$ of a scheme $\mathcal{A}$ and its verifier $\hat{\mathcal{V}}_{dec}$ are respectively translated into $\mathcal{A}_\tau$ and $\mathcal{V}_\tau$.

- **DeepSpec.** For any translated verifier $\mathcal{V}_\tau$ and the translated evidence parser $\mathcal{P}_\tau$, we obtain logically equivalent deep CakeML embeddings $\mathcal{V}^*$ and $\mathcal{P}^*$, respectively. We then use them to establish end-to-end desired properties, based

on CakeML's I/O model, about the interaction of the executable verifier with its hosting operating system.

- **Compilation.** Instantiations of CakeML's compiler for generating machine executable verifiers happen in this module. For any deeply embedded $\mathcal{V}_i^*$, corresponding to $\hat{\mathcal{V}}_{dec}$ in $\mathbf{STV}_i$, using proofs established in **DeepSpec**, we instantiate the compiler to synthesise executable verifier $\mathcal{V}_i$.

5.3 The Generic Machine in HOL4

We proceed through a cycle of specification, implementation and verification of implementations in HOL4 to formally verify the generic machine and its components. We have two motivations for including declarative specifications as part of the development cycle.

- Recall that we require minimising the trusted computing base for computing with generated verifiers. The specifications put together with proofs which we establish for the correctness of the implementations against their specification help us eliminate a trusted layer in the verifier's synthesis process. By including logical specifications, we can verify the correctness of the implementations in HOL4 so that there is no need to lay trust in this phase of the development.

- We also have another motivation for developing the auxiliary module through this three-part process of specification, implementation and then verification. Familiarity, but no expertise, with our framework and its general purpose, is needed for extending it to synthesise an executable verifier for one's desired STV scheme. On the one hand, an average user may lack enough skills in grasping functional programming style which we rely on for implementing verifiers in HOL4. However, the same user most probably has had exposure to formulations of mathematical properties in first-order logic syntax. Including purely descriptive logical assertions as our specifications, proven equivalent to their implementations, facilitates the users with means to understand the functionality of components, modules and the system as a whole simply by inspecting the specifications instead of implementations.

5.3.1 Data Structure of Machine States

We choose the data structure given in Definition 5.8 for implementing the abstract data type underlying the generic STV. We have four reasons for this choice of data structure.

- As the abstract syntactic representation of evidence closely represents concrete evidence, designing the parser and inspecting its correctness is less challenging.

- HOL4 has well-developed libraries comprised of verified assertions of operations on list structure. By structuring the data on lists, we facilitate us and users

```
val _ = Datatype `
  judgement =
    NonFinal (
     (* ballots    *)              ballots #
     (* tallies    *)              tallies #
     (* piles      *)              piles #
     (* backlog of elected   *)    (cand list) #
     (* backlog of removed *)      (cand list) #
     (* elected    *)              (cand list) #
     (* continuing *)              (cand list) )
  | Final
    (* winners *) (cand list)`;
```

Figure 5.4: Data Structure of the Generic Machine in HOL4

with an exploitation of already verified tools to formalise the framework and avoid inventing the unnecessary from scratch.

- HOL4 also has well-developed tactics for discharging proof obligations on assertions involving list structure and operations on it. We therefore come to provide us with means for ease of verification of the formalised assertions.
- Understanding the data structure used in the framework implementation is critical for ease of its usability and extensibility by third parties. The type *judgement* closely models concrete evidence such as the one in Figure 4.18. Therefore one can sensibly perceive the abstraction step taken for modelling concrete data which in turn enhances understandability of the framework and its mechanism.

Definition 5.8 *We formalise machine states as an inductive type judgement whose constructors are NonFinal and Final as in Figure 5.4:*

A Final (w) *judgement declares* w *as winners of the election. A NonFinal* $(ba, t, p, bl_1, bl_2, e, h)$ *judgement consists of uncounted ballots* ba*, tally* t*, pile* p*,* bl_1 *the backlog of elected candidates,* bl_2 *the backlog of eliminated candidates, and* e *and* h *for the list of elected and continuing (hopeful) candidates. The types ballots, tallies and piles are respectively abbreviations for* $((cand\ list) \times rat)list$*,* $(cand \times rat)\ list$ *and* $(cand \times (ballots\ list)\ list)\ list$ *where rat is the HOL4 type of fractional numbers.*

Recall that objects whose type is *piles* serve as containers for recording the ballots allocated to each candidate. As we described in Section 3.2, our choice for type of piles allows us to store the ballots received by each candidate in chunks of lists rather than one single list containing all of them. Also we explained that this way of implementing the data structure makes it possible for the framework to accommodate formalisation of a larger class of STV algorithms, and therefore a variety of certificate verifiers.

There is also a reason for choosing the type *rat* instead of e.g. floating numbers. Elections with an STV counting scheme have a small margin of victory especially

for the last vacancy left to fill [18]. This margin may happen to be less than the magnitude of the error caused by the accumulation of rounding errors as it is with floating points. Hence one must sensibly avoid the high cost of electing a wrong candidate by falling into traps of imprecise calculations due to unintelligent choices. Using exact fractions allows safe handling of calculations.

5.3.2 Specification, Implementation and Verification of the Small-Step Semantics

As we elaborated in Subsection 2.1.3, the semantics of each machine transition label consists of conjunctions of formally specified general pre- and postconditions across STV schemes. The conditions enforce when and how to take a tallying action and what the immediate effect is.

To demonstrate the process of specification, implementation and verification of the machine transitions and their semantics, we discuss the transfer-elect transition. We choose transfer-elect for two reasons. The first is that the role it plays in the STV mechanism has a close relationship with that of elect transition. Second, we already have seen in Chapter 4 how this process proceeds for the elect transition. Therefore, here we avoid duplication in our discussion and instead elaborate on another part of the narrative which presents another part of the STV story.

STV schemes *explicitly* declare four conditions that must be satisfied for any legitimate application of transfer-elected;

a. there are still vacancies to fill

b. There are no uncounted ballots to deal with

c. there is no vote from any eliminated candidate still awaiting distribution, and

d. There are surplus votes of elected candidates to transfer.

e. No continuing candidate in the pre-state has reached or exceeded the quota.

Also, there are some *implicit* conditions present in legal documents describing transfer-elect. These constraints come to attention either as the result of a straightforward understanding of the explicit conditions above or as auxiliary components that are taken for granted by legislators but are necessary for proper functioning of the explicit constraints;

c_1. every candidate in the pre- and post-state of transfer-elected has a unique tally and pile

c_2. no one is elected or eliminated by applying transfer-elect

c_3. no candidate attracts any new vote by transfer-elect and therefore tallies remain the same

c_4. any elected candidate is no longer a continuing candidate so that they do not receive votes any further

```
val TRANSFER_Auxiliary_def = Define `
  TRANSFER_Auxiliary ((qu,st,l):params) (ba: ballots) (t: tallies) t'
                          (p: piles) (p': piles) (bl: cand list) (bl2: cand list)
                          (bl2': cand list) e e' h h' <=>
        (bl2 = [])
     /\ (bl <> [])
     /\ (ba = [])
     /\ (bl2' = [])
     /\ (t' = t)
     /\ (e' = e)
     /\ (h' = h)
     /\ (LENGTH e < st)
     /\ (!d. MEM d (h++e) ==> MEM d l)
     /\ (!d. MEM d bl ==> MEM d l)
     /\ ALL_DISTINCT (h++e)
     /\ (Valid_PileTally t l)
     /\ (Valid_PileTally p l)
     /\ (Valid_PileTally p' l)
     /\ (Valid_Init_CandList l)
     /\ ALL_DISTINCT (MAP FST t)
     /\ ALL_DISTINCT (MAP FST p)
     /\ (!c'. (MEM c' h ==> (?x. MEM (c',x) t /\ ( x < qu))))`;
```

Figure 5.5: The Generic Transfer-elect transition

c_5. the list of competing candidates in the election is not empty and has no duplication of names

c_6 there is no duplication of candidates' names in either of the list of elected and continuing candidates in the pre- and post-state.

c_7 every candidate's name appearing in the backlog of the elected candidates also appears among the initially declared competing candidates.

Conjunctions of formal declarations of the above explicit and implicit conditions form the semantics of the generic transfer-elect transition. The predicate TRANSFER_Auxiliary in Figure 5.5 defines the semantics of this transition. As is the case for other formalised generic transitions, TRANSFER_Auxiliary is parameterised by the quota qu, the number of initial vacancies st and the list l of all candidates competing in the election. The semantics of the transition declares that given the ballots ba, tally t, pile p, backlog of elected bl, backlog of eliminated bl_2 and the list of elected e and continuing candidates h in the pre-state of TRANSFER_Auxiliary, and their respective counterparts in the post-state characterised by having a prime symbol, some conditions as specified above are satisfied by the transition.

Figure 5.6 shows which clauses in the explicit and implicit conditions match with which lines of the formal counterpart of the generic transfer-elected transition. We continue by demonstrating how the formal specification of each clause realises the informal one. We also detail the implementation of some of the formal specifications

Clause	Line(s) in Figure 5.5	Clause	Line(s) in Figure 5.5
a	83	c_2	81-82
b	78	c_3	80
c	79	c_4	86
d	77	c_5	90
e	93	c_6	86
c_1	87-90	c_7	85

Figure 5.6: Correspondence of the Informal Clauses with the Formalised Ones

and illustrate how we verify the implementations against their respective specification.

Formalisation and verification of a vote-counting algorithm require attention to small details which are often taken as obvious. For example, in every election, there are candidates who are competing for vacancies. Also, the name of any candidate participating in the election appears only once in the list of all competing candidates announced to the public and appearing on the ballots. We formalise these two trivial properties through the predicate Valid_Init_CandList (Figure 5.7).

```
⊢ ∀l. Valid_Init_CandList l ⟺ l ≠ [] ∧ ALL_DISTINCT l
```

Figure 5.7: Specification of the Checks for the List of Competing Candidates

Also in every non-final state of computation, every candidate has only one tally and pile. To realise this simple requirement, we define the predicate Valid_PileTally.

```
⊢ ∀t l. Valid_PileTally t l ⟺ ∀c. MEM c l ⟺ MEM c (MAP FST t)
```

Figure 5.8: A Specification of Allocation of Piles and Tallies

Recall that tallies are pair objects where the first component is a candidate's name and the second one is a rational number standing for the amount of votes he/she has attracted. The predicate Valid_PileTally declares that every first component of a tally (pile) list appears in the list l consisting of the announced competing candidates in the election. Conversely, the predicate also states that every competing candidate in the election has a tally (pile).

The uniqueness of the tally and pile allocated to every candidate at each non-final state results from attending to two facts. First, note that the initial list of competing candidates has no duplication as declared by the predicate Valid_InitCandList. Second, as established by the predicate Valid_TallyPile, the first components of a tally (pile) object belong to the list of competing candidates and vice versa.

We implement two Boolean-valued functions that together implement the specification Valid_PileTally. The first function in Figure 5.9 decides if every element in the list of competing candidates happens as the first component of a pair in a tally (pile) list.

```
val Valid_PileTally_dec1_def =
   ⊢ (∀l. Valid_PileTally_dec1 [] l ⟺ T) ∧
     ∀h t l.
         Valid_PileTally_dec1 (h::t) l ⟺
         MEM (FST h) l ∧ Valid_PileTally_dec1 t l
```

Figure 5.9: Implementation of Valid_Pile_Tally Part I

The function Valid_PileTally_dec2 checks if every element appearing as the first component of a pair in a tally (pile) list belongs to the list of competing candidates.

```
⊢ (∀t. Valid_PileTally_dec2 t [] ⟺ T) ∧
  ∀t l0 ls.
      Valid_PileTally_dec2 t (l0::ls) ⟺
      if MEM l0 (MAP FST t) then Valid_PileTally_dec2 t ls else F
```

Figure 5.10: Implementation of Valid_Pile_Tally Part II

We prove that the specification Valid_Piletally matches with the two functions that implement it. Therefore, the correctness of the implementation is witnessed by the proof established for them, instead of trusting the implementation.

```
⊢ ∀l t.
      Valid_PileTally t l ⟺
      Valid_PileTally_dec1 t l ∧ Valid_PileTally_dec2 t l: thm
```

Figure 5.11: Proof of the Correctness of Valid_PileTally Implementation

We need functions that look through a given tally list and find the amount of votes that a candidate has received. For this, we implement a decision procedure called get_cand_tally that accepts a tally list and a candidate's name as input and returns the rational number which is the amount votes received by the candidate.

We prove (Figure 5.13) that indeed every candidate has only one tally so that the implementation of the function get_cand_tally satisfies the expectations of the tally list specification.

Using the get_cand_tally function, we implement (Figure 5.14) the decision procedure less_than_quota which decides if every continuing candidates' tally in a given state of computation is below the quota. To guarantee the correct behaviour of less_than_quota, we prove two main properties about it. The first one (Figure 5.15) asserts that the function computationally realises its specification.

The second proof in the Figure 5.15) demonstrates that less_than_quota logically forces its specification. Therefore, we come to establish a match between the implementation of the specification of this Boolean-valued function.

Recall that line 93 formalises the clause (e) of the informal semantics of the generic transfer-elected. Drawing on the proofs presented in Figures 5.15, we conclude that the implementation of less_than_quota matches with the expectation of the generic transfer-elected instructions.

```
⊢ ∀c ls.
      get_cand_tally c ls = case ALOOKUP ls c of NONE => -1 | SOME r => r
```

Figure 5.12: A Boolean-valued Function for Finding Tally of Candidates

```
val EVERY_CAND_HAS_ONE_TALLY =
   ⊢ ∀t c x. ALL_DISTINCT (MAP FST t) ∧ MEM (c,x) t ⇒ get_cand_tally c t = x:
   thm
```

Figure 5.13: Uniqueness of Candidate's Tally in Implementation

In a similar manner, we define Boolean-valued functions that implement a condition given in the specification TRANSFER_Auxiliary. Conjunctions of these Boolean-valued computational assertions form an implementation of the transfer-elected semantics as in Figure 5.16.

Eventually, by drawing on the proofs of equivalence established between the components of the transfer-elected semantics as a declarative specification (Figure 5.5) and as a computational implementation (Figure 5.16), we prove (Figure 5.17) that the implementation TRANSFER_Auxiliary_dec matches with the specification TRANSFER_Auxiliary.

Similarly, we first proceed to obtain an informal specification of the semantics of other transitions. We then specify the informal semantics in HOL4 and implement decision procedures for each specification. We then prove the equivalence between the specifications and the implementations so that the decision procedures later used for the synthesis of verifiers are provably correct computational entities. In particular, we formally prove a theorem, similar to the one in Figure 5.17, for the match between each transition's implementation with its specification.

Theorem 5.1 *Assume* $M^{spec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{spec})_{t\in\mathcal{T}} \rangle$ *is the specification of the machine and* $M^{dec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{dec})_{t\in\mathcal{T}} \rangle$ *is its computational implementation. Then for any* $t \in \mathcal{T}$, $\mathcal{S}_t^{spec} = \mathcal{S}_t^{dec}$.

In the next subsection, we provide an example to exemplify how the generic verifier operates on data for validating concrete certificates.

5.3.3 Verifying a Sample Concrete Certificate

One can perform experiments with the Boolean-valued implementation of the generic verifier on concrete certificates inside the environment of HOL4. We give two examples to illustrate how the generic verifier processes information on concrete certificates.

Figure 5.18 consists of the preferences recorded in an election which uses the ACT STV as its vote-counting mechanism. The election parameters consisting of the quota, number of vacancies, the candidates competing in the election and preferences cast in the election match with the ones in Figures 4.8 and 4.9. Therefore, the reader can refer to the Subsection 4.2.3 to see how the winners of this election are computed.

```
val less_than_quota_def =
   ⊢ ∀qu l ls.
        less_than_quota qu l ls ⟺ EVERY (λh. get_cand_tally h l < qu) ls
```

Figure 5.14: A Function for Deciding if Everyone is Below the Quota

```
   ⊢ ∀qu t h.
        (∀c. MEM c h ⇒ ∃x. MEM (c,x) t ∧ x < qu) ∧
        ALL_DISTINCT (MAP FST t) ∧ (∀c''. MEM c'' h ⇒ MEM c'' (MAP FST t)) ⇒
        less_than_quota qu t h: thm
   ⊢ ∀h t0 t1 qu.
        less_than_quota qu (t0::t1) h ∧ Valid_PileTally_dec2 (t0::t1) h ⇒
        ∀c. MEM c h ⇒ ∃x. MEM (c,x) (t0::t1) ∧ x < qu: thm
```

Figure 5.15: less_than_quota Computationally Realises Its Specification

```
val TRANSFER_Auxiliary_dec_def =
   ⊢ ∀qu st l ba t t' p p' bl bl2 bl2' e e' h h'.
        TRANSFER_Auxiliary_dec (qu,st,l) ba t t' p p' bl bl2 bl2' e e' h h' ⟺
        NULL bl2 ∧ e' = e ∧ h' = h ∧ t' = t ∧ LENGTH e < st ∧
        list_MEM_dec (h # e) l ∧ list_MEM_dec bl l ∧ ALL_DISTINCT (h # e) ∧
        Valid_PileTally_dec1 t l ∧ Valid_PileTally_dec2 t l ∧
        Valid_PileTally_dec1 p l ∧ Valid_PileTally_dec2 p l ∧
        Valid_PileTally_dec1 p' l ∧ Valid_PileTally_dec2 p' l ∧ ¬NULL l ∧
        ALL_DISTINCT l ∧ ALL_DISTINCT (MAP FST t) ∧
        ALL_DISTINCT (MAP FST p) ∧ less_than_quota qu t h ∧ ¬NULL bl ∧
        NULL ba ∧ NULL bl2': thm
```

Figure 5.16: Implementation of the Generic Transfer-Elected's Semantics

```
⊢ TRANSFER_Auxiliary_dec = TRANSFER_Auxiliary: thm
```

Figure 5.17: Implementation and Specification of Generic Transfer-Elected's Semantics Match

Here, we demonstrate how the generic verifier validates the certificate produced for that instance of election. In particular, we focus on validating the transfer-elected transition.

Figure 5.19 shows the concrete certificate produced for the instance of computation with the ACT STV in this election. In Figure 5.19, there are two coloured lines. One line is in blue and the other is in orange color[3]. We discuss how the transition from the machine state in orange to the one in blue occurs by a correct application of the TRANSFER_Auxiliary_dec inside the environment of HOL4.

Recall that the Definition 5.7 specifies it as a Boolean-valued procedure where one valid step has to occur in order to move from one machine state to the next. The valid step in Definition 5.6 declares that such a move from one state to another is legitimate only if it happens according to the semantics of a transition label. To this

[3]In case the reader is reading a back and white hard-copy of the chapter, the blue line is the 7th line from top to the bottom of the certificate and the one is orange is the 8th one from the same direction,

Ballot	Preference	Ballot	Preference
b_1	$[A, B]$	b_5	$[C, B, A]$
b_2	$[A, B, C]$	b_6	$[B]$
b_3	$[B, C, A]$	b_7	$[B, C]$
b_4	$[B, A]$	b_8	$[]$

Figure 5.18: Ballots' Preferences of a Sample Election with ACT STV

10/3

2

[A,B,C]

$[B, A]$

$[(b_5,1/1),(b_3,2/3),(b_7,2/3)]$; $A\{8/3\}$ $B\{4/1\}$ $C\{7/3\}$; $A[[(b_1,1/1),(b_2,1/1)],[(b_4,2/3)]]$ $B[]$ $C[]$; ([],[]); $[B]$; $[A]$

$[]$; $A\{8/3\}$ $B\{4/1\}$ $C\{7/3\}$; $A[[(b_1,1/1),(b_2,1/1)],[(b_4,2/3)]]$ $B[]$ $C[[(b_5,1/1)],[(b_3,2/3),(b_7,2/3)]]$; ([],[]); $[B]$; $[A, C]$

$[(b_3,2/3),(b_4,2/3),(b_6,2/3)]$; $A\{5/2\}$ $B\{4/1\}$ $C\{1/1\}$; $A[[(b_1,1/1),(b_2,1/1)]]$ $B[]$ $C[[(b_5,1/1)]]$; ([],[]); $[B]$; $[A, C]$

$[]$; $A\{2/1\}$ $B\{4/1\}$ $C\{1/1\}$; $A[[(b_1,1/1),(b_2,1/1)]]$ $B[[(b_3,2/3),(b_4,2/3),(b_6,2/3),(b_7,2/3)]]$ $C[[(b_5,1/1)]]$; ([B],[]); $[B]$; $[A, C]$

$[]$; $A\{2/1\}$ $B\{4/1\}$ $C\{1/1\}$; $A[[(b_1,1/1),(b_2,1/1)]]$ $B[[(b_3,1/1),(b_4,1/1),(b_6,1/1),(b_7,1/1)]]$ $C[[(b_5,1/1)]]$; ([],[]); []; $[A, B, C]$

$[(b_1,1/1),(b_2,1/1),(b_3,1/1)$ $(b_4,1/1),(b_5,1/1),(b_6,1/1),(b_7,1/1),(b_8,1/1)]$; $A\{0/1\}$ $B\{0/1\}$ $C\{0/1\}$; $A\{\}$ $B\{\}$ $C\{\}$; ([],[]); []; $[A, B, C]$

Figure 5.19: A Concrete Certificate of an Election

end, the verifier which recursively calls the ValidStep function checks to see which one of the transitions may have happened to make the move.

Assume the generic verifier validates all the steps happening before the orange line as correct. It is time now to check if the transition from the orange line to the blue one is also correct. For this, the verifier tries the Boolean-valued decision procedure TRANSFER_Auxiliary_dec. Every conjunction in the definition of TRANSFER_Auxiliary_dec given in Figure 5.16 has to return the Boolean-value true in order to certify that the transition taken from the orange line to the blue one is correct.

For example, the Boolean sub-procedure Valid_PileTally_dec1 and Valid_PileTally_dec2 check if the tally lists appearing in the orange and blue lines allocate only one tally to each competing candidate. As another example, the function list_MEM_dec called on the appended list of the elected and continuing candidates in the pre-state (here the orange line) checks if every element in these two lists is indeed a member of the competing candidates.

Every Boolean sub-procedure of TRANSFER_Auxiliary_dec returns the value true except one of them. The sub-procedure in the definition of TRANSFER_Auxiliary_dec which inspects if the tally list has remained the same detects an anomaly, namely that the candidate A has received $1/2$ votes illegally. As a result, the returned value of this sub-procedure and consequently of TRANSFER_Auxiliary_dec is false. The generic verifier however does not flag invalidity of the transition from the orange line to the blue one yet. It first tries to see if some other transition can acceptably apply. Only when there is no transition left to examine, the generic verifier sends an error message back rejecting the certificate as invalid.

The second example is also about the same concrete certificate in Figure 5.19. There is another problem with this certificate that the generic verifier does not and can not recognise. In the pile of the candidate B in the orange line, there is vote the $(b_7, 2/3)$ that is maliciously not included in the list of uncounted ballots appearing in the blue line. But the instructions of the ACT STV dictate that this vote should also be in the list of uncounted ballots in the blue line. There is no decision sub-procedure in any of the generic transitions' semantics, including transfer-elected, that performs a check on whether or not all or some of the surplus votes of an elected candidate is correctly staged for being counted next. Therefore, as far as the generic verifier is concerned, this mistake does not exist.

One should not be surprised that the generic verifier fails in finding the error discussed in the preceding paragraph. The generic verifier only encapsulates the commonalities of the STV family and checks if the commonalities are preserved in a certificate. If one desires to examine the correctness of a certificate based on the particular STV algorithm used for producing it, then they have to add more Boolean sub-procedures for performing more detailed checks as per instructions of the algorithm.

Nonetheless, one can already proceed to synthesise from the machine a verifier that is executable in an operating system's environment. Such a generic verifier can correctly decide if given evidence ω claimed to have been produced by an algorithm whose counting scheme is STV, instead of e.g. First-Past-The-Post [53] where voters choose whom they prefer most and the candidate with the most amount of votes is elected. However, we wish to generate verifiers that can recognise and validate according to which specific STV algorithm the evidence ω has been output. Hence we need to enrich the computational content of the machine semantics with more pre- and postconditions which are particular to individual STV schemes. We refer to this enrichment process as an instantiation of the machine and discuss it further in the subsequent chapter.

Chapter 6

Instantiating the Generic Machine for Automated Synthesis of Specific Verifiers

We divide this chapter into two parts. The second part demonstrates how we automate the synthesis of an executable certificate verifier for a successful instantiation of the generic machine with an STV algorithm. It discusses the content of the modules **Verifier**, **Translation**, **Parser**, **DeepSpec**, and **Compilation** modules which we briefly explained in the Section 5.2.

In the first part of the chapter, we discuss how to instantiate the generic STV machine presented in the previous chapter in order to obtain verifiers of specific STV algorithms. We shall progress through the same cycle comprised of a specification of the algorithm, implementation of Boolean decision procedures for the specifications, and then verifying the correctness of the implementations against their specification.

We shall only focus on the instantiations of the transfer-elected transition. There are two reasons for our choice. First of all, the process is modular and uniform for every transition. The same development cycle comes into practice to finalise the formalisation and verification of a transition label. Therefore, explaining more than one transition introduces unnecessary duplication. Second, it completes our discussion on the generic transfer-elected in the preceding chapter. As a result, the reader comes to see a finished picture of how the development cycle advances for a single transition. In light of the first point, it also lays down a view of how the procedure happens for every other transition.

To demonstrate the flexibility of our system in producing verifiers of various STV algorithms, we provide three STV instances. The first one elaborates on the instantiation of the machine with the ACT STV. The second example describes how the instantiation process proceeds with the Victoria STV. The last instantiation example details how the same process happens for the CADE STV.

The purpose of including the ACT and Victoria STV schemes is to illustrate the extent to which our framework addresses the current needs of verifying computation with STV algorithms. On the other hand, the case of CADE STV exemplifies the extent to which our system can be extended and easily adapted to accommodate

generating verifiers for radically different STV schemes. The ease of extensibility and adaptability of our framework with different STV algorithms witnesses usability of the system for addressing any possibly emerging future needs for proving the correctness of computation with an STV algorithm.

6.1 Instantiation with the ACT STV

The ACT STV's protocol explains under the section 'Step 3' and 'transfer surplus from elected candidates' under what conditions and how to transfer surplus votes of an elected candidate. We have already identified and formalised some of the clauses appearing in these sections that happen to be common among all STV algorithms. There nonetheless exists some other clauses that are specific to transferring surplus votes under the ACT STV's terms. We summarise and rephrase the clauses appearing in sections as follows.

- $\bullet_1$ the parcel of an elected with surplus is not empty.
- $\bullet_2$ distribution of the surplus of an elected candidate proceeds in one single step.
- $\bullet_3$ surplus of elected candidates is distributed one at a time beginning with those who are elected earlier.
- $\bullet_4$ pile of the candidate whose surplus is transferred is emptied in the post-state.
- $\bullet_5$ only the last parcel of votes received (which resulted in a surplus) is transferred. It may be that the last parcel is the only parcel in a candidate's pile (if only one application of the count action has occurred previously), or more parcels exist (if the count transition has been applied several times earlier)
- $\bullet_6$ pile of any candidate other than the one whose surplus is transferred at this stage remains the same.
- $\bullet_7$ the fractional transfer value is then computed depending on whether or not the last parcel is the only parcel of ballots in the pile of the elected candidate.

We define declarations in HOL4 that formally specify the above clauses. Conjunctions of these formal specifications with the assertion $\mathrm{TRANSFER\_Auxiliary}$ comprise the formal declaration of the ACT STV's transfer-elected semantics as Figure 6.1 illustrate it. We continue by explaining the formal assertions appearing in the figure and pointing at the informal clause(s) with which they match.

- *Lines 54, 55, 65.* the pre-state and post-states on which $\mathrm{TRANSFER}$ operates are both non-final machine states. The components of these states have to satisfy the requirements of the generic transfer-elected formalised in $\mathrm{TRANSFER\_Auxiliary}$.

```
51  val TRANSFER_def = Define `
52    TRANSFER ((qu,st,l):params) j1 j2 =
53      ? ba nba t nt p np bl nbl bl2 nbl2 e ne h nh.
54       (j1 = NonFinal (ba, t, p, bl, bl2, e, h))
55       /\ (TRANSFER_Auxiliary (qu,st,l) ba t nt p np bl bl2 nbl2 e ne h nh)
56       /\ ? l' c.
57                ((bl = c::l')
58              /\ (MEM c l)
59              /\ (!l''. MEM (c,l'') p ==> l'' <> [])
60              /\ (nbl = l')
61              /\ (nba = LAST (get_cand_pile c p))
62              /\ (MEM (c,[]) np)
63              /\ (!d'. ((d' <> c) ==> (!l'. (MEM (d',l') p ==> MEM (d',l') np)
64                                    /\ (MEM (d',l') np ==> MEM (d',l') p)))))
65              /\ (j2 = NonFinal (nba, nt, np, nbl, nbl2, ne, nh))`;
```

Figure 6.1: Specification of the ACT STV's Transfer-elected Semantics

- *Lines 57, 59.* the backlog of the elected candidate is not empty as the candidate c is in the head position of the list. Also, there are surplus votes in the pile of the candidate c to transfer upon this application of TRANSFER (item $\bullet_1$).
- *Line 59, 61.* this application of TRANSFER distributes candidate c's surplus in one step (item $\bullet_2$). Also, once the application of TRASNFER is over, name of the candidate c is removed from the backlog of the elected candidates in the post-state nbl (item $\bullet_3$). Moreover, only the last parcel of votes received by c are staged for distribution among the continuing candidates (item $\bullet_5$).
- *Lines 62, 63, 64.* The pile of the candidate c whose surplus distribution is done is emptied in the post-state (item $\bullet_4$). Also, every candidate's pile in the post-state other than c's remains the same as in the pre-state (item $\bullet_6$).

Note that we take care of formalising item $\bullet_7$ in the semantics of the elect transition. We shall next exemplify how to implement the formal specification given in lines 63 and 64 which correspond to the item $\bullet_6$. We then proceed to formally establish the correspondence between the implementation and the specification of this component.

Figure 6.2 contains the Boolean decision procedures called subpile1 and subpile2 which implement lines 63 and 64.

```
val subpile1_def =
   ⊢ ∀c p1 p2.
        subpile1 c p1 p2 ⟺
        EVERY (λp. MEM (if c = FST p then (c,[]) else p) p2) p1
val subpile2_def =
   ⊢ ∀c ps p1.
        subpile2 c ps p1 ⟺ EVERY (λp. if c = FST p then T else MEM p p1) ps
```

Figure 6.2: Implementation of the Lines 63 and 64 in Figure 6.1

```
⊢ ∀p1 p2 c.
      (∀d'.
           (d' ≠ c ⟹ ∀l. MEM (d',l) p1 ⟹ MEM (d',l) p2) ∧
           (d' = c ⟹ MEM (c,[]) p2)) ⟹
      subpile1 c p1 p2: thm
⊢ ∀p1 p2 c.
      (∀d'.
           (d' ≠ c ⟹ ∀l. MEM (d',l) p2 ⟹ MEM (d',l) p1) ∧
           (d' = c ⟹ ∃l. MEM (c,l) p1)) ⟹
      subpile2 c p2 p1: thm
```

Figure 6.3: Logical Specifications of subpile1 and subpile2 Force their Computational Contents

Figure 6.3 illustrates the proof that subpile1 and subpile2 computationally realise their specification.

Conversely, if the computational content of subpile1 and subpile2 are correct, then their logical declarations are also correct as proofs in Figure 6.4 demonstrate.

```
⊢ ∀p1 p2 c.
      subpile1 c p1 p2 ⟹ ∀d'. d' ≠ c ⟹ ∀l. MEM (d',l) p1 ⟹ MEM (d',l) p2:
thm
⊢ ∀p1 p2 c.
      subpile2 c p2 p1 ⟹ ∀d'. d' ≠ c ⟹ ∀l. MEM (d',l) p2 ⟹ MEM (d',l) p1:
thm
```

Figure 6.4: The Computational Content of subpile1 and subpile2 Forces their Logical Specifications

In a similar manner, we implement other decision procedures each of which cor-

```
26  val TRANSFER_dec_def = Define `
27    (TRANSFER_dec ((qu,st,l):params)
28      (NonFinal (ba, t, p, bl, bl2, e, h))
29      (NonFinal (ba', t', p', bl', bl2', e',h')) ⇔
30        (TRANSFER_Auxiliary_dec (qu,st,l) ba t t' p p' bl bl2 bl2' e e' h h')
31     /\ (case bl of [] => F | hbl::tbl =>
32           let gcp = get_cand_pile hbl p
33             in
34                (~ NULL gcp)
35             /\ (MEM hbl l)
36             /\ (bl' = tbl)
37             /\ (ba' = LAST gcp)
38             /\ (MEM (hbl,[]) p')
39             /\ (subpile1 hbl p p') /\ (subpile2 hbl p' p))) ∧
40    (TRANSFER_dec _ (Final _) _ = F) /\
41    (TRANSFER_dec _ _ (Final _) = F)`;
```

Figure 6.5: Implementation of the Semantics of the ACT STV's Transfer-elected

responds with some specification(s) in the definition of TRASNFER. Conjunctions of these Boolean-valued functions comprise a function (Figure 6.5) that is the implementation of the transfer-elected semantics.

We next demonstrate how the framework modularly extends to instantiations with various STV algorithms. To this end, we present two cases. First, we discuss instantiation of the transfer-elect based on the Victoria STV. Then we explain the transfer-elected of the CADE STV and show how our system realises an instantiation with this algorithm.

6.2 Instantiations with the Victoria and CADE STV

We shall exemplify the modularity in the implementation of the framework component by discussing the case of transfer-elected transition. We shall elaborate on the textual descriptions of the Victoria STV and the variant algorithm by comparing them against the ACT STV's. Then, we present the Boolean procedure which implements the informal semantics clauses of the transfer-elected of the two algorithms and show which informal clause(s) are implemented by which function(s).

We neither illustrate how to obtain a formal specification of the informal conditions above, nor we demonstrate how the implementation of the Boolean functions matches with their formal specifications. We nonetheless encourage the astute reader to construct such formal semantics from the informal clauses that we provide shortly afterwards, or refer to the GitHub repository for details of the formalisation.

The legislature of Victoria's counting scheme does not strictly speak about the notion of the parcel. However, as explained under the subsection 3.2 it transfers votes of an eliminated candidate stepwise which therefore requires separating votes into different chunks (or parcels). Hence, the pile of every candidate has to be divided into parcels so that they can subsequently be distributed one by one. Consequently, the type of a pile object is a list of ballots list, instead of a list of ballots.

Having reminded a necessary remark on the legislature of the Victoria protocol, note that Victoria STV matches with ACT STV on every $\bullet_i$ item under the Section 6.1 except for $i \in \{5, 7\}$.

- $\bullet_{5'}$ transfer all of the surplus votes of an elected candidate at a reduced fraction.
- $\bullet_{7'}$ the fractional transfer value is computed based on all of the surplus (not necessarily depending on the last parcel).

The Clause $\bullet_{7'}$ appears in the semantics of the Victoria's elect transition. But the clause $\bullet_{5'}$ is the main difference between the Victoria's and ACT's informal semantics of the transfer-elected. Figure 6.6 contains the implementation of the Victoria's transfer-elected. As you see in line 33, the transition distributes all of the surplus votes of an already elected candidate (clause $\bullet_{5'}$).

The modularity in implementation of the system clearly shows its strength in the formalisation of the Victoria's transfer-elected. Breaking the formalisation into separate parts each of which encapsulates some part of the protocol, instead of a

```
val TRANSFER_dec_def = Define `
  (TRANSFER_dec ((qu,st,l):params)
    (NonFinal (ba, t, p, bl, bl2, e, h))
    (NonFinal (ba', t', p', bl', bl2', e',h')) ⇔
        (TRANSFER_Auxiliary_dec (qu,st,l) ba t t' p p' bl bl2 bl2' e e' h h')
      /\ (case bl of [] => F | hbl::tbl =>
            (bl' = tbl)
          /\ (ba' = FLAT (get_cand_pile hbl p))
          /\ (MEM (hbl,[]) p')
          /\ (subpile1 hbl p p') /\ (subpile2 hbl p' p))) ∧
  (TRANSFER_dec _ (Final _) _ = F) /\
  (TRANSFER_dec _ _ (Final _) = F)`;
```

Figure 6.6: Implementation of the Victoria's Transfer-elected

gigantic formalisation, has the advantage of using the already existing infrastructure to accommodate STV cases which are close to the already formalised ones. Here, we do not need to re-do everything in order to realise Victoria's transfer-elected. We only need to add the formal clauses for which the Victoria varies from the ACT.

The next STV example, makes the flexibility of our system stand out. The transfer-elected of this STV algorithm is considerably different from Victoria's and ACT's. Nonetheless, we shall see how easy it is to formalise the variations on top of the existing implementations.

6.2.1 Instantiation with the CADE STV

Despite the fact that STV algorithms have had stable characteristics, there have historically been changes to them as well [54]. Therefore, a framework such as ours for dealing with STV algorithms has to facilitate enough structure for being easily adaptable to possibly emerging STV algorithms. Our framework accommodates formalisation and verification of the CADE STV which significantly varies from standard STV algorithms. This case exemplifies the degree to which our system easily extends to a considerably varying vote counting application with STV algorithms.

We discussed in Subsection 4.4 that there is more than one way of formalising the CADE STV in the first component of the framework. We argued how one can merge the semantics of the transfer-elected with that of the elect transition to obtain a more efficient implementation for the CADE STV. The second component also allows specifying and verifying CADE STV in different ways. However, we illustrate how to formalise the transfer-elected semantics in HOL4 for constructing a verifier of the CADE STV without merging the semantics of the transfer-elected with the semantics of the elect transition. From this demonstration one simply understands how to proceed with the integration of the two semantics, if they choose this alternative. The following comprise the informal clauses of the CADE STV's tranfer-elected.

- $\star_1$ the backlog of elected candidates contains one element.
- $\star_2$ backlog of the elected candidates is emptied in the post-state of transfer-elect.

$\star_3$ the election restarts after each round of transfer-elect.

The clause $\star_3$ itself consists of the following sub-clauses.

$\star_{3a}$ pile of all of the candidates is emptied in the post-state.

$\star_{3b}$ all ballots in the piles of all candidates are placed back into the list of uncounted ballots with the name of already elected candidate removed from those ballots.

$\star_{3c}$ the eliminated candidates are "resurrected" meaning they start to be continuing candidates in the post-state.

```
23  val TRANSFER_dec_def = Define `
24    (TRANSFER_dec ((qu,st,l):params)
25      (NonFinal (ba, t, p, bl, bl2, e, h))
26      (NonFinal (ba', t', p', bl', bl2', e',h')) ⇔
27           (TRANSFER_Auxiliary_dec (qu,st,l) ba t t' p p' bl bl2 bl2' e e' h h')
28        /\ (case bl of [] => F | hbl::tbl =>
29                (NULL bl')
30             /\ (NULL tbl)
31             /\ (ba' = APPEND_ALL p)
32             /\ (~ NULL (get_cand_pile hbl p)))
33             /\ (ALL_EMPTY p')
34             /\ (List_Diff e' h' l)) ∧
35    (TRANSFER_dec _ (Final _) _ = F) /\
36    (TRANSFER_dec _ _ (Final _) = F)`;
```

Figure 6.7: Implementation of the CADE's Transfer-elected

We explain in short how some of the clauses correspond with which lines in the Figure 6.7.

- Line 31 is a decision procedure stating that the list of uncounted ballots in the post-state of the transition consists of all of the votes recorded in the piles of candidates (clause $\star_{3b}$).
- Line 33 is another Boolean-valued procedure deciding if the pile every candidate in the post-state is empty (clause $\star_{3a}$).
- Line 34 comprises a Boolean-valued function which decides if the list of continuing candidates in the post-state consists of every candidate competing in the election except those who are already elected (clause $\star_{3c}$).

Once an instantiation of the machine completes we next proceed to verify logical equivalences of the specification of each transition as a unity with its implementation. Based on our experience, proving such equivalences roughly requires 200 lines of HOL4 encoding. We desire to automate them in a way that they practically work for different instances of the machine instantiations with various STV schemes. We remark on two points which are important to notice in order to understand how we progress towards automating these equivalences.

Remark 6.2.1 *In this section, when we instantiated transitions of the machine, particularly transfer-elect, we named instantiations uniformly. We represented an instantiation of transfer-elect with the ACT, Victoria, and CADE STV algorithms by* TRANSFER_dec. *Recall from Subsection 5.2 that every machine instantiation happens in a separate module (see Figure 5.3) two of which we schematically show in the figure by calling them* $\boldsymbol{STV}_1$ *and* $\boldsymbol{STV}_2$. *Therefore, naming the implementation of the transitions of different STV algorithms similarly does not cause any clash when calling them in dependent modules. For example, the* ***verifier*** *module in Figure 5.3 which comprises the formalisation and verification of the verifier notion given in Definition 5.7 calls each instantiation module separately one at a time. Hence, the theorem prover HOL4 identifies the right reference to* TRASNFER_dec *without any confusion.*

Remark 6.2.2 *The Core calculus of HOL4 uses* term rewriting *to manipulate assertions expressed in higher-order logic and ML programming style. Since HOL4 is a rewriting system, what appears as the name of an assertion on the left of* $\Leftrightarrow$ *is therefore secondary to its definitional content on the right side. In light of the previous remark, we can uniformly refer to* names *of instantiated transitions regardless of the algorithm used. Therefore we can formulate evidence verifier based on the names of the transitions but call in the desired instantiation module to embody the names with the semantics of the algorithm intended to obtain a verifier for. Consequently the* ***Verifier***, ***Translation***, ***DeepSpec*** *and* ***Compilation*** *modules in Figure 5.3 which all depend on instantiation modules as their parents are developed once and for all.*

6.3 Automating Verification of the Semantics Implementations

We desire to develop the **Proofs** module in Figure 5.3 in such a way that the statement of the Desideratum 6.1 holds for a large class of STV algorithms. Also, we wish to automate the proofs so that they practically approximate the following desideratum.

Desideratum 6.1 *Assume* $\mathcal{A}$ *is an arbitrary STV algorithm,* $\hat{\mathcal{A}}_{spec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{spec})_{t\in\mathcal{T}} \rangle$ *is a specification of* $\mathcal{A}$*'s instantiation into the machine and* $\hat{\mathcal{A}}_{dec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{dec})_{t\in\mathcal{T}} \rangle$ *is its implementation. Then for any* $t \in \mathcal{T}$ *the framework automatically proves* $\mathcal{S}_t^{dec} \Leftrightarrow \mathcal{S}_t^{spec}$.

We engineer the framework in a way that the **Proofs** module calls instantiation modules as its parents one by one. Considering Remark 6.2.1, we uniformly declare the desired equivalence between the specification and implementation of an instantiated transition and discharge the correctness obligation in the **Proofs** module.

Engineered this way, the desideratum is met under the following two scenarios for the proofs module. However, for a third scenario, proofs in the module may break so that proving the equivalences becomes a semi-automatic interactive procedure. The proofs nonetheless remain reusable after proper refinements so that one needs not re-encoding 200 lines.

Scenario one We have already formalised and verified instantiation of the machine with five different STV algorithms[1]. If the algorithm $\mathcal{A}$ already exists in our framework then the desideratum is satisfied. No effort beyond following simple instructions to execute on the command line is required for synthesising an executable verifier for $\mathcal{A}$.

Scenario two Another possibility is that $\mathcal{A}$ as a whole does not literally match with any of the already existing instantiations of the machine in the framework. However, clauses describing pre- and postconditions of transitions which are components of the semantics of $\mathcal{A}$ do exist in the **Auxiliary** module[2]. Then all one needs doing is to call the formal specifications of the clauses and their respective implementations into the specification and implementation of transitions, respectively, to formally obtain an instantiation of the algorithm. In this case, the proofs also succeed in satisfying the desideratum.

Scenario three Suppose there is a clause in the description of $\mathcal{A}$ whose specification, and therefore implementation, does not exist in the **Auxiliary** module. Then one trivially has to extend the **Auxiliary** module by specifying and implementing that clause and then verifying the implementation correct against the specification. If one was overconfident in their implementation, then they can take the implementation as its specification in which case no verification is required. Note that as including the generic formal machine transitions in the semantics of each transition instantiation is mandatory there are few numbers of such clauses and consequently few lines of encoding needed to formalise them.

Once the proofs of equivalence between the specification and implementation of a machine instantiation either automatically or interactively succeed, the rest of verifier synthesis process for the instantiated machine completely automatically follows. We have fully automated the **Verifier**, **Parser**, **DeepSpec**, and **Compilation** modules. Also translation of assertions in the **Translation** modules happens automatically by calling the name of the translator tool of CakeML. Therefore, from this point onward, no user has to carry out any more task to obtain an executable version of a verifier for their desired STV algorithm. We shall discuss the details of how this automated process advances in the subsequent part of this sequel.

6.4 Verifier in HOL4: Specification, Implementation and Verification

In this section, we discuss the formalisation of the content of the Definition 5.7 which mathematically describes what a verifier is. As you see in the definition, the specification of a verifier of computation with some STV algorithm $\mathcal{A}$, and consequently its

[1] See the Github repository at https://github.com/MiladKetabGhale

[2] See Figure 5.3 in Section 5.2 for more information on this module.

computational content as well, depends on an instantiation of the machine with $\mathcal{A}$. Despite this dependence, the *way* in which the verifier inspects evidence produced upon executions of $\mathcal{A}$ is universal. A verifier, regardless of which algorithm is instantiated into the machine, is a *Boolean procedure* which *checks* the correctness of input evidence by *recursively validating two consecutive steps at a time*. Also, there is more universality to the mechanism of verifying evidence, namely that the inspection of consecutive steps for *validation happens by examining* the correctness of the *disjunctions* of *the semantics of instantiated transitions*.

The above understanding of the behaviour of a verifier for processing information in evidence of instances of computation opens the possibility of automating several modules in the framework's component. In particular, we formalise and implement the Definition 5.7, and then verify the correctness of the implementation once and for all in such a way that it automatically operates for any successful instantiation of the machine[3].

To make the above happen, in light of the above discussion and the Remarks 6.2.1 and 6.2.2, we engineer the **Verifier** module in such a way that it calls in one instantiation module at a time. As a result, the specifications, implementations and proofs developed in the **Verifier** module are adapted to work for creating a verifier for the called instantiation module. This latter means that the verifier which we consequently obtain is a verifier for computation with the algorithm that has been successfully instantiated into the machine and called in the **Verifier** module. We obtain a verifier for various STV algorithms by varying the instantiation module which is called in the **Verifier** module.

So suppose $\hat{\mathcal{A}}^{spec} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t\in\mathcal{T}} \rangle$ is the specification of a machine instantiation with the formalised algorithm $\mathcal{A}$ as defined in Definition5.4 and discussed in Section 6.1. Also assume $\hat{\mathcal{A}}^{dec} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t\in\mathcal{T}} \rangle$ is the implementation of the algorithm $\mathcal{A}$ for which the Desideratum 6.1 either automatically or interactively have been proven correct.

Then the technical specification of a verifier consists of two steps. First, we need to validate whether the first machine state in the certificate is a valid initial state of the count. A valid initial judgement is one where

- every candidate's tally is zero as no votes have been counted yet,
- everyone's piles are empty because no votes have yet been allocated to anyone,
- the backlog of the elected and eliminated candidates are empty,
- the list of elected candidates is empty because no one has yet been elected, and
- the list of continuing candidates consists of every competing candidate in the election as it is the starting point of the competition.

Figure 6.8 shows the formal specification of a valid initial machine state. At this point, we can explain about one other difference between the formalisation of the

[3]By a "successful instantiation of the machine" we mean an instantiation of the machine for which the Desideratum 6.1 either automatically or interactively has been proven correct

generic STV as a machine which we did in Coq with the one carried out in HOL4. In the framework component developed in Coq, the data structure of the machine has a constructor for specifying the initial states of the machine. As a result, the set of machine transitions in Coq have a transition called **Start** for moving from an initial state to an intermediate machine state. One notices that all the **Start** transition does is to assure that the tallying happens only based on the formal (valid) votes of the election and that the initial values of the tally, pile, lists of elected and eliminated candidates and the backlogs are as we have specified them above.

```
val initial_judgement_def =
   ⊢ ∀l j.
         initial_judgement l j ⇔
         ∃ba t p bl bl2 e h.
             j = NonFinal (ba,t,p,bl,bl2,e,h) ∧
             (∀c. MEM c (MAP SND t) ⇒ c = 0) ∧
             (∀c. MEM c (MAP SND p) ⇒ c = []) ∧ bl = [] ∧ bl2 = [] ∧ e = [] ∧
             h = l: thm
```

Figure 6.8: Specification of a Valid Initial Machine State

We choose a different way of accommodating the notion of a valid initial state of computation in our HOL4 formalisation. For checking that the first line of a certificate of computation matches with a valid initial state, we simply perform the checks that Figure 6.8 illustrates them in the definition of initial_judgement. This choice of formalisation has two man advantages;

1. the data structure for constructing machine states becomes simpler than the one given for the Coq component of the framework. Also, the mechanism of the vote-counting used for checking the correctness of certificates produced becomes simpler as we have one less transition to deal with.

2. it is more cost-efficient in terms of time and memory. In the Coq component, the initial state holds information of the votes recorded in the system in order to filter the valid ones. This costs memory for keeping the data and also for during the parsing phase. Also, since we do have a **Start** transition any more, it saves time in processing the information in a certificate for validation or rejection.

The second step for specifying the certificate verifier is to define formally define what a valid transition from one machine state to another is. The following assertions declare that in order to legitimately move from one state to another, at least one transition should be legally applicable.

Using Valid_Step_Spec, we define a recursive procedure for validating a list of intermediate machine states given in a certificate.

Putting the specification of a valid initial machine state and a list of non-final machine states together, we obtain a formal declaration for a valid certificate.

For checking a formal certificate we therefore first verify that certificate starts at a permissible initial stage. We then iteratively check that transitions have happened correctly and that the terminating state is a final one where winners are declared.

```
val Valid_Step_Spec_def =
   ⊢ ∀params j0 j1.
         Valid_Step_Spec params j0 j1 ⟺
         HWIN params j0 j1 ∨ EWIN params j0 j1 ∨ COUNT params j0 j1 ∨
         TRANSFER params j0 j1 ∨ ELECT params j0 j1 ∨
         (∃c. MEM c (SND (SND params)) ∧ ELIM_CAND c params j0 j1) ∨
         TRANSFER_EXCLUDED params j0 j1: thm
```

Figure 6.9: Specification of a Valid Step From one State to Another

```
val Valid_intermediate_judgements_def =
   ⊢ ∀params J.
         Valid_intermediate_judgements params J ⟺
         J ≠ [] ∧ (∃w. LAST J = Final w) ∧
         ∀J0 J1 j0 j1. J = J0 ⧺ [j0; j1] ⧺ J1 ⟹ Valid_Step_Spec params j0 j1
```

Figure 6.10: Specification of a Valid List of Non-final Machine States

```
val Valid_Certificate_def =
   ⊢ (∀params. Valid_Certificate params [] ⟺ F) ∧
     ∀params first_judgement rest_judgements.
         Valid_Certificate params (first_judgement::rest_judgements) ⟺
         initial_judgement (SND (SND params)) first_judgement ∧
         Valid_intermediate_judgements params
           (first_judgement::rest_judgements): thm
```

Figure 6.11: Specification of a Valid Formal Certificate

We implement Boolean-valued decision procedures for each of the above specifications. In particular, the specification of a valid certificate corresponds to the following computational formal certificate checker.

```
val Check_Parsed_Certificate_def =
   ⊢ (∀params. Check_Parsed_Certificate params [] ⟺ F) ∧
     ∀params first_judgement rest_judgements.
         Check_Parsed_Certificate params (first_judgement::rest_judgements) ⟺
         Initial_Judgement_dec (SND (SND params)) first_judgement ∧
         valid_judgements_dec params (first_judgement::rest_judgements): thm
```

Figure 6.12: Implementation of the Formal Certificate Verifier

The correctness of the implementation of the formal certificate verifier above rests on the equivalences we have already established (Figure 6.13) between the specifications and the computational counterparts of the Boolean sub-procedures, namely Initial_Judgement_dec and valid_judgements_dec which the Check_Parsed_Certificate calls to validate a formal certificate. Drawing on the correctness proofs of the sub-procedures on which Check_Parsed_Certificate relies, we establish (Figure 6.15) the fact that it process information in a formal certificate according to its specification in HOL4.

We are ready to proceed to synthesise executable verifiers for performing actual

```
⊢ valid_judgements_dec = valid_judgements: thm
⊢ Initial_Judgement_dec = initial_judgement: thm
```

Figure 6.13: Proofs of the Implementations Correctness for Sub-procedures of Check_Parsed_Certificate

```
val Check_Parsed_Certificate_iff_Valid_Certificate =
   ⊢ ∀params J.
        Check_Parsed_Certificate params J ⟺ Valid_Certificate params J: thm
```

Figure 6.14: Correctness of the Formal Certificate Verifier

computation on a concrete certificate. This process progresses uniformly and modularly for every successful machine instantiation. As mentioned, this part of our framework is completely automated. Therefore, no user has to experience any of the challenges that we as developers faced and overcome during this phase.

6.5 Modular Automated Synthesis of Executable Verifiers Using CakeML

The verified certificate-checking function, Check_Parsed_Certificate, described above, is a good starting point for a verifier, but still has two shortcomings: it is a function in logic rather than an executable program, and as a consequence, its inputs must be provided as elements of the respective data types, whereas certificates are purely textual. We now demonstrate how to address these shortcomings and obtain a verified executable for checking certificates. Our final theorem about the verifier executable is presented at the end of this section.

Parsing. The input to the verifier is a textual concrete certificate file, in a format similar to Figure 5.19 whose grammar follows the rules discussed in Subsection 4.5.3. We also specify this file format indirectly in the ecosystem of CakeML, by defining an executable specification of a concrete certificate parser.
Specifically, we define functions that take a string representing a line in the file and return either NONE or SOME (x:unit), where x is the parsed information from the line. Given these parsing functions — parse_quota, parse_seats, etc. — we write the verifier as a function, above, that parses lines from the file then calls Check_Parsed_Certificate to do the verification.

Translation into CakeML and I/O Wrapper. Using prior work on proof-producing synthesis [105] we can automatically synthesise an implementation of the function Check_Certificate in the programming language CakeML. To this end, we have to translate every Boolean function on which Check_Certificate depends. As a result, we need to translate every Boolean sub-procedure of a successful machine instantiation, and components of the certificate parser as well.

```
val Check_Certificate_def = Define`
  Check_Certificate lines =
    case lines of
    | (quota_line::seats_line::candidates_line::winners_line::jlines) =>
      (case (parse_quota quota_line,
             parse_seats seats_line,
             parse_candidates candidates_line,
             parse_candidates winners_line,
             OPT_MMAP parse_judgement jlines) of
       (SOME quota, SOME seats, SOME candidates, SOME winners, SOME judgements) =>
         Check_Parsed_Certificate (quota,seats,candidates)
           (REVERSE (Final winners::judgements))
       | _ => F)
    | _ => F`;
```

Figure 6.15: Shallow Embedding of Certificate Verifier in CakeML's Ecosystem

Translating the implementations from the HOL4 to CakeML simply amounts to calling the verified proof producing tool of CakeML which accepts as input name of a function and creates a translated version of that function in CakeML. The synthesis tool for CakeML produces a theorem relating the semantics of the synthesised program back to the logical function. However, the result of translating Check_Certificate is a *pure* function that expects the lines of a file as input. The function is useful only for execution in the environment of CakeML, but we need an executable version of Check_Certificate that operates in the environment of an operating system. Such an executable allows validating a large-size certificate of a real election, whereas the Check_Certificate is not suitable for execution on big inputs in the CakeML.

To obtain an executable concrete certificate verifier which actually opens the file in an operating system and reads lines from it, we write the impure wrapper check_count (making use of the CakeML Basis Library) around the pure function, and verify the wrapper using Characteristic Formulae for CakeML, as described by Guéneau et al. [67]. The result is a complete CakeML program (called check_count) whose I/O semantics is verified, witnessed by the theorem check_count_compiled below, to implement Check_Certificate on lines from standard input.

To elaborate further on the above step, the impure wrapper check_count calls two impure functions parse_line and loop. The former, calls I/O functions to read one line at a time from the concrete certificate given as lines on the standard input and parse it. It comprises two phases; one for the header of the certificate file consisting of the quota, seat number, and the initial list of candidates, and the other is for parsing judgement lines. If the parsing fails due to malformedness of a line, the parser messages the appropriate error on the standard output with the line number included.

Nonetheless, if parsing succeeds, the parsed line is fed to the loop function (Figure 6.16) to check if the transition from two consecutive parsed judgement lines is a valid step. The parsing and checking of judgement lines continue until either all

```
val loop = process_topdecs`
  fun loop params i j1 j0 =
    if valid_step params j0 j1 then
      case TextIO.inputLine TextIO.stdIn of
        None =>
          if initial_judgement_dec (snd (snd params)) j0 then
            TextIO.print "Certificate OK\n"
          else
            TextIO.output TextIO.stdErr "Malformed initial judgement\n"
      | Some line =>
      case parse_judgement line of
        None => TextIO.output TextIO.stdErr (malformed_line_msg i)
      | Some j => loop params (i+1) j0 j
    else TextIO.output TextIO.stdErr (invalid_step_msg i)`;
```

Figure 6.16: The Deeply Embedded loop Function in CakeML

steps are verified as correct, or an incorrect step is encountered.

The theorem loop_thm in Figure 6.17 asserts that the loop function returns the correct output NONE if and only if the initial line of judgements in the certificate file is indeed valid and all steps taken to move from one judgement line to its successor are correct.

```
val loop_thm = Q.store_thm("loop_thm",
  `loop params i (Final w) j0 js = NONE ⇔
   EVERY (IS_SOME o parse_judgement) js ∧
   Check_Parsed_Certificate params (REVERSE ((Final w)::j0::(MAP (THE o parse_judgement) js)))`
```

Figure 6.17: The Deeply Embedded Certificate loop Function Returned Value Is Correct with Respect to the HOL4 Definition of Certificate Verifier

We also prove end-to-end properties about the loop function (Figure 6.18). The following theorem states how the loop function behaves once executed (as a subprocedure of the check_count) in an operating system.

Informally speaking, the theorem loop_spec asserts the following. Assume the inputs param the parsed election parameters, the parsed initial machine state j0, the parsed winners lines j1 respectively have the correct types pv, j0v, and j1v and they are read from a system file fs by using the foreign function interface ffi mechanism of the operating system. Then the returned value of an application of the loop function on these inputs and the machine states recorded in the file fs is either the message "Certificate OK" written to the file fs, or it is the error message consisting of the number of line in which it happens.

We also specify and prove similar end-to-end properties for the parse_line as well. However, we simply present the final theorem that proves the correct behaviour of the check_count function upon compilation through the instantiation of the CakeML compiler, and also upon every execution in an operating system.

```
val loop_spec = Q.store_thm("loop_spec",
  `∀i iv j1 j1v j0 j0v fs.
   PARAMS_TYPE params pv ∧
   INT i iv ∧
   CHECKERSPEC_JUDGEMENT_TYPE j1 j1v ∧
   CHECKERSPEC_JUDGEMENT_TYPE j0 j0v
   ⇒
   app (p:'ffi ffi_proj) ^(fetch_v"loop"(get_ml_prog_state()))
     [pv;iv;j1v;j0v]
     (STDIO fs)
     (POSTv uv.
       &UNIT_TYPE () uv *
       STDIO (
         dtcase loop params i j1 j0 (MAP implode (linesFD fs 0)) of
         | NONE => add_stdout (fastForwardFD fs 0) (strlit"Certificate OK\n")
         | SOME (err,j) => add_stderr (FUNPOW (combin$C lineForwardFD 0) (Num(j-i)) fs) err))`
```

Figure 6.18: An End-to-End Specification and Proof of Correct Behaviour of the Executable Version of the loop Function in an Operating System

Compilation in Logic Finally, we would like an executable verifier in machine code (rather than CakeML code). To produce this, we use the verified CakeML compiler [136], which can be executed within the theorem prover itself. This is a time-consuming process: compilation within logic can be a thousand times slower (e.g., half an hour) than running the compiler outside the logic (a second or two). But the payoff is a final theorem which only mentions the final generated machine-code implementation: all dependence on the CakeML language and implementation is discharged by proof.

The final theorem, which we explain further below, is about the generated machine code, represented by the constant check_count_compiled.

```
val check_count_compiled_thm = Q.store_thm("check_count_compiled_thm",
  `wfcl cl ∧ check_countProof$wfFS fs ∧
   check_countProof$x64_installed check_countProof$check_count_compiled (basis_ffi cl fs) mc ms
   ⇒
   ∃io_events fs'.
   targetSem$machine_sem mc (basis_ffi cl fs) ms ⊆
   semanticsProps$extend_with_resource_limit {ffi$Terminate ffi$Success io_events} ∧
   extract_fs fs io_events = SOME fs' ∧
   (stdout fs' (strlit "Certificate OK\n") ⇔
    Check_Certificate (lines_of (implode (get_stdin fs))))`,
```

Figure 6.19: Correctness of the Executable Concrete Certificate Verifier with Respect to Its Logical Specification

We assume (x64_installed) that this code is loaded into memory in an x86-64 machine represented by mc and ms, and that the command line (cl) and file system (fs) are well-formed. The conclusion of the theorem concerns the semantics (machine_sem) of executing the machine: it will terminate successfully (or fail if there is not enough memory) with a trace of I/O events (io_events) such that if we replay those events

on the initial file system, we obtain a resulting file system fs' for which the string "Certificate OK" is printed on standard output if and only if Check_Certificate succeeds on the lines of standard input.

6.6 Experimental Results with Certificate Verifiers

We have performed experiments with the certificate verifier synthesised for the ACT STV. The evaluation has happened on the certificates generated for ACT Legislative Assembly elections in years 2008 and 2012. More specifically, Figure 6.20 illustrates the experimental results of certificate validation for the three ACT's electoral districts Brindabella and Ginninderra in 2012, and the Molonglo district in the Legislative Assembly election held in 2008. The latter is the biggest certificate instance that we have for a real election.

The verifier synthesised for validating instances of computation with the ACT STV verifies each evidence as valid[4]. Note that certifying evidence validity is costlier than detecting the invalidity of evidence of the same size because the former takes parsing *all* lines and verifying *every* transition while the latter does not.

Electorate	Certificate size (mb)	Validation time (sec)	year
Brindabella	57.5	1627	2012
Ginninderra	72.7	1789	2012
Molonglo	195.6	12063	2008

Figure 6.20: Certificate Validation; ACT Legislative Assembly Elections in 2008/2012

It may come as a surprise that the Molonglo certificate is computed by the extracted Haskell program in almost 22 minutes. However validation of the certificate takes about 200 minutes. The main reason for the difference is that CakeML compiler produces an executable machine code that only runs on one core of the machine. On the other hand, the Haskell extracted program runs on three cores. Another reason relates to reading large lines of data and parsing them before processing them for validation. In contrast, the structure of the ballots read and parsed by the wrapper file that we use for preparing ballots to be input to the Haskell program is considerably simpler. As a result, the parsing phase for pre-processing ballots to be counted requires calling less number of functions. Also, the original ballots that are basically comma-separated lists of names are significantly smaller in size compared to a certificate line.

Contrary to underestimations that theorem provers are basically meant for verification, rather than efficient computation, the generated verifier for ACT STV shows good performance. For example, Molonglo certificate is the largest evidence among other Legislative Assembly elections as Molonglo district has the biggest number of vacancies and candidates at the same time. Despite costly computation for meticulously examining the correctness of every small detail in the evidence, the verifier

[4]Using one processor of an Intel Core i7-7500U CPU 2.70 GHz×4

validates it in about three hours. Considering the typical time lapse in publicly announcing real election results, this performance is acceptable.

6.7 Trusted Computing Base of Software Synthesised from Our Framework

In this subsection, we first provide a definition of trusting a computing component and then explain what parts of our tools a user has to trust when computing with the synthesised software from our framework.

Our definition of trusted computing base rests on the concept of functional correctness. In the context of formal verification of a formalised system $\mathcal{S}$, functional correctness of $\mathcal{S}$ refers to having formal proofs that $\mathcal{S}$ satisfies a formal specification of its behaviour in every possible scenario. The theorem given in Figure 6.19 is exemplary of a formal functional correctness statement and its proof which we established for certificate verifiers as executable machine code in an operating system on x86_64 architecture.

Note that in order to demonstrate the functional correctness of a system, one must prove that every component of the system does what it is expected to in every possible situation and that components of the system as a whole function as they are required to. All of these demonstration must happen formally using tools that themselves have been proven functional correct. By *trustworthiness* of a system $\mathcal{S}$, we mean existence of a functional correctness proof that does not depend on any unverified assumption.

We find the above conception of trustworthiness idealistic because we have not heard of a system whose correctness of functionality does not depend on a formally invalidated assumption. In reality, there are some *trust* assumptions made either as direct correctness hypothesis when proving the functional correctness of a system $\mathcal{S}$, or they come into the picture because of the tools relied on when establishing the functional correctness of $\mathcal{S}$. By trusted computing base of a system used for computation we mean the collection of all such trust assumptions. For example, formal verification using a theorem prover assumes that the theoretical arithmetic based on which reasoning happens in the theorem prover has logical consistency. Such a hypothesis is part of the *trust assumptions* of the functional correctness of $\mathcal{S}$.

Although trustworthiness in the formal verification sense stipulated above is far from reach, different levels of the trustworthiness are certainly attainable. From an engineering point of view, trustworthiness can be perceived as a quality to build into the system and not simply a property that an engineered product either has or lacks completely. The degree to which a system approximates the ideal trustworthiness depends on the machinery used as tools for developing the system, the proof statements stated for describing the behaviour of the system and its component, and proofs established for these statements. In general, the lesser the number of trust assumptions and the weaker they are the smaller the TCB of the system is and the higher its trustworthiness will be.

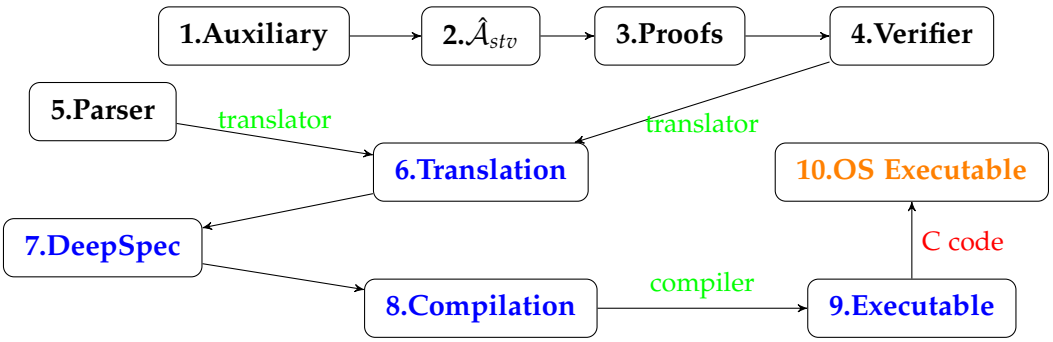

Figure 6.21: The Toolchain of Synthesising an Executable Certificate Verifier

6.7.1 The Toolchain

We next explain about the toolchain of producing machine code for certificate verifiers which are executable in an operating system. This discussion sets the stage for a concrete explanation of the trusted parts of the toolchain which constitute the TCB of software synthesised from our framework.

Figure 6.21 illustrates the toolchain for building a certificate verifier for a formalised STV algorithm $\hat{\mathcal{A}}_{stv}$ as an instance. The direction of the arrows in the figure shows the work flow. As the figure shows synthesising begins from step number (1) that is the **Auxiliary** module and eventually ends at step number (10), namely OS Executable. This latter stage is the final chain of the synthesis process. It is the machine code of the verified certificate checker for the formalised STV algorithm $\hat{\mathcal{A}}_{stv}$ and is executable in an operating system. All the transitions between chains of the toolchain are verified except for moving from item (9) to produce item (10) as we shall explain shortly afterwards.

The toolchain almost entirely develops only inside the theorem prover HOL4 except item (10) in the figure. The modules appearing in black color words, numbered from (1) to (5), are developed in the usual environment of HOL4. Modules numbered (6)-(9) appearing in blue color are implemented in the ecosystem of CakeML which itself is accessed and activated from within HOL4 environment. Therefore, from the beginning of the formalisation and verification and for almost every stage of the synthesis, the development process never leaves the environment of HOL4. This is important because using more than one environment for building software requires aggregation of the TCB of the environments used for creating the software. By staying inside HOL4 for as long as it is possible, we limit the TCB of the technical base of our framework to that of HOL4 only.

Verified Parts of the Toolchain The combination of HOL4 and CakeML together with proofs that we establish reduces the TCB of the toolchain to a significant extent.

In particular the following parts of the toolchain are formally verified to function correctly.

- *verified algorithm implementation.* As we discussed previously, we define computational assertions that are expected to implement parts of an STV algorithm. We verify every such computational function implemented in HOL4 against a logical specification that is meant to describe the behaviour of the function. In other terminology, every computational entity implemented in HOL4 is proven functionally correct. Therefore, plain trust is replaced by formal proof of the correctness eliminating it from our TCB.

- *code translation.* We rely on the verified proof producing tool of CakeML in order to translate the computational assertions proven correct in the usual environment of HOL4 to semantically equivalent assertions in CakeML. This has the benefit of extending the verification performed in the above step for implementations in HOL4 to their equivalent CakeML functions. As a result, we know the translated functions also have functional correctness in the ecosystem of CakeML.

- *end-to-end proofs.* Using the I/O modelling and its associated libraries implemented in CakeML, we establish end-to-end properties that describe and verify the behaviour of all of the toolchain. This guarantees that not only parts of the chain are proven functionally correct but also the integration of the chains functions correctly as well.

- *verified compilation.* CakeML compiler is proven to only produce verified machine code. Using the verified compiler of CakeML, we generate Executable for a certificate verifier of an STV algorithm that is proven correct down to machine code. This achievement together with verification layers obtained in its preceding steps, means that the Executable behaves precisely as its formal specification in HOL4 describes.

Trusted Parts of the Toolchain. We minimise TCB of tools produced, however we cannot eliminate every trusted layer. In the following we list TCB of software produced from our framework and hint on how it can be addressed.

1. First of all, one has to trust that logic on which reasoning in HOL4 relies on is consistent. Second, we need to trust that the HOL4 kernel consists of about 4000 lines of ML code correctly implements the logic. Third, the Poly/ML compiler, operating system and the hardware on which HOL4 runs have to be trusted as well. To mitigate this TCB of using HOL4, it is possible to output a trace of computation and proofs that HOL4 performs for finalising a proof in order to machine check the trace independently. But we must note that using any theorem prover introduces such TCB regardless of the way which we have used it.

2. Also, our certificate parser is not formally verified. It consists of about 150 lines of ML code written inside HOL4 using HOL4's verified primitives. We have however investigated the correctness of the parser by testing. Indeed, the fact that our certificate verifiers validate certificates produced from vote counting tools extracted from the Coq component adds confidence in the correctness of the parser's behaviour.

Kumar et al [88] thoroughly elaborate on the TCB of using CakeML for synthesising machine code. In order to keep this monograph a standalone piece of narrative, we shortly mention the layers that one has to trust when using our framework.

3. The CakeML compiler compiles the program inside the logic, meaning the environment of HOL4, which is proven correct up to this point. However, the compiled machine Executable code has to be written to a file in an operating system for execution, which provides us with the OS Executable. This move from the logic environment to the operating system has to trust a small piece of C code that writes the generated machine code to a file [88] (called OS Executable in the Figure6.21). The TCB involves trusting that the file is not tampered with and that the linker (next paragraph) and OS loader operate correctly [88]. The reader has already noticed that these assumptions are already captured in the final proof of the CakeML compiler instantiation given in Figure 6.19. Therefore, we are not claiming an assertion that is beyond our verification capacity. Moreover, one could validate these assumptions using runtime checks on startups [88].

4. The final theorem in Figure 6.19 is about the execution of a formal machine model. One trusts that the hardware behaves according to this model and that the operating system and other processes do not interfere with the CakeML process [88]. Finally, I/O facilities, including command line, files, and standard streams, in CakeML's basis library are supported by a small C interface to the underlying system calls, for instance for opening a file. One trusts that the implementation of this interface, the C compiler (on the interface code only) and linker [88].

seL4 OS for smaller TCB. As mentioned in the paragraph 3 and 4 above, part of the TCB encountered using CakeML relates to the operating system. For example, it is assumed that other processes do not interfere with CakeML process. Such non-interference can be accomplished if the user runs CakeML on a verified operating system such as the seL4 [85] microkernel that guarantees the non-interference security property [104]. In other words, it is the user who also has the responsibility of enhancing the security of computation by drawing on the right machinery to complement the synthesis process from our framework.

Specification and TCB. One does not need trusting that we have interpreted the textual description of STV algorithms correctly into a formal specification of the algorithms. The reason for excluding the correctness of the specification from the TCB

of our framework is that we have given a thorough explication in Chapter 2 on the textual STV algorithms, and on specific STV instances such as ACT STV in Chapter 3. In the community of humanity research, explication stands as a well known legitimate mode of reasoning on textual documents of various forms including legal documents that can establish new knowledge through interpretative communication with the document [96]. The explication given in previous chapters, therefore (a) provides a justification of our formal specification and (b) provide the reader with a chance to hold us accountable by asking for further explication or possibly even disagreeing with the way that we have understood the STV family. Note that the mathematical formulation presented in Chapter 2 is only an educational step and does not constitute any part of our TCB.

6.8 Adequacy of Evidence Representation

Adequacy of evidence representation in essence questions the relation between a formal trace of computation produced in HOL4 which is reconstructed from a concrete certificate of a computation instance with existence of a corresponding formal certificate in Coq for this same computation instance. We shall subsequently unravel the preceding sentence in more detail and elaborate on an answer for it. To this end, we remind the reader of an earlier discussion in Section 4.5 about the Coq component of the framework. Also, we need to revisit how a verifier operates. The two elaborations together sets the stage for a thorough description of this adequacy notion and the answer followed.

- The reminder on the Coq component. Recall that a Haskell program obtained from the Coq component of the framework, extracts an instantiated version of the generic Termination theorem (see Figure 4.14). Such an instantiation as explained in Subsection 4.5.1 proceeds by instantiating the STV record (see Figure 3.19) with the formally verified STV instance which satisfies the sanity checks of the machine. Also remember that according to the Termination theorem, a formal certificate is produced upon each (symbolic) execution of the machine. This formal certificate is an object of the data type Certificate formalised in Coq as Figure 4.17 illustrates it. Formal objects of the data type Certificate are records of the traces of computation performed by the instantiated machine corresponding to instances of symbolic machine executions. Upon program extraction, a user obtains a Haskell program that has an extracted version of the formal Certificate data type. We refer to this latter extracted data type as *concrete Certificate* data type. Every execution of a Haskell program prints out a concrete object called *concrete certificate* which visualises the trace of machine state visited by the (instantiated version) extracted of the machine to compute the winners based on input data.
- About the Verifier. We explained that every computational assertion which we define in HOL4 is translated using the CakeML translator into a corresponding construct in CakeML. In particular, the data type of machine states defined

in HOL4 (see Figure 5.4) is translated into a CakeML data type whose objects specify machine states. For the sake of the current discussion, we shall call this CakeML data type *CakeML machine states*, and refer to a list of CakeML machine states as a *CakeML Certificate*. Also, the check_count function given in Section 6.5 deeply implements the verifier in CakeML which is then synthesised into an executable verifier upon compilation. The function check_count consists of sub-procedures for parsing the header of certificates and another sub-procedure called loop. The latter function is essentially the corresponding CakeML deep implementation of the HOL4 functions initial_judgement_dec and Valid_Step_dec wrapped with the parser for parsing intermediate lines of the concrete certificate before data validation happens. As the implementation of loop shows, upon synthesis of an executable verifier an input concrete certificate is parsed one line at a time and whenever a parsed line of the concrete certificate successfully matches with a CakeML machine state, the executable verifier parses the consecutive concrete certificate line and if this one also corresponds to a CakeML machine state, the synthesised version of the loop function executes CakeML's equivalent of the Valid_Step_dec to decide if the transition occurred between the two lines has progressed according to a valid machine transition. This process continues until the whole concrete certificate is validated as correct, or at some point an error due to either failure in parsing a line to a valid CakeML machine state happens or an invalid transition is encountered.

To eventually present a tangible formulation of the first adequacy conception, we make three simplifications to the process explained in the previous paragraph. First of all, as the CakeML translator provably translates HOL4 assertions into CakeML constructs in a semantic preserving manner, we identify the objects of the CakeML machine states with objects of the HOL4 machine states. Consequently, CakeML certificates are identified with HOL4 certificates. Second, for the same reason as in the former simplification, we identify a CakeML computational assertion with its HOL4 corresponding function. Third, we shall assume that an executable verifier of an algorithm parses a whole concrete certificate into a whole CakeML certificate first and then validates the whole CakeML certificate for the correctness. The three simplifications considered together mean that one can identify validation of a concrete certificate with an executable verifier with the corresponding symbolic execution of the HOL4 verifier on the corresponding formal certificate.

We note that the formalisation of the machine model in one component is completely independent of the other component. This independence is due to the fact that the HOL4-CakeML part of the framework addresses the verifiability of computation with STV schemes and as a result its development must not rely in any way on the Coq component. However, one may wonder if we could prove that both formalisation in Coq and HOL4-CakeML indeed realise the object of their formalisation (meaning the STV machine) in an equivalent way. By equivalence one means a formal theorem stating that from the HOL4 formal certificate constructed for validating

a given concrete certificate of computation with an algorithm $\mathcal{A}$, one can reconstruct the Coq formal certificate for the same instance of computation. Since both formalisation realise the machine, it is not absurd at all to ask whether one can formally reconstruct a symbolic machine execution trace in Coq from the one created in HOL4 for a concrete trace of computation with an algorithm $\mathcal{A}$.

We are now ready to bring forth an understanding of evidence adequacy. A formal demonstration of the reconstructability of Coq certificate from a HOL4 certificate as in the preceding paragraph is what one can mean by the adequacy of evidence representation. A HOL4 formal certificate only consists of information provided in the corresponding concrete certificate and nothing more. Reconstructability of a corresponding machine execution in Coq therefore rests on the adequacy of the concrete certificate in being enough representative of the computation performed.

The answer to asking about evidence adequacy as reconstructability of Coq proofs from HOL4 formal certificates is positive with the following elaboration. A concrete certificate is merely a printout which follows a required format. There is no way to obtain knowledge from either of the format and data in a concrete certificate about the underlying implementation of the program which created the certificate. However, with access to the source code of the program, one can reconstruct the steps taken by the implementation of the algorithm for generating the concrete certificate. We do have access to the Coq formalisation and therefore we can see what is needed based on definitions of the transitions, the machine states data structure and the type Certificates to reproduce a formal certificate (equivalently a machine execution in Coq) from a given concrete certificate. Although in principle it sounds doable, we have not formally demonstrated such a reconstructability so far. A demonstration as such would certainly be a challenging endeavour and we shall address it in future work.

At this point the astute reader may think of a possible situation akin to a philosophical thought experiment. On the one hand, they may use our Coq component to produce a vote counting implementation of their favorite STV algorithm and execute it on election data. On the other hand, they may use the HOL4-CakeML component to synthesise a certificate verifier for the same algorithm. In absence of a formal demonstration of evidence adequacy, one then may find himself (in a thought experiment) in a situation where the certificate verifier rejects an instance of computation carried out by the extracted Haskell program. In such moments, one would be puzzled as which one to trust! Our experiments, part of which presented in Section 6.6, illustrate that a situation as such does not occur. We advocate relying on formally verified tools for reducing the trust on software used for computation and therefore we understand the word of mouth is no guarantee. Hence the creator (of this framework) has the following advice. If one plays the role of an authority responsible for conducting an election with an STV algorithm, you may rely on the Coq component as it is meant for such users to conduct verified computation. If one is a voter holding the right for the verifiability of computation performed, then you may only trust the decision of the verifier.

Chapter 7

Related Work

To delineate what research relates to us, once again, we recall what the main focus of the thesis is. Our work draws on automata and programming languages theory, the technique of certifying algorithms, and state-of-art verification tools, namely the theorem provers Coq and HOL4 and the CakeML compiler with its ecosystem. Using the above theories, the technique, and the tools, we build the framework that produces tools for verifying or verified computation with STV algorithms. Therefore, we rely on the following criteria for deciding what research is related work to ours.

1. work that relies on the notion of verifier (or checker) either theoretically or practically for purposes of verifying algorithms or particular implementations of an algorithm whether in the context of election schemes or not.

2. work which uses verification tools and/or a relevant methodology for verifying certifying algorithms regardless of whether they are used for verifying voting counting algorithms,

3. any work that focuses on STV from a formal verification perspective.

There is some work on the security aspects of preferential voting systems, STV in particular. We only mention them briefly as the security of elections, in general, is not a topic with which the thesis is concerned. It, however, is future work that we are motivated to address later.

Also, we understand that STV has received attention from the community of social and political sciences as well. We believe the research accomplished in these areas are interesting. However, they are not technically related to this thesis. Nonetheless, we briefly mention those of which we are aware at the end of this chapter so that a reader potentially keen on the literature is guided in the right direction.

7.1 Certifying Algorithms and Checking Computation

The term certifying algorithm refers to a technique where an algorithm is designed to produce an independently checkable certificate for the correctness of computation performed in execution of the algorithm. The term Certifying algorithm is a

terminology related to but distinct from another technical phrase known as proof-carrying code [107]. Before exploring the roots of the term certifying algorithm and how it has entered into the voting literature, we shall remark on the differences of proof-carrying code.

1. Proof-carrying code concerns a priori agreement on a safety protocol for code execution. However, certifying algorithms can be used when the parties involved a priori agree on a soundness notion for a valid protocol execution where proof is expected to be output at run-time witnessing the conformity of the execution with the policies.

2. Proof-carrying code requires verifying compliance with the policy before the execution so that running the untrusted code on the hosting machine is guaranteed to be harmless. But certifying algorithms is used for checking the correctness of computation after the execution has happened.

3. Proof-carrying code describes a static specification of code. In contrast, certifying algorithms concerns the run-time behaviour of an execution.

4. Proof-carrying code thus establishes some level of trust in the static behaviour of the program. However, certifying algorithms can be used for providing confidence in the dynamic behaviour of the program.

Certifying algorithms has a close relationship with the concept of verifier or checker. Indeed, historically the latter has appeared before the former. Blum and Sampath [20] introduced the notion of checker as a probabilistic Turing machine which decides whether or not a program $\mathcal{P}$ correctly computes instances of a given language $\mathcal{L}$. Their work is a mathematical characterisation of what it means to check the correctness of program output. As it is an abstract idea, their paper does not detail the exact mechanisms of checking instances of computation. Consequently, one wonders how their conception of checking handles non-deterministic programs that may produce different results in two different executions. Also, the checker as they describe it is not itself a verified program. The authors demand to improve the reliability of a checker through *testing* instead of formally verifying it. Moreover, related to the last observation, the checker is formulated to identify an incorrect instance of computation with a negligible possibility [63] of failure. Therefore, a program's computation is meant to be checked not with a full (logical) guarantee of the correctness but instead in terms of probabilistic assurances.

The seminal work of Blum and Sampath influenced research in other areas. Only one year after the introduction of the checker concept, Babai et al [11] employed the notion of instance checking to present a theoretical framework for inspecting the correctness of computation carried out by untrusted parties in a game-theoretic setting where a prover communicate pieces of information with a verifier in order to convince the verifier that he has computed the result correctly. Here their definition of checking is also probabilistic. Blum and Sampath's work through Babai et al's has

also influenced other research in the domain of computational complexity to, for example, provide different characterisations of the NP-class [8].

Motivated by the computation checking as set forth by Blum and Sampath, Sullivan and Masson [133] introduce a theoretical framework applicable in the software fault tolerance field. Here the initial conceptions of certificate emerge where the authors speak about "certification trails" as a novel method that combined with N-version programming [9] is used to provide a solution for fault tolerance of software. In short, for a given computational problem, they implement two programs (2-version programming style) $\mathcal{P}_1$ and $\mathcal{P}_2$. The program $\mathcal{P}_1$ returns an output y and a "certification trail" ω for every input value x. Then the second program $\mathcal{P}_2$ accepts x and ω as input and returns a value y' as its output. If both y and y' are the same then they conclude that $\mathcal{P}_1$ has correctly computed y. Otherwise, the mismatch indicates an occurrence of an error (caused by transient fault phenomena). In the case of the mismatch, they require the program $\mathcal{P}_2$ to either signal an error message indicating where the mistake happens in the "certification trail" output by $\mathcal{P}_1$ or itself recompute a correct output y' for the instance of the computational problem being dealt with. Moreover, Sullivant et al [134] in subsequent work extend the same method to support a larger class of data structures and various implementations of these structures. Sullivan et al's work basically fits into the category of recomputation approach discusses in Section 5.1. Also, the program $\mathcal{P}_2$ that they use for recomputing the result is not verified. Consequently, one has to trust that its implementation behaves correctly.

Mehlhorn with his colleagues in a series of papers use the notion of "program checking" or "result checking" (see e.g. [103, 102]) to verify the correctness of computation carried out by programs computing geometric problems [103] and various graph problems (see e.g. [86, 48]) to mention a few. However, neither of these programs outputs a certificate for the instance of computation. Mehlhorn with others [86] eventually in 2003 coin the term "certifying algorithms" as described above. Here for the first time, the concepts of certifying algorithms and checking come together in accordance with how we have used them in the current thesis whereby a checker checks the certificate of the computation for guaranteeing the correctness of an output result instead of recomputing the end result or running a second program similar to Sullivan et al's [134]. In later work, Mehlhorn with his colleagues provides a survey [99] of the certifying algorithm where many algorithms in LEDA [101] have already been implemented as certifying algorithms accompanied by certificate checkers.

The certificate checkers discussed above that Mehlhorn and fellows implemented are not formally verified [3]. To address this shortcoming, Alkassar et al [3] use the combination of light-weight verification tools such as VCC [39] and the theorem prover Isabelle/HOL [108] to statically verify the correctness of checkers. In the initial version of the framework that they created, checkers are developed in the language C and then the automatic code verifier VCC is invoked to discharge loop invariants used to prove the loop statements of the checker program correct with respect to some invariants. Also, the general properties of an algorithm for which

they implement and verify a checker is proved in the theorem prover Isabelle/HOL.

In subsequent work, Notschinski et al [110] replace VCC with Simpl [128] and the verified AutoCorres [66] tool to reduce the trusted computing base required for translating C code into the Isabelle/HOL environment and then prove properties about it. Essentially, they implement their checkers as C programs. To verify the correctness of a checker, they use Norrish's C-to-Isabelle parser [109] to deeply embed C code into a program in the Simpl imperative language. Simpl language has been designed as part of the seL4 project [84]. It allows the parsed C code to be annotated by invariants and then desirable properties to be expressed as Hoare triples about the annotated program. Then one uses the Hoare expressions to reason about the program behaviour. However, to make the reasoning process easier and more human readable, they use the verified AutoCorres tool.

AutoCorres translates in a provably correct way a deeply embedded C code from Simpl into a shallowly embedded monadic representation of the program in the Isabelle/HOL environment. It then takes three other steps to convert the Hoare triples expressed in the environment of Simpl (for the deeply embedded code) into human readable higher-order formulas of the Isabelle/HOL. Notschinski et al then discharge these higher-order proof obligations to establish the correctness of the checker.

Arkoudas and Rinard [7] in 2005 independently of Mehlhorn and others[1] come to invent the term "certifying algorithms". The original ideas that they draw on goes back to 1990 where Rinard introduced the notion of a credible compiler [121]. Accordingly, a credible compiler is (theoretically) designed to comprise components each of which performs some transformation operations on an input program and produces a proof that the transformation preserves some invariants called simulation invariants. In a nutshell, they first assume that there are variables whose external behaviour is observable and whose value remain the same in a transformation. Based on such invariants, they then define correspondence properties about a program and its transformed one where the value of the original program at a particular node is expected to be the same as the equivalent node of the transformed version of the program. They call these properties simulation invariants. They also introduce two logics for reasoning about the correctness of the analysis and the transformation performed by each component of the credible compiler. Here the logics are used for checking the correctness of the computation carried out by each component through inspecting the proof generated by the component upon analysis and transformation.

Arkoudas and Rinard [7] building on the ideas from the credible compilers, introduce a framework for design, implementation and run-time verification of certifying algorithms. They refer to their framework as Denotational Proof Languages (DPLs) and implement the framework in Athena [6]. In essence, by a certifying algorithm, they mean an implementation of a traditional algorithm as a proof-search mecha-

[1] Neither Arkoudas and Rinard nor Mehlhorn and others seem to have known about the work of the other as they do not cite any of the publications of the other party. Only in a Mehlhorn's PhD student thesis, namely C. Rizkallah's, Rinard's work is recognised. However, their publications preceded the submission date of Rizkallah's thesis by a margin of a few years.

nism where every execution of the algorithm on an input x computes a result r and at the run-time also verifies the correctness of r. Therefore, the certificate produced, which represents r, is already a proven correct witness for the computation carried out. In other words, the checking mechanism of certificates produced is built into the algorithm design and implementation whereas our approach is to separate the verification process of the certificate verifier from certifying computation. As a result of their methodology, Arkoudas and Rinard's tools are useful for creating certified algorithms but they not are applicable for verification of certifying vote counting programs that have been implemented in environments different programming than Athena's.

Certifying algorithms were introduced in the context of formal verification of vote counting schemes through the work of Schürmann [130] in 2009. He also independently of Mehlhorn et al and Rinard and Arkoudas describes a technique called "trace emitting computation" whereby individual steps taken by an election voting machine are recorded in order to preserve the voter's intent. The paper mainly focuses on introduction of the "trace emitting computation" mechanism and how the traces are produced and then independently verified for the correctness of the vote casting phase. In subsequent work, DeYound and Schürmann [45] pioneer the use of theorem proving for vote counting purposes. They use a rewriting system for the "vanilla" STV and First-Past-The-Post (FPTP) [53] schemes expressed in Linear Logic [62]. They then implement the systems in Celf [127] which supports linear logic programming. Moreover, they mention producing "traces of [the implemented] rewriting" steps taken to compute winners of an election. This trace is meant to provide independent auditing of the tallying process of counting votes with the software used. However, they neither explicitly give an instance of a trace nor they clearly talk about a trace checker.

Pattinson and Schürmann [114] are the first ones to explicitly present instances of certificates for vote counting schemes. They formalise the "vanilla" STV in the theorem prover Coq as a system of logical rules. An instance of counting votes then is realised as a sequence of logical rules applications. A certificate is simply then a trace of the derivation tree [138] that begins at an initial state and ends at a final state where winners are announced. Verity and Pattinson [139], Ghale et al [58], and Pattinson and Tiwari [115] all follow a similar trend of certification. The work of these researchers is accompanied by certificate checkers as well. However, most of them [114, 139, 115] do not formally verify their checker to substantiate the correctness of the checking phase. Their checkers are written as Haskell programs that themselves have to be trusted for the correctness. To our best knowledge, Ghale et al [60, 61] is the only work that gives a full static verification of certificate checkers for STV algorithms.

There is some other work that uses the notion of certification and checking. De Nivelle and Piskac [42] formally prove the checker for the priority queues implemented in the LEDA library correctly. Bulwahn et al [25] discuss a case study where they

introduce a "SAT checker" for independently checking the unsatisfiability of an assertion claimed by an SAT solver. This way they reduce the trusted computing base of using the SAT solver. They prove a soundness property for the checker that if the checker rejects an imported propositional resolution proof for the SAT solver then indeed the proof witnesses unsatisfiability of the assertion solved in the SAT environment. However, there is no completeness theorem proving the other direction that if an assertion is inconsistent, meaning unsatisfiable, then the checker for sure rejects the checker also supports the inconsistency claim of the SAT solver. Moreover, for importing the proofs from the SAT solver into the checker (implemented in Isabelle/HOL and then extracted as an ML program), they use an inference rule whose soundness relies on an assumption that is not proven correct. It rests on the assumption that their model of monadic program introduced in their paper is compatible with the semantics of ML language. Last, Thiemann and Sternagel [137] implement a checker in Isabelle/HOL for inspecting termination of some rewriting systems. Then they use the code extraction mechanism of Isabelle/HOL to obtain a Haskell checker program.

7.2 Theorem Provers for Vote Counting

Before beginning the discussion on the use of theorem proving tools in the field of electronic voting, we bring an understanding to the attention of the astute reader. We indeed understand that a fair evaluation and review of the creative work conducted in this area, similar to any other field, should consider the state-of-the-art methods and tools available at the time of carrying the research and engineering out. Because of this understanding, we acknowledge their effort in extending the formal method techniques to the field of electronic vote counting, specifically complex voting algorithms such as STV schemes.

Use of theorem proving to verify vote counting is an emerging field. The only work that uses the theorem proving method for verifying certificate checkers of vote counting programs is our own work [60, 61]. Other approaches focus on the formalisation of some election schemes for obtaining provably correct implementations in order to correctly compute election results. Although they all present their implementation as verified, we shall discuss in the current and also the subsequent section that the level of verification established varies from one work to another.

DeYoung and Schürmann [45] as mentioned earlier use Linear Logic syntax as an intermediate step for formalising the FPTP and the "vanilla" STV schemes. They first give a mathematical characterisation expressed in Linear Logic syntax for the schemes. Essentially, they specify the clauses of the vote counting protocol as a set of logical axioms. They come to view the axioms given for each scheme as multiset rewriting rules which are then implemented in the environment of Celf as a linear logic program. They draw two conclusions from the fact that the implemented program is a translation of the mathematically specified counting schemes:

a. the semantics of the specification and implementation match.

b. there is no need to verify the implementation because no gap exists between the specification and the implementation.

We agree that the semantics of the specification language used is the same as the semantics of the language into which they implement their program. Correspondence of the semantics of the source and target environments provides more confidence into the reliability of the specification transliteration into an executable. However, we have reasons to hesitate to accept either of the claims (a) and (b) above. First of all, we know that manual translation is not a (formally) verified process. Also, as they themselves admit in their paper, "interpretation" of axioms take place in order to understand the logical specification as such and such in order to implement it. This "interpretation" is separate from the interpretation required for obtaining the logical specification from the textual description of the counting protocol. As a result, one expects to reason, instead of pure trust, that such an "interpretation" of the specification into the implementation is indeed justifiably trustworthy.

Moreover, assuming both items (a) and (b) are correct, one expects the implementation to be verified against some general properties that instances of computation with an STV scheme must satisfy. For example, one must (formally) demonstrate that the rewriting rules collectively process information as per instructions of the algorithm[2]. However, DeYong and Schürmann do not present any such formal proof. Therefore, we have to trust that the program as a whole behaves correctly.

We should note that the work of DeYoung and Schürmann has been significant to the formal verification community working on the electronic election from (at least) three aspects:

1. they pioneered the use of formal specification for providing more confidence into the correctness of specification of vote counting algorithms.

2. they also started employing theorem proving for verifying vote counting schemes. Their work opened the possibility of considering the use of the theorem provers for further research and engineering endeavours to formally verifying voting schemes.

3. they are the first to mention applicability of producing a trace of computation for *auditing* the tallying process.

Pattinson and Shürmann [114] introduce another perspective into a different perspective of formalising a vote counting scheme. A novelty of their work is discovering that the counting happens through discrete states which are then formally represented as a type. The second novelty is expressing the textual description of the vote counting mechanism as a logical rule-based proof system where clauses of counting protocol are realised as side conditions for each rule. The combination of these two

[2]We have established such a proof in the applicability theorem in Section 3.4.2.

gives rise to a third novelty which is formalising the counting mechanism of an STV algorithm as a gigantic inductive type whose constructors are the formalised rules.

They apply their approach to formally verify the implementation of the "vanilla" STV and FPTP schemes in Coq. They use the extraction mechanism of Coq to generate a Haskell executable program for actual counting of election ballots. The Haskell program produced outputs a certificate for every instance of computation. Here for the first time, we come to transparently see a concrete example of a vote counting certificate. Finally, they are the first ones to use the certification technique as means providing *independent verifiability of the tallying phase*.

Pattinson and Schürmann's modelling of vote counting as logical rule application is elegant. However, the use of a gigantic inductive type for formalising the counting mechanism prevents their work to be uniformly scalable for modular design and implementation of various STV algorithms. As a consequence of their formalisation style, adapting their encoding to a different STV scheme requires changing the formal specification of the type constructors of the gigantic inductive type which in turn forces the user to make ad hoc modification of various part of the encoding to fix the broken proofs[3].

Moreover, the data structure used for encapsulating the states of the computation defines the types of tallies as a function from the type of candidates into the type of natural numbers in Coq. Therefore, their formalisation does not support tie-breaking in a possibly occurring situation where two candidates have an equal amount of votes and are equally weak but one of them must be removed to progress the counting. Also, they transfer the surplus votes of an elected candidate at the full rate of 1 instead of reducing the votes at a fractional transfer value. The main reason behind this choice of formalisation for transferring votes of an elected candidate is the type which they have used for representing tallies, namely natural numbers instead of exact fractional numbers in Coq.

Also, the formal specification of piles does not support a formalisation of STV schemes such as Victoria, Tasmania, and ACT STV without modifying the inductive type which defines the counting states. Consequently, adapting their system to other STV algorithms other than "vanilla" STV requires a significant amount of re-encoding or fixing broken proofs and implementations. In other words, their encoding does not enjoy from modularity in design or implementation.

Pattinson and Tiwari [115] formally verify Schulze voting method [129] using the Coq theorem prover. Then using the extraction mechanism of Coq, they obtain a provably correct implementation of the algorithm which produces a certificate upon every execution of the program on input competing candidates and election ballots.

In order to provide a concrete foothold for describing their work, we first explain in short what the Schulze voting scheme is. Schulze method is a voting scheme used in some elections of open software communities [146] for electing a single winner. The ballot design is similar to STV where voters can rank candidates according to

[3]See the discussion presented at the end of Section 4.4 for a detailed elaboration on other problems with choosing a gigantic type for formalising the counting mechanism of an election scheme.

their preferences. However, in contrast with STV schemes, the Schulze system allows voters ranking candidates equally by assigning them the same number.

The counting mechanism of Schulze relies on the concept of the relative margin of victory of A over B defined as the number of votes which prefer A over B, which we denote by the symbol $\mathcal{R}_m(A, B)$. Then the algorithm introduces the notion of a path p from a candidate A to the candidate B defined as a sequence $p[A, B] = \langle C_1, ..., C_n \rangle$ where $A = C_1$, $B = C_n$, and for every $1 \leq i \leq n$, $\mathcal{R}_m(C_{i+1}, C_i) < \mathcal{R}_m(C_i, C_{i+1})$. In other words, a path from A to B consists of candidates who are successively preferred less along the path. Schulze defines the strength of a path to be equal to the relative margin of victory of the weakest link in the path. In mathematical symbols, the strength of a path p from A to B is $min\{\mathcal{R}_m(C_i, C_{i+1}) \mid \forall C_i, C_{i+1} \in p\}$. Schulze imposes a transitive relation, called generalised margin of victory, on the set of candidates. The generalised margin of victory of a candidate A over B is defined as the maximum of the strength of all paths from A to B. Using the generalised margin of victory, he then illustrate that every instance of counting votes with his algorithm has a winner A where

1. for any candidate B, there exists a path p from A to B.

2. for any candidate B, the generalised margin of victory of B over A is not greater than the generalised margin of victory of A over B. In other words, for any candidate B, there is no path p' from B to A whose strength is stronger than p.

Pattinson and Tiwari take three steps in order to obtain a certifying correct implementation of the Schulze method. They first encode a propositional-level specification of the items (1) and (2) in Coq together with auxiliary assertions needed to formally define the Schulze scheme. Then they implement a type-level computational implementation for the items (1) and (2) as well. They prove that the propositional specification is equivalent to the type-level implementation of the algorithm. As a result, they extract the proven correct type-level implementation in order to (a) compute the winner of an election having the Schulze vote counting algorithm, (b) produce a certificate for each instance of executing the extracted implementation. Every certificate output upon execution of the extracted program provides information for independently checking the correctness of both item (1) and item (2) above.

An interesting aspect of their work relates exactly to the part of a certificate as such that gives information that no path from a defeated candidate B to the winner A is stronger than the strongest path from A to the defeated B. For this, Pattinson and Tiwari draw on their mathematical insight that the inductively defined predicate which encapsulates item (1) above can also be equivalently realised in terms of least fixpoints of appropriate defined monotonic operators on the set of candidates. They then come to see that non-existence of a path from the defeated candidate B to A as in item (2) is equivalent to membership in the greatest fixpoints of operators each of which is dual to a fixpoint operator above. Pattinson and Tiwari show that the greatest fixpoint as such is the supermum of "co-closed" sets. The fruit of this understanding is that they can effectively compute a co-closed set and consequently

(a) decide on the non-existence of a path from B to A, and (b) record the co-closed set computed for visualising them in a certificate.

Dawson et al [41] are the only ones who use the theorem prover HOL4 for formalising a vote counting scheme. They formally specify the Tasmania STV and also provide an implementation of the scheme in HOL4. Then using the implementation in HOL4, they obtain an ML program that is said to correctly compute elections based on the specification of the scheme.

They initially claim (on page 6 of the paper) that they obtain the specification independently of the implementation and directly based on the textual description of the Tasmania STV. However, elsewhere in their paper (on page 7 of the paper) they say that "the [specification] SPECHOL must be build based on assumptions about [the implementation] IMPHOL's structure". The authors also admit that such reliance of the specification on the implementation results in "certain level of coupling between the SPECHOL and IMPHOL". However, we are not sure that they are correct in claiming that such coupling "can not be avoided". The problem with this approach is that once one allows the specification to be affected by the implementation, the integrity of the whole formalisation and the authenticity of the specification correctly representing the actual counting scheme becomes questionable.

Moreover, they admit that their approach is "very labour-intensive" to the point where Dawson who has "at least 20 years of experience in using higher-order theorem provers" has spent "at least six months of full-time work" in order to complete the proof obligations. Given that their encoding is not modular, adapting it to another STV would possibly require (at least) as much work as it had taken an expert theorem prover to establish some proofs. Therefore, the usability of their method and its practicality faces serious questions.

Also, the main proof obligation that they discharge proves that the implementation IMPHOL logically enforces the specification SPECHOL. This provides a soundness proof that IMPHOL is not logically inconsistent. However, the inverse direction of the implication, namely that whenever the specification is correct the implementation also necessarily returns the correct result, meaning IMPHOL implies SPECHOL, is missing.

Additionally and more importantly than any of the above observations about Dawson et al's work, they manually translate the IMPHOL from the environment of HOL4 into an ML-style program executable in an operating system's environment for counting votes. Unfortunately, the semantics of source (meaning HOL4) and the target (ML language) differ. On the other hand, manual translation of code itself is not a verified method for translating an assertion declared in HOL4 to an ML-style declaration in a semantic preserving way. Therefore, the verification carried out for the IMPHOL and any of its components do not legitimately extend to the ML implementation used for vote counting.

7.3 Light-weight Formal Methods for Vote Counting

There is some work which studies voting schemes, in particular, some variants of STV, using the light-weight verification tools. Some of them discuss the design and implementation of a remote electronic voting [81, 82, 83, 33]. Other work focus on introducing techniques for analysing different STV algorithms with respect to social choice theoretic aspects [15, 14], or analysing the properties of the vote casting and ballot counting processes in an election.

Kiniry and his colleagues in a series of papers [81, 82, 83, 33] report on their progress and contributions in creating an open source remote internet voting system called KOA. The system has been used in European Parliamentary election of 2006 [83]. It consists of different components each of which takes care of a different phase in a remote internet election [82]. For example, the module responsible for tallying votes has been implemented in Java [82]. To provide confidence in the correctness of the tallying program, the Security of Systems group at the Radboud University who was responsible for the development and verification of the tallying module, first used the JML [89] to annotated the Java program with specifications. The team then used the ECS/Java2 [55] tool to check the code against the specification. Accordingly, 50 percent of the code has successfully been covered through this method. For the rest of the 50 percent left, they have generated test cases all of which have successfully been passed.

Dennis et al [44] do a case study on the Kiniry et al's work above. They use their own analysis tool called Forge [43] to check the specification obtained above by Kiniry and his colleagues [81, 82, 83, 33] to see if their tool is competitive enough with others. They discover counterexamples to the JML contracts that have not been spotted by either of the ECS/Java2 tool or the unit tests performed by Kiniry and his fellows.

Cochran and Kiniry [33] first analyse the textual description of the Irish Proportional Representation STV (PR-STV) [53, 54] used in national and European elections. Then they specify the algorithm in the syntax of the first-order logic embedded in the Alloy [75] model finder. Using Alloy, they check the soundness of their specifications and formulae implemented. Consequently, they obtain some degree of confidence in the correctness of their specification.

Beckert et al [15] use light-weight formal methods for deciding whether or not a given voting scheme satisfies some choice-theoretic criteria. They demonstrate their approach by discussing the behaviour of the "standard" STV with respect to two "tailor-made" criteria that are meant to be satisfied in order for the scheme to reflect some degree of fairness. They use the first-order logic syntax for specifying the conditions. They use a bounded model checking technique to verify that the scheme satisfies the specified criteria. They perform a case study on CADE STV and discuss the deficiency that this scheme has with respect to their formal criteria for STV algorithms.

In another work, Beckert et al [14] follow a similar approach as above in relying on the first-order logic to represent the semantics of voting schemes, in particular,

STV. However, this time they use different tools for implementing the specification of the "standard STV". They use the logical framework Celf for realising the algorithm and declarative properties, and building model checking tools for constructing counterexamples.

We must note that none of the work carried out using a light-weight method is modular. They all focus on one particular algorithm and adapting their approach to various STV algorithms demands technical effort.

7.4 TCB of the Related Work's Approach Compared with Ours

We have engineered our framework based on four requirements, namely approximating the universal verifiability of tallying through minimisation of the TCB, verifying the tools generated from the framework, transparency of computation with the tools produced, and practicality in deploying them. Our framework is the only existing environment that allows a modular treatment of the STV family. Therefore, it is impossible to compare other related work with ours considering all of the above criteria. We can nonetheless objectively compare the methodology of the related work with ours in terms of the verifiability and verification degree that they establish. In other words, there is an objective ground for comparing the TCB of adapting their approach for carrying the same task that we have performed.

As mentioned, some approach verification of vote counting software, STV algorithms, or properties that an STV scheme is expected to satisfy using light-weight verification tools. The main problem with light-weight verification method is that software, such as Z3, VCC, and ECS/Java2, employed in the light-weight approach are not proven complete. Consequently either full coverage of code is not possible with them [32] or even if it was, the absence of a counterexample does not and cannot imply the (full) correctness of the program or algorithm specified. Moreover, these tools are themselves complex programs which are not verified. Therefore using them requires a huge amount of trust.

Also, we have already elaborated, while explaining Dawson et al [41], on how large the TCB of manual code transliteration is when the semantics of the source and the target differ from each other. A subsequent improvement on code transliteration is DeYoung and Schürmann where the semantic difference between the source and the target is removed. Nonetheless, there is an intrinsic large TCB introduced when relying on a manual code transliteration regardless of other considerations. This happens mainly because the methodology is unverified.

There are verified tools developed for eliminating the trusted layers that are typically introduced when moving from environment to another. The work of Noschinski et al [110] is an instance of a framework that employs such verified tools, namely the AutoCorres. However, Noschinski et al's methodology also has some TCB. Figure 7.1 shows a schematic picture of the overall methodology exercised by Noschinski et al for creating their framework. Two of the trusted bases relate to the I/O calls hap-

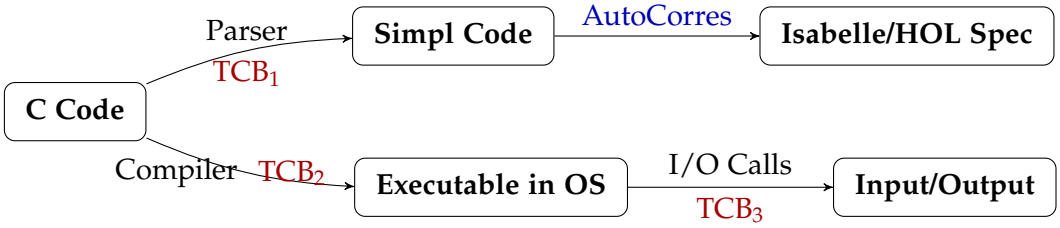

Figure 7.1: TCB of Notschinski et al's Framework

pening as interactions between the executable code and the operating system. The C compiler which compiles the C checker program also has to be trusted. Finally, the C-to-Isabelle parser which the Simpl and AutoCorres rely on consists of about 25000 lines of unverified code which has to be trusted [65].

We also have discussed some work that uses the extraction mechanism of a theorem prover, mainly Coq, to generate executable Haskell programs for computing the winners of an election. The tools that Pattinson and Tiwari's work uses, similar to other work [114, 139, 58, 59] using Coq for generating an executable program, has non-trivial TCB. Figure 7.2 illustrates the layers that are generically trusted when using Coq for extracting a Haskell program for verified vote counting. The amount of trust required in this approach is independent of which language one extracts. The wrapper file in the figure consists of commands for instructing the compiler on how to visualise the data upon execution, and a parser which reads input data either from a file in the operating system or on the command line. Such a wrapper usually comprises few lines of codes which can be inspected (assuming the source code is available for auditing), and it only calls few built-in libraries. Despite that an observer may obtain a level of confidence through mere auditability of the wrapper file, it has to be trusted together with all of its dependencies from the Haskell language.

Generally, advocates of Coq would present two arguments for justifying this TCB. They explain that the theoretical framework behind the extraction mechanism is studied [91, 90] and some pen-and-paper proofs of correctness exist for this tool. Moreover, the tool has been used in several complex projects successfully which itself is

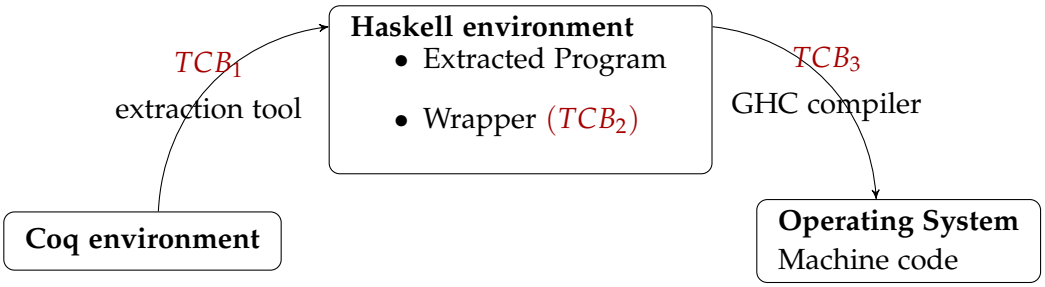

Figure 7.2: Trusted Computing Base of the Coq's Extraction Tool

evidence for the reliability of the Coq's extraction means. We agree that the explanation gives some justification to use of the tool as we have also employed the same mechanism for developing the first framework component. However, we aspire for the ideal and attending to the established standards of engineering trustworthy systems one of which is minimising the TCB as much as it practically is possible.

It is noteworthy that replacing Coq with the theorem prover Isabelle/HOL which also has a code extraction tool [68] suffers from the same large TCB. Similar to Coq, the extraction tool of Isabelle/HOL is not also verified. More generally, using any theorem prover with such an unverified code generation means requires the same amount of trust to lay in the correctness of the process and outcome.

The above TCB consequently casts questions on the verifiability of the tallying and the correctness of the computation carried out. The certifying algorithms technique that some researchers [114, 139, 58, 59, 115] wisely incorporate into their work is indeed a method introduced in the context of vote counting partly for addressing the deficiencies caused by such code generation methods. However, unfortunately, they either do not use a certificate verifier [139] or the implemented verifier itself is not proven correct [114, 115]. Therefore, the unverified verifier itself adds to the TCB of their system instead of reducing it. Consequently, there remains trust to invest in the verifiability of the tallying and correctness of the computation carried out using such systems.

7.4.1 Smaller TCB Using Our Framewrok

When one begins engineering a product for carrying a task out, a natural question is about tools and the methodology which are more suitable for crafting a solution. In this subsection, we discuss why our approach and the tools we employed as the technical basis for building the framework is superior, in terms of the trustworthiness, to the methodology and tools adapted by the related work above.

Verified Code Translation. Unlike some related work (e.g. [41, 45]), we avoid untrusted manual code transliteration whose correctness is left at the mercy of the meticulousness of the programmer. We achieved verification of program translation from HOL4 to the environment of CakeML by using the verified proof-translator tool [105] of CakeML. The tools are guaranteed to translate a computational assertion from HOL4 to CakeML in a semantic preserving manner. As a result, all of the proofs established for our functions verified in HOL4 also extend to the translated counterparts in CakeML environment.

Proving end-to-end Properties. None of the above approaches can even theoretically establish any formal proof about any informative relation between the source of the implementation and the behaviour of the executable program in the operating system's environment. For example, Noschinski et al [110] simply cannot prove any end-to-end property about their C code source and its machine executable version. Although they rely on for verified AutoCorres for a reliable translation of *the parsed*

C code source into the theorem prover's environment, however various stages of their verification process (including code translation) and code compilation proceed in different environments neither of which has direct trustworthy access to the content of the other. Consequently, they cannot demonstrate any end-to-end property about computing with their checkers. In contrast, CakeML's ecosystem of proofs, the translator tool, and the verified compiler are all activated and accessed from within HOL4's environment. Also, the I/O model of CakeML allows expressing and reasoning about the behaviour of the source code if the program as in HOL4, its HOL4 specification, and the machine executable version of the program having been compiled by the verified CakeML compiler. In particular, we establish formal proofs that the machine executable certificate verifiers behave in accordance with their HOL4 specification (see e.g. Figure 6.19).

Verified Code Compilation. We use CakeML compiler which is proven to only synthesis machine executable code that is verifiably correct with respect to the specification of the program compiled and deep end-to-end specification established above. No existing work on vote counting in the context of elections comes even close to this level of verification. All of them rely on unverified compilers for executing their program. Note that one may argue that approaches such as Noschinski et al [110] whose source checker code is written in C language can use the CompCert [21] compiler for executing their checkers on data. We must remark the astute reader that verified compilation alone is not enough; it is trustworthy code compilation *together with end-to-end deep specifications about a program's behaviour* that provide the outstanding degree of minimising the TCB that our work achieves. Use of verified compilation in isolation simply means that no matter what the program does, the compiler does its job right when compiling the program whose behaviour may not even be verified.

7.5 Literature on Security of the Preferential Voting Protocols

There are only a handful of implemented voting systems that support preferential, particularly STV, voting schemes. Prêt à Voter [124, 37, 70, 125], vVote [26, 5], and iVote [69] accommodate voting with an STV algorithm and have been used in real elections. Also, Benaloh et al [16] implement their voting system which supports STV. However, to the best of our knowledge, their system is only an effort to test the feasibility of their approach. In this section, we briefly outline the general aspects of the work carried out for designing and developing these systems.

Ryan [123] introduces Prêt à Voter as a (theoretical) voting system aiming at establishing end-to-end verifiability [17, 126] of election outcome while securing the privacy of the voter and the vote. The system has been extensively studied, refined, extended and improved throughout years. Although there are some technical differences between various versions of this system, they all have commonalities in how they generally try to establish the trustworthiness in using them [124]. Here we only

explain about the general method of how Prêt à Voter handles an election and refer the reader to Ryan et al [124] for a detailed technical comparison among different system versions.

Prêt à Voter [70, 124, 37, 125] uses a two-column ballot design detachable into a left and a right hand side. The left hand side consists of the randomised competing candidates' names. The right-hand side comprises blank spots corresponding to candidates' names one of which is to be marked as the choice of the voter. The right-hand side also encapsulates at its bottom encrypted information on the random permutation of candidates appearing on the left half of the ballot. This encrypted information is referred to as "onion". Every voter goes to a private booth, marks his/her preferred candidate(s) as their choice and then casts the vote into the system. The system creates a digital signature [63] for the vote cast and then issues a receipt to the voter. Once the polling place closes, authorities publish all of the receipts given to voters in a public bulletin board so that a voter can check if their receipt appears among the published ones. Tallying votes proceeds with the decrypted votes. In order to eliminate any linking between the encrypted and the unencrypted votes, to protect the vote integrity and the voters' privacy, Prêt à Voter uses a combination of mixnets [30]. Each mix is run by an authority who performs, using his/her private key, a random secret permutation on the votes received from the preceding mix and outputs the results as an input to the next mix. Moreover, every mix transforms the ballot forms to hide the secret permutation performed. Nonetheless, every mix at the same time removes one encryption layer from the "onion" so that the final resulting votes are all plain texts as cast by the voters. To audit, and thereof establish the verifiability of, the operations of mixnets, Prêt à Voter versions use Randomised Partial Checking [76]. Once the votes are verifiably anonymised and decrypted, the tallying progress towards determining the winner of the election[4].

Prêt à Voter aims at establishing end-to-end verifiability through auditability of data throughout the process, but not formal verification of the machinery and implementations [124]. Accordingly [124], versions of Prêt à Voter satisfy the individual and universal verifiability [5]. The system is versatile for supporting different voting schemes [124]. Also the community developing, maintaining and enhancing the system have the concern of improving the user-friendliness of Prêt à Voter [124]. Moreover, it provides guarantees into the privacy preservation of the vote and the voters' identity and satisfies the receipt-freeness property as well. However, voter coercion is still arguably an issue for Prêt à Voter and every other system which relies on mixnets for handling a preferential voting scheme such as STV [106, 16].

Burton et al [26] build on the Prêt à Voter protocol to design a version suitable for the STV scheme used in the Victoria state for the state elections. They discuss the modifications required to make to the ballot design and the auditing process in order to make the protocol desirable for a preferential voting scheme with over 35 compet-

[4]It is noteworthy that while some variants of Prêt à Voter combine the mixing and decrypting, there are other versions which combine decrypting and tallying [123].

[5]see Cortier et al [36] and Jonker et al [77] for surveys on the verifiability notions.

ing candidates. This work subsequently [5] integrates into a real implementation of a voting system called vVote used for casting and recording about 1200 votes in the Victorian state elections held in November 2014. Culnane et al [27] report on the challenges overcome during the implementation process. They also elaborate on the human processes involved including user experiences, and also provide a security analysis of the system.

As mentioned above, voting protocols for preferential system arguably suffer from a risk of voter coercion[6]. They may fall victim to the "Italian attack" [106] where a coercer forces a voter to cast his/her first preference as the coercer wishes and then rank the rest of the preferences in such an odd way that the coercer after the publication of the anonymised decrypted votes can, with high probability, check whether or not the coerced voter acted as the coercer has forced them to. Benaloh et al [16] use a combination of secret permutation of ballots and homomorphic tallying of votes for an STV election which is similar to the Victoria STV used for the Senate elections in the Victoria state of Australia. Their aim is to introduce a mechanism that can be integrated into a full fledged voting protocol where votes can be anonymised through mixing, however at the final mix instead of decrypting the votes for tallying them, the homomorphic tallying can be adapted. They implement their method for tallying some portion of the votes used in a real state election in the Victoria state which accordingly is feasible for deployment in a real-size election.

All versions of Prêt à Voter including vVote are precinct-based voting system where voters have to attend personally, as opposed to remotely, to the polling place for casting their vote. We know of a voting system, namely iVote, which allows remote voting for the STV scheme used in the WA state of Australia. The system has been employed for the state elections in WA state in 2017. Halderman and Teague [69] and Culnane and Teague [38] analyse the system and report to the authorities of the state that iVote lacks transparency, verifiability and security. It also compromises the privacy of the vote and the voter as it relies on third parties for communicating the vote being cast into the system over the internet [38]. Finally, Culnane and Teague, resonating with many other scholars [17], recommend discontinuing remotely operated elections or at least limiting their use to a great extent.

Roland Wen with his colleagues [100, 144, 143] discuss new cryptographic voting solutions that provide receipt-freeness and verifiability for preferential voting systems such as AV and STV. In a nutshell, their objective is, for online voting, to address the voter coercion vulnerability that preferential systems suffer from by introducing cryptographic solutions. They tackle the problem in such a way that their solution (a) relies on weaker assumptions which are at the same time more realistic for implementation in a online voting system, and (b) is also versatile for implementing various STV and AV schemes securely so that the solution provides verifiability of

[6]See Juels et al [78] for thorough an elaboration on the voter coercion notion.

the process and robustness when conducting an election under their protocol.

In order to achieve their goal, Roland and his colleagues [142] recognise that existing protocols for online voting when trying to prevent voter coercion rely on assumptions which are hard to realise in implementation. Roland as it follows mentions [142] three categories of assumptions one of which typically is assumed in order for these protocol to satisfy the security property meant to protect privacy.

1. some protocols assume that the voter has access to online untappable channels,
2. some other protocols assume an anonymous channel is provided to the voter,
3. other protocols presume existence of local untappable channels and trusted devices for voters to use during casting their votes.

Roland and colleagues find any of the above assumptions impractical for implementing a voting system that preserves privacy. Instead, they introduce a cryptographic solution called masked ballot scheme [143] which provides receipt-freeness and verifiability while depending on weaker assumptions and more pragmatic for implementation. In short, the main idea behind masked ballot voting is to disguise the vote cast with a secret mask that allows the voter to escape being coerced into revealing their vote to an adversary. The scheme proceeds in three stages, namely registration, voting and unmasking. The registration happens before an election begins where a voter is assumed to be provided with an untappable offline channel for producing fake mask transcript for any possible mask. Then during election, the voter can cast their intended vote which is then combined, via cryptographic techniques, with the mask created in advance. Subsequent to casting and recording of votes in an authenticated bulletin board, the votes are processed for tallying to decide on the winners. Their method satisfies the *zero-disclosure* property whereby the only information gained by an adversary is the winners of the election together with verfiability of the outcome.

Also Roland and colleagues [142] create secure abstract data type meant for facilitating implementation of STV schemes in an online voting system for secure integration of voting and counting. The data type is essentially the known abstract mathematical notion of a data type which is interestingly integrated with cryptographic requirements for performing secure arithmetic on votes to decide on winners in an STV scheme. However, implementation of the data type remains a future work to address.

There are a few obvious differences between Roland's work and ours. First of all, Roland and colleagues' work aims at online voting systems where casting and recording is taken care of whereas we only focus on the tallying phase of an election. Second, to guarantee the correctness of tallying, they rely on cryptographic techniques for data auditing, but we use formal methods for producing the end result using verified software and relying on verified evidence checkers for validating data flow. Third, their work remains as a theoretical, although significantly valuable, effort in conducting secure online voting while ours exists in implementation. Moreover, although the secure abstract data type that they discuss allows different STV schemes

to be securely realised within their mathematical construct, it is questionable where such a realisation can happen modularly.

Nonetheless, a future direction for research and engineering would be to examine if Roland's secure abstract data type can be integrated into the Coq component of our framework for creating an encompassing online voting scheme whose various stages of computation, casting, recording and tallying, are all trustworthy through a combination of verifiablity and verification.

7.6 Social Choice Theoretic and Political Studies on STV

Research on STV from the choice theory or political sciences has focused on three aspects; the proportionality profile of STV compared to other PR systems, consequences of adapting STV for the practice of politics in elections and voting, and the strategic effects of STV system. However, to the best of our knowledge, there is still no comprehensive research comparing different versions of STV family against one another from a social choice theory perspective. Farrell and McAllister [54] study STV schemes in Australia and provide a comparison of differences that they have in the quota, number of vacancies, and transferring surplus of elected candidates. Nonetheless, they do not compare the STV schemes alone against one another from the choice theoretic dimension to, for example, a study which STV scheme is "fairer" or more proportional than others.

Also, we know of some work [18] that tries to compute the margin of victory for an STV algorithm. However, there is no study on how changes in the algorithmic content of different STV schemes affects the margin of victory. For instance, as far as we know, no one has yet analysed how using a different formula for updating the fractional transfer value of elected candidates used by an STV algorithm may change the final outcome of tallying votes based on the modified algorithm.

Rae [120] is the first one to study STV from the proportionality perspective. He considers three factors to decide on how proportional STV is. They consist of the district magnitude, the counting algorithm, and ballot structure [120, 53]. Although these three factors are commonly used for studying the proportionality of STV (and also other systems such as PR schemes), the conclusions that researches draw vary significantly. The reason is that for examining the proportionality of the STV schemes, different researches use different methods/formulations for measuring the concept of proportionality [53]. Some conclude that STV is only a semi-proportional system [19, 57, 92, 93, 135]. In contrast, some research considers STV to be the most existing proportional electoral system [19]. However, there is agreement [135, 53] that for STV schemes to reflect the proportionality more, the district magnitude plays a more important role than the other twočiteD.Farrell and it should be at least five [135, 53, 54, 93].

There is research on the relation of STV and the way politics is exercised for attracting votes. Farrell [50], Bowler and Farrelll [22], Bolwer et al [23], and Farrell and McAllister [52] explore vote management strategies and effects such as "friend and

neighbour" voting. Also, Farrell [51] discusses the relation of STV and the Irish political party system in the context of party stability. Moreover, Carty [29], Farrell [49], Katz [79], and Parker [113] discuss how the two main features of STV, namely being candidate-based and preferential (as opposed to PR schemes) influences the emphasis on how politicians practice politics for attracting votes, exercising parliamentary work, becoming localist in their concerns rather than more nationalistic, and the way members of one party compete with other fellow party members during campaigning for election. Also, Farrell [53], Darcy and McAllister [40], Kelley and McAllister [80], and Robson and Walsh [122] discuss on how placing names of candidates on ballots affects who may have higher chances for being elected.

Finally, some have studied STV from the computational social choice theoretic aspects. Dummett [47], Nurmi [111], and Barm and Fishburn [13] examine the behaviour of STV against properties such as majoritarian criteria where a candidate (also a party) receiving more first preferences is expected to have higher chances for winning. They criticise STV for contradicting the majoritarian rule. Also, there has been some effort [18] in computing the margin of victory of an election based on an STV scheme. Finally, Walsh [141] discusses the computational NP-hardness of manipulating an STV algorithm.

Chapter 8

Conclusion

We shall not cease from exploration and the end of all our exploring will be to arrive where we started and know the place for the first time.[1] This monograph has introduced a framework for modularly engineering tally software for verifiably correct, trustworthy, transparent computation with various STV algorithms. The framework comprises two standalone components. The first component is developed for constructing proven correct vote counting programs with various STV algorithms. Each program produced from this component generates a run-time certificate for every instance of computation carried out which allows auditability of the tallying process of computing the winners. The second component of the framework allows synthesising verified certificate verifiers for checking the correctness of the certificates produced by any program used for counting votes with an STV algorithm.

Both components of the framework formalise and verify the similarities of STV algorithms as an abstract machine and realise differences of various STV algorithms as instantiations into the machine. They provide a uniform and modular process of (a) producing tools that carry out verified computation with an STV algorithm and (b) synthesising means for verifying the computation carried out independently of the computation's source code. The framework components also provide flexibility and ease for adapting and extending them to a variety of STV schemes. Moreover, we automate almost all proofs that we establish in Coq, HOL4 and CakeML so that new instances of verified and verifying tools for computation with a variety of STV algorithms can be created with no (or minimal) extra verification. Also, our experimental results with executable code demonstrate the feasibility of deploying the framework for verifying real size elections having an STV counting mechanism.

We minimise the trusted base in the correctness of the tools synthesised by using the Coq and HOL4 theorem provers and the ecosystem of CakeML as the technical basis. The trustworthiness of the tools created from our framework rests on the certifying algorithms technique and using a proven trustworthy technical basis. Run-time certification provides auditability, and thereof transparency, of the tallying phase. On the other hand, formal verification of the software produced by relying on means that have a minimal trusted computing base removes several trusted layers in computing with our tools. As a result, we approximate satisfying the universal verifiability of

[1] T. S. Elliot

tallying through minimising the trusted computing base with a quality that no preceding work has succeeded in even coming close to.

It is noteworthy that our framework is not limited to being integrated only with an end-to-end verifiable voting system. As our work concerns the tallying phase only, any voting system, regardless of what notion of verifiability they choose for the system to implement, that does not rely on homomorphic tallying of votes can easily adapt our tools. Indeed, even unverifiable voting systems can employ tools created from our framework. Adapting our tools at least provides guarantees in the correctness of the tallying outcome with respect to the inputs to the tallying software.

Moreover, our framework accommodates Alternative Voting (AV) schemes as well. These schemes also referred to as instant-run-off voting, are used in several countries such as Australia, the United States, and the UK for important elections such as electing parliament members. This family of algorithms is basically a subclass of STV schemes. The mechanism of these algorithms is exactly as an STV scheme except that they are used for electing one winner in an electoral district. In other words, an AV algorithm is an STV algorithm where the number of vacancies is one. Since there is only one winner, the rule elect only applies once. As a result, AV algorithms do not have a transition for transferring votes of an elected candidate.

Based on our thorough discussion in Section 4.4, it is obvious how our framework realises an AV system. All of the rules are formalised like an STV scheme, for example the ACT STV, except that one needs including a dummy transfer-elected transition to instantiate the generic STV record in the Coq component to successfully discharge the semantics constraints of the machine. For the second component, one can simply define the transfer-elected as the logical falsehood. Consequently, for checking a certificate, time is not consumed for inspecting a case that never occurs in a certificate for computation with an AV algorithm.

We shall next elaborate on some of the possibilities that our work has opened for future endeavours and accomplishments. The author wishes to see the day when his work has come to be integrated into an encompassing framework for addressing the whole end-to-end, or any other type of, verifiability of voting systems for STV.

8.1 Future Work

We discuss the possibilities for future improvements to our work under three headings. First, we explain where and how to enhance the first component of the framework created in Coq. Then, we elaborate on ways to improve the second component developed in HOL4 and CakeML. Last, we outline how the whole framework can be used for further work in the area of election security and social choice theory.

8.1.1 Tie-breaking

In our framework we facilitate but not thoroughly implement tie-breaking methods. A comprehensive formalisation, implementation and verification of various possible tie-breaking remains for future investigations. There are a few main reasons explaining the manner in which we have treated tie-breaking so far.

- First of all, there are various tie-resolution methods [94, 145, 71, 142] for deciding whom to eliminated among equally weakest candidates who are tied at the least amount of votes among other continuing candidates.
- Second, different STV schemes use different techniques for dealing with ties. This together with the first point make us conclude that for the formalisation and verification of the generic STV in Coq and HOL4, we do not need to, and indeed must not, include any of these specific methods of tie-breaking in the specification of the generic machine. The reason is obvious; none of the *specific methods* permeates the STV algorithms generally and hence does not realise a core part of the STV.
- Third, aside from the fact that tie-resolution methods vary among different STV algorithms, specification of the tie-breaking described in textual explanation of an individual STV scheme is so vague that leaves us with more than one interpretation of the method. Consequently, we can not decide for sure which interpretation to settle on and implement while instantiating the generic machine in either Coq and HOL4.
- Fourth and related to the above points, as far as the specification of the generic elimination transition is concerned, for stipulation of the weakest candidate to eliminate, it suffices to include the simple requirement that the candidate must have accumulated an amount of votes that is less than or equal to other continuing candidates (which is what currently our specification of the elimination rule for generic STV expresses). This choice has a number of advantages:

    1. it makes the framework more general and encompassing of various STV algorithms with different tie-breaking techniques. By under-specifying the tie-breaking, we allow the properties and proofs established for the generic machine to hold for a broader variants of STV algorithms. This could no longer be accomplished had we chosen to specify the tie-breaking in further details than what it currently is for the generic elimination transition.
    2. it provides the user with a greater flexibility in that it accommodates different interpretations of the tie-breaking of an STV algorithm. The framework allows instantiating the existential variable appearing in the definition of the elimination rule in Coq with a computational entity that captures the exact interpretation that the user has from textual document of the STV scheme at hand. Similarly for the HOL4 component, upon instantiation of the generic machine one decides on their interpretation of the tie-breaking,

formalises that understanding, implements it and then verifies the implementation correct against their formal specification. Therefore the underspecification of tie-resolution opens the chance for a user to formally realise, verify and synthesise their own interpretation, instead of ours, of the tie-breaking method of an STV scheme using our framework.

We must note that our experiments with historical data of the Legislative Assembly elections in ACT illustrated in Subsections 4.5.4 shows that no tie actually occurs in those instances. Therefore, we are sure that both extracted Haskell program from Coq and the synthesised certificate verifier for ACT STV compute correctly. On the other hand, we do not have access to any other real historical data of any election. Moreover, the intention behind experiments with our software has never been to unveil possible counting errors in any past election. As a result, we simply avoid falling into the trap of highly ambiguous documents that do not substantiate enough material for confidently interpreting the tie-breaking mechanism of ACT STV, or any other STV scheme, in a correct way.

As mentioned earlier, There are various ways according to which different STV algorithms decide on tie-breaking for elimination of a continuing candidate, if and when such a situation occurs. We shall next mention the tie-resolution methods of which we are aware and subsequently describe how best one may use our framework for incorporating them when instantiating the generic machine. But first we define what is meant by "weakest candidates" so that the references is clear.

Definition 8.1 (weakest candidates) *Assume C_1, ..., C_k are some continuing candidates in an instance of computation with an STV algorithm. Assume in this instance of the algorithm execution, we encounter an intermediate state st. We call candidates C_1, ..., C_k the weakest candidates at this stage if and when the following hold:*

- *every C_j and $C_{j'}$ where j and $j' \in \{1, \cdots, k\}$, have the same tally,*
- *for any other continuing candidate C_i where $i \notin \{1, \cdots, k\}$, the tally of C_i is strictly bigger than C_j for any $j \in \{1, \cdots, k\}$.*

Assume in an execution of an STV algorithm on input data including an initial list of competing candidates, the execution reaches a state st where an application of the elimination transition must apply to remove one of the weakest candidates C_1, ..., C_k, from the tallying process. Let us assume $\mathcal{C}_w$ represents the duplicate-free list consisting of these weakest candidates. Also suppose the chronological list of machine states visited before reaching to the current state st is $[st_0, \cdots, st_n]$ where st_0 is the initial state of the machine where no transition has yet applied. Moreover, assume a counter i whose value ranges between the natural numbers 1 and n. Having explained the preliminaries, the following are some tie-resolution techniques of which we know that decide on the one who is eliminated.

- *Forwards tie-breaking.* initialise the value of i with 1. To break the tie between the weakest candidate perform the following steps:

1. eliminate the candidate $C_i \in \mathcal{C}_w$ whose tally at st_i is less than the other candidates $C_j \in \mathcal{C}_w$.
2. if no single candidate exists whose vote is less than others in the list $\mathcal{C}_w$, increment i by one and repeat the above process for state st_i.

- *Backwards Tie-breaking.* let the initial value of i to be n. Then break the tie as follows:

  1. remove the candidate $C_i \in \mathcal{C}_w$ whose tally amount is less than other candidates in $\mathcal{C}_w$.
  2. if no single candidate exists whose vote is less than others in the list $\mathcal{C}_w$, decrease i by one and repeat the above process for state st_i.

- *Arbitrary tie-resolution.* take two of the weakest candidates in $\mathcal{C}_w$ in whichever arbitrary order, for instance alphabetically against their family name, then eliminate the one whose tally is less than the other.

- *Random tie-breaking.* randomly select two candidates in $\mathcal{C}_w$, compare their tallies and then remove the weakest one from the process.

- *Coombs tie-breaking.* Choose the candidate C_i in $\mathcal{C}_w$ with the most last place votes compared to other candidates in $\mathcal{C}_w$ and eliminate him/her.

- *Borda tie-resolution.* Choose the candidate $\mathcal{C}_i$ in $\mathcal{C}_w$ whose Borda score is lowest compared to other candidates in $\mathcal{C}_w$.

The astute reader immediately identifies issues with each of the above methods. In fact, each of the techniques have been criticised for different reasons. For example, there can be a situation that two more candidates are tied at all of the previous steps of the computation. Such tie cases are referred to as *strong ties* and can, although the possibility of their occurrence is extremely low, create difficulties in how to proceed with the elimination in a fair manner. Neither of the forwards or backwards tie-breaking can finalise the decision on whom to remove in an instance of a strong tie. Indeed, many STV algorithms are either silent on such strong tie cases or mostly instruct a random coin toss to break the tie between two candidates. However, the question remains what if three candidates are tied in all previous states of the counting? How would the coin toss proceed that fairly eliminates one among the three tied candidates? Or some algorithms speak merely about randomly eliminating a candidate in a strong tie but do not further explain through what exact random process! As another example, methods such as random selection of two candidates among all of the other weakest candidates opens the way on fate to decide who the winners are. This means that instead of will of voters expressed through their votes, as the main decider on who shall be excluded and who shall eventually be elected, randomness effects the outcome. This stands in a marked contradiction to democratic reasons for conducting an election. Democracy literally means rule of the people which in modern world is expressed through voting in elections. The end result of vote counting

based on an STV algorithm, including tie-breaking techniques, if they intend to be democratic, must solely be determined by unfolding the content of votes cast into the voting system (or the election ballot box). If randomness came to finalise winners of an election, then in such cases democracy would be understood as the rule of fate but not people.

To conclude our discussion on tie-breaking, we shall remark on how a user may implement their tie-breaking using data structure of the Coq component. As you see in Figure 3.6 which illustrates the data structure used for implementing the machine states, we formalise tallies as a list of closures. This list can be then used for implementing forwards or backwards tie-breaking method above. However, the author himself would hesitate to go ahead and use the list of tally closures to this end. The main reason is that it would be highly computationally inefficient. This in turn stems from the fact that when going one or some steps backwards for comparing tallies of weakest candidates at those previous states, they have to recompute the tally again. There is a more efficient way of dealing with this computational problem which we explain further below.

In the Coq component, tallies are updated upon applications of the count transition where votes placed in the pre-state are counted and then allocated to continuing candidates based on next preferences expressed on those votes. In order to efficiently break ties, using forwards or backwards method, one can add formal (computational) statements to the body of the count transition upon instantiation of the transition with their specific STV count rule to implement an associated look-up table for candidates and their tally. Such a table is, from an abstract data type point of view, a mathematical construct of type $\mathbf{List}(C \times \mathbf{List}(\mathbb{Q}))$[2]. In other words, one has access to a list where every element of the list is pair. The first component of every pair is the name of a candidate and the second component is the list of the amount of votes he/she has attracted throughout the counting process. The advantage of having this look-up table is obvious; one need not recompute the tallies from their closure and can simply use the recorded tally amounts in the list which is significantly more efficient for tie-breaking purposes.

The reader may wonder at this point why on earth we, knowing the computational cost of using tally closures, have not formally included such a look-up tally list in the first place. The main reason for our choice is simplicity, precision and the elegance of formalisation that we obtain using the closure list for tallies. Since we construct tallies as functions, it is readily given that every candidate has one unique tally. Had we chosen to go with the look-up table instead, we would have to include many statements just to capture what the tally formalise as a function automatically provides. This in turn has a secondary effect that including lesser statements for expressing the obvious enhances the readability of our formalised transitions which itself eases understandability of the system. In contrast, look-up table would require formalisation and verification of some functionalities that demand some effort from

[2]See Figure 2.1 as the reference to what these symbols stand for.

the user to explore them in the body of our code to know what they are, what they do and how. This would reduce the abstraction level of our system where the user is exposed to some details which they would not need going through with the closure list for tallies.

Accommodating tie-breaking in our HOL4 formalisation, simply amounts to specifying the type of tallies in the data structure of the machine to be $\mathbf{List}(C \times \mathbf{List}(\mathbb{Q}))$. The look-up table does not introduce any new challenges or difficulties for us here because of two reasons. First of all, verifiers process information from certificates which are already association lists and not functions. Therefore, one has to deal with data encapsulated into lists and consequently we have to include some formal statements to express the fact that at every stage every candidate has a tally and that tally is unique. Second, we have indeed already had the pleasure of specifying such properties, implementing them and verifying the implementation as demonstrated in Subsection 5.3.2. Therefore the task is already taken care of and the only change required is to modify the type of the tallies in the data structure given in Figure 5.4.

8.1.2 Future Work on the Coq Component

In general, one can advance this component of the framework from three dimensions.

- **Extensibility.** This framework component accommodates formalisation and verification of a large class of STV schemes. However, it does not currently support producing vote counting software for STV algorithms that use randomness for transferring a portion of an elected candidate's votes. We have not included random mechanisms in this component because it would require outsourcing the computation to another software. The problem with outsourcing is it enlarges the TCB significantly as both the outsourcing mechanism and the software used for carrying random computation introduce unverified layers.

  Moreover, Meek STV uses a fix-point mechanism that alters the quota to reach a termination for the counting process. Our Coq component allows formalisation of this kind of STV scheme. It is also possible to verify (meaning discharging the sanity checks) each of the formally specified Meet STV transitions except for transfer-elected. The Coq component at the moment does not support verification of the transfer-elected transition of Meek STV [72] and any other STV scheme which uses such a mechanism for transferring surplus votes. The main obstacle is the limitations that the complexity measure imposes on verification of such transfer-elected transitions. When using a Meek way of transferring surplus votes, neither of the components of the complexity measure reduces. Therefore, verification of this kind of transition, and subsequently the whole algorithm, is not currently possible within our framework.

  It is noteworthy that the HOL4-CakeML component can indeed with some minor adjustments produce a certificate verifier for Meek STV or any STV whose transfer-elect mechanism is similar to Meek's. Nonetheless, a furture direction for extending the component is to replace the termination theorem which relies

on the measure with a different termination proof in such a way that Meek STV is supported without losing the usability of the Coq component as it stands.

- **Performance.** There are at least two ways to optimise the efficiency of computing with vote counting tools created from this component. Applications of the count and elect transitions are computationally costlier than others'. There are two reasons for such costs. The first one is somewhat inherent to using theorem provers' libraries. Relying on functions and assertions already proven verified in a theorem prover's libraries is a sword with two edges. On the one hand, it reduces the workload when building the framework or extending it to include ones' favourite STV which altogether improves the usability of the framework. On the other hand, as these functions are mostly not tails recursive, computing with them is time-consuming.

  More specifically in the context of our work, there are functions such as map and filter called from Coq's library for lists and used in the count and elect transition formalisation, which is verified but are not tail recursive. One can consider programming tail recursive extensionally equivalent functions for the non-tail recursive ones in Coq's libraries and use the tail recursive functions whenever possible. Note that because of technicalities related to the extensionality-intensionality problem, only occurrences of a function call on an input can be replaced by an extensionally equivalent tail recursive version.

  One realises that performance involves the structure underlying data as well and is not only about implementing efficient algorithms. As you see in Figure 3.6, the first part of an object whose type is ballot is a list of candidates. Therefore, throughout the process of formalisation, any ballot has to obey this type no matter what transition applies. As a consequence of this limitation, when applying the count transition for allocating votes to candidates based on preferences, no candidate name is removed from a ballot that has already been counted. This means ballots do not become shorter as the counting progresses. To make sure that when applying the count transition every ballot is allocated to the next preference stated on the ballot, we program the function that starts at the head position of a ballot and ignores candidates on the list who have already been elected or eliminated (are not in the continuing list) and continues until it reaches to the first continuing candidate on the ballot and assigns the vote to him/her.

  The above choice of data structure for ballots makes computing the first preference of each ballot quadratic. Based on the profiling results of the Haskell vote counting programs extracted, it costs us twenty to thirty percent of the computation time. In order to make the computation linear and subsequently faster, we can decompose the type of ballots to consists, as its first component, a pair of lists the first of which includes that part of the ballot already processed (consisting of elected or eliminated candidates). The second part of this pair would be the list the remainder of the vote yet to be dealt with. The advantage of this formalisation is that the head position of the second list is the next

continuing preference of the ballot accessed in constant time. Therefore, computing the first preferences reduces to linear time as one needs considering if the head position of the second part of each ballot belongs to the list of elected or continuing candidates.

- **Automation.** Currently instantiating the generic STV for obtaining a verified version of specific STV algorithm requires five hundred lines of encoding. One hundred lines consist of a formal definition of transitions which is inevitable and hardly can be fully automated. Half of the remaining four hundred lines can be automated so that a user merely needs dealing with two hundred lines of proofs. One can automate the two hundred lines simply by writing Ltacs in Coq which would take care of proof goals that have similar proof structure across different instantiations. However, full automation of the proofs due to the use of dependent types which makes computation and proofs interwoven is a challenging task.

- **Algorithm Analysis/Computational Social Choice.** At the moment, we do not know of any existing framework that accommodates a computational comparative study on STV algorithms in a modular way. Our framework allows one to obtain provably correct executable software that in turn can be used for analysing different STV algorithms against one another based on social choice theoretic factors such as fairness. Moreover, the framework accommodates modifying one single STV algorithm in order to compare it, from a computational perspective, with different versions of the algorithm itself.

  Furthermore, the Coq component not only permits computational analysis of the STV family but also facilitates syntactic comparisons of the schemes as well. Consequently, instead of struggling with textual convoluted descriptions of the schemes, one can simply use our framework to study them from a purely syntactic, instead of a semantic interpretational, point of view.

8.1.3 Future Work on the HOL4-CakeML Component

Similar to the Coq component, there is room for improving the performance of tools constructed from this part of the framework. Once again, one can program tail recursive provably extensionally equivalent versions of the non-tail recursive functions in HOL4 library and use them instead.

Additionally, one can reformulate some of our functions in such a way to optimise the computation. In particular, recall the definition of a valid step given in Figure 6.9. As you see in the definition, eliminating a candidate is highly costly. The main reason is the way that we have defined a valid step. The definition does not happen based on pattern machine on the type of the premise and conclusion of the assertion. It generically takes as input and output two machine states j_0 and j_1 and checks if any of the transitions' semantics is satisfied representing a correct jump from the premise j_0 to the conclusion j_1. The problem is when the correct transition from j_0 to j_1 is, in fact, an application of the elimination rule. However, since we do not define

valid step based on pattern matching against the constructors of j_0 and j_1, when the verifier tries to figure out who is the candidate that has been removed, it has to start from the head of the list of all competing candidates and apply elimination of one candidate, check whether the semantics of the rule are satisfied or not. The verifier has to continue in this way until it verifies that indeed an application of elimination has happened and who has been removed.

We can reformulate the definition simply by pattern matching against the type of machine state that j_0 and j_1 are. When knowing that j_0 and j_1 are respectively $NonFinal(ba_0, t_0, p_0, bl_0^1, bl_0^2, e_0, h_0)$ and $NonFinal(ba_1, t_1, p_1, bl_1^1, bl_1^2, e_1, h_1)$ machine states in order to check if an application of elimination has occured and who has been the removed candidate, one simply computes the member who is in h_0 but not h_1. This improves the computation considerably as many costly non-tail recursive functions in the elimination body are not called several times.

One can reduce the TCB of this framework component as well. The only place to improve the verification level of the component is to formally verify the parser. As it stands, the parser consists of about 150 lines of formally unverified code. We have tested the parser for correctness through a partial examination on input data. Moreover, the fact that the verifier validates certificates generated from computation using tools extracted from the Coq component adds assurance to the reliability of the parser. However, since our standards when it comes to trustworthiness are high, this part of our framework needs improvement.

There is ambition on integrating research on computing the margin of victory of elections with an STV scheme and the certificate verifier. More specifically, There seem to be opportunities for combining the work of Blom et al [18] with the design and implementation of certificate verifiers. Every verifier currently rejects a certificate if simply the ordering of two candidates in a ballot has maliciously been changed. This appears restrictive at first glance. Nonetheless, we already know that elections having an STV counting mechanism have a small margin of victory. Therefore, from this perspective, it is reasonable for a verifier to take it hard on small anomalies encountered in a certificate. This restriction can be loosened by implementing verifiers in a way that the tolerate malicious tampering with a certificate up to a threshold which approximates the margin of victory of the election. Engineering the framework for producing efficient verifiers that take into account the margin of victory, given the verification requirements that demand performing verified checks on data, is a challenging problem to tackle.

8.1.4 Future Work Using the Framework

We have an idea for integrating the framework into a voting system that uses homomorphic tallying technique. Here, we merely outline a solution and leave details for the future. Recall that from our discussion in Subsection 1.3.1 and Section 7.5 that advocates of homomorphic tallying of votes aim at preventing the possibility of voter coercion in elections with an STV counting scheme. As a result, they would not allow issuing a certificate for computation which contains information on the ballots

recorded in the system even after having anonymised the votes.

In a nutshell, software using homomorphic tallying can generate an encrypted version of a certificate. Then, certificate verification can be combined with the decryption mechanism to validate the computation performed by the software for computing the winners. In order to reduce the costs of computation, as certificate verifiers process two consecutive machine states at a time, decryption can also happen stepwise instead of decrypting the whole certificate and then executing the verifier. Of course, transparency of the certificate validation process reduces compared to the mixnet approach simply because the certificate is no longer publicly accessible. However, one can establish guarantees that (a) encrypting and decrypting the certificate is verifiable and (b) it is indeed the certificate verifier that is being executed on the certificate to check its validation.

Since voter coercion is the main debate for using homomorphic tallying alongside mixnets, there is another way for providing confidence in voter coercion resistance of the voting system without relying on a homomorphic tallying technique. The basic idea stems from secure Multi-Party Computation (SMPC) [118]. Here, one can use a mixnet, the certification technique and certificate checking but also prevent voter coercion problem.

In short, once the system has cast, recorded, and anonymised the votes, it can decrypt them and ready them for being counted. Next, the voting system can use a certifying vote counting program for computing the winners and producing a certificate for the instance of computation performed. However, the decryption, counting votes, and certification happen in a so-called enclave [87] where there is no public access to it but there is guarantee into the preservation of data integrity so that voters are confident about the integrity of the computation happening in an enclave. The voting system can keep the certificate private residing in the enclave. However, they must provide secure channels for voters to run the certificate verifier, remotely, on the certificate. The certificate checker source code can be publicly accessible so that voters know how it performs the validation.

An advantage of this approach is that companies wishing to keep their implementation of the STV algorithm used privately. Nonetheless, There are significant difficulties to overcome for making the MPC approach to work securely in the context of elections. For example, there need to guarantee that what is being held in the enclave is an actual certificate as output by the vote counting software. Also when a voter remotely runs the verifier, there must be proof that it is indeed being executed on the certificate. Additionally, from the authorities perspective, one has to prove that no attacker can run, together with the verifier execution, their malicious code so that no damage is done to hardware holding the certificate.

Another direction of future work is providing a thorough attacker model, and then determining against which kind attackers the verified implementation can defend

itself[3]. As we advocate formal methods for demonstrating correctness of assertions, the model should be formally specified and reasoned for in order to substantiate the security properties.

Finally, when reading about the Schulze voting scheme to review Pattinson and Tiwari's work [115], ideas from our earlier version of the Coq component came back to my mind that can be used for building an entirely new different framework. The idea mainly concerns constructing a framework for modular formalisation and verification of vote counting properties. For example, consider the Condorcet property where a winner of the election is the candidate who defeats every other candidate in a one-one comparison. Schultz method is an instance of a vote counting scheme which satisfies this property. Now, one would desire to create a framework that accommodates formally proving such properties for every voting scheme that has the properties. In order to outline the macro-level design of such a framework, we draw on generic programming. Here, in a generic style, one can specify the property and prove generically that an instantiation of the generic types also has the property. Then in different modules, one can instantiate the types with gigantic inductive types each of which stands as the formal counterpart of a voting scheme. Then in each module, one can establish that the inductive definitions each satisfy the generic requirements. Therefore, in a modular, one can demonstrate the properties for various voting algorithms.

[3]The author would like to mention that one of the anonymous reviewers has pointed out at this idea for further work.

Bibliography

1. ACT, E. C., . Act past elections. https://www.elections.act.gov.au/elections_and_voting/past_act_legislative_assembly_elections. (cited on page 89)

2. ACT, E. C., . Fact sheets, hare-clark. https://www.elections.act.gov.au/publications/act_electoral_commission_fact_sheets/fact_sheets_-_general_html/elections_act_factsheet_hare-clark_electoral_system. (cited on pages 62 and 64)

3. Alkassar, E.; Böhme, S.; Mehlhorn, K.; and Rizkallah, C., 2014. A framework for the verification of certifying computations. *J. Autom. Reasoning*, 52, 3 (2014), 241–273. (cited on page 139)

4. Andrews, P. B., 1986. *An introduction to mathematical logic and type theory - to truth through proof*. Computer science and applied mathematics. Academic Press. ISBN 978-0-12-058535-9. (cited on page 15)

5. ane, C.; Ryan, P. Y. A.; Schneider, S. A.; and Teague, V., 2015. vvote: A verifiable voting system. *ACM Trans. Inf. Syst. Secur.*, 18, 1 (2015), 3:1–3:30. doi:10.1145/2746338. https://doi.org/10.1145/2746338. (cited on pages 11, 151, and 153)

6. Arkoudas, K., 2000. *Denotational Proof Languages*. PhD Thesis MIT, Mmssachusetts, USA. (cited on page 140)

7. Arkoudas, K. and Rinard, M. C., 2005. Deductive runtime certification. *Electr. Notes Theor. Comput. Sci.*, 113 (2005), 45–63. (cited on page 140)

8. Arora, S. and Safra, S., 1992. Probabilistic checking of proofs; A new characterization of NP. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, 2–13. doi:10.1109/SFCS.1992.267824. https://doi.org/10.1109/SFCS.1992.267824. (cited on page 139)

9. Avizienis, A., 1985. The n-version approach to fault-tolerant software. *IEEE Trans. Software Eng.*, 11, 12 (1985), 1491–1501. doi:10.1109/TSE.1985.231893. https://doi.org/10.1109/TSE.1985.231893. (cited on page 139)

10. Baader, F. and Nipkow, T., 1998. *Term rewriting and all that*. Cambridge University Press. ISBN 978-0-521-45520-6. (cited on page 15)

11. Babai, L.; Fortnow, L.; Levin, L. A.; and Szegedy, M., 1991. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, 21–31. doi:10.1145/103418.103428. https://doi.org/10.1145/103418.103428. (cited on page 138)

12. Barendregt, H. P.; Dekkers, W.; and Statman, R., 2013. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press. ISBN 978-0-521-76614-2. http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types. (cited on page 14)

13. Barms, S. and Fishburn, P. C., 1984. *Choosing an Electoral System: Issues and Alternatives*, chap. Some Logical Defects of the Single Transferable Vote. Praeger, New York. (cited on page 156)

14. Beckert, B.; Goré, R.; and Schürmann, C., 2013. Analysing vote counting algorithms via logic - and its application to the CADE election scheme. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, 135–144. doi:10.1007/978-3-642-38574-2_9. https://doi.org/10.1007/978-3-642-38574-2_9. (cited on pages 66, 76, and 147)

15. Beckert, B.; Goré, R.; and Schürmann, C., 2013. On the specification and verification of voting schemes. In *E-Voting and Identify - 4th International Conference, Vote-ID 2013, Guildford, UK, July 17-19, 2013. Proceedings*, 25–40. doi:10.1007/978-3-642-39185-9_2. https://doi.org/10.1007/978-3-642-39185-9_2. (cited on page 147)

16. Benaloh, J.; Moran, T.; Naish, L.; Ramchen, K.; and Teague, V., 2009. Shufflesum: coercion-resistant verifiable tallying for STV voting. *IEEE Trans. Information Forensics and Security*, 4, 4 (2009), 685–698. doi:10.1109/TIFS.2009.2033757. https://doi.org/10.1109/TIFS.2009.2033757. (cited on pages 11, 151, 152, and 153)

17. Benaloh, J.; Rivest, R. L.; Ryan, P. Y. A.; Stark, P. B.; Teague, V.; and Vora, P. L., 2015. End-to-end verifiability. *CoRR*, abs/1504.03778 (2015). http://arxiv.org/abs/1504.03778. (cited on pages 2, 151, and 153)

18. Blom, M. L.; Stuckey, P. J.; and Teague, V. J., 2018. Computing the margin of victory in preferential parliamentary elections. In *Electronic Voting - Third International Joint Conference, E-Vote-ID 2018, Bregenz, Austria, October 2-5, 2018, Proceedings*, 1–16. doi:10.1007/978-3-030-00419-4_1. https://doi.org/10.1007/978-3-030-00419-4_1. (cited on pages 4, 75, 105, 155, 156, and 166)

19. Blondel, J., 1969. *An Introduction to Comparative Government*. Weidenfeld and Nicolson, London. (cited on page 155)

20. Blum, M. and Kannan, S., 1989. Designing programs that check their work. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, 86–97. (cited on page 138)

21. Boldo, S.; Jourdan, J.; Leroy, X.; and Melquiond, G., 2013. A formally-verified C compiler supporting floating-point arithmetic. In *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, 107–115. doi:10.1109/ARITH.2013.30. https://doi.org/10.1109/ARITH.2013.30. (cited on page 151)

22. Bowler, S. and Farrell, D., 1996. *British Elections and Parties Yearbook*, chap. Voter Strategies under Preferential Electoral Systems: A Single Transferable Vote Moch Ballot Survay of London Voters. Cass, London. (cited on page 155)

23. Bowler, S.; Farrell, D.; and McAllister, I. Consistency campaigning in parliamentary systems with preferential voting: Is there a paradox? *Electoral Studies*, 15, 4 . (cited on page 155)

24. Brooks, L. and Griffits, A., 2017. NSW council elections: Computer 'guesstimate' might have ignored your vote. *ABC News*, (9 2017). Available online at http://www.abc.net.au/news/2017-09-14/computer-algorithms-may-sway-local-council-elections/8944186. (cited on page 4)

25. Bulwahn, L.; Krauss, A.; Haftmann, F.; Erkök, L.; and Matthews, J., 2008. Imperative functional programming with isabelle/hol. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, 134–149. doi:10.1007/978-3-540-71067-7_14. https://doi.org/10.1007/978-3-540-71067-7_14. (cited on page 141)

26. Burton, C.; Culnane, C.; Heather, J.; Peacock, T.; Ryan, P. Y. A.; Schneider, S.; Srinivasan, S.; Teague, V.; Wen, R.; and Xia, Z., 2012. A supervised verifiable voting protocol for the victorian electoral commission. In *5th International Conference on Electronic Voting 2012, (EVOTE 2012), Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC, July 11-14, 2012, Castle Hofen, Bregenz, Austria*, 81–94. https://dl.gi.de/20.500.12116/18227. (cited on pages 151 and 152)

27. Burton, C.; Culnane, C.; and Schneider, S., 2015. Secure and verifiable electronic voting in practice: the use of vvote in the victorian state election. *CoRR*, abs/1504.07098 (2015). http://arxiv.org/abs/1504.07098. (cited on page 153)

28. CADE. Single transferrable vote algorithm. http://www.cadeinc.org//bylaws. (cited on page 23)

29. Carty, R. K., 1981. *Party and Parish Pump: ELectoral Politics in Ireland*. Wilfrid Laurier Press, Waterloo. (cited on page 156)

30. Chaum, D., 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24, 2 (1981), 84–88. doi:10.1145/358549.358563. http://doi.acm.org/10.1145/358549.358563. (cited on pages 11 and 152)

31. Chlipala, A., 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. ISBN 978-0-262-02665-9. http://mitpress.mit.edu/books/certified-programming-dependent-types. (cited on page 14)

32. Cochran, D., 2012. *Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms*. Ph.D. thesis, ITU. (cited on page 148)

33. Cochran, D. and Kiniry, J. R., 2013. Formal model-based validation for tally systems. In *E-Voting and Identify - 4th International Conference, VoteID 2013, Guildford, UK, July 17-19, 2013. Proceedings*, 41–60. doi:10.1007/978-3-642-39185-9_3. https://doi.org/10.1007/978-3-642-39185-9_3. (cited on page 147)

34. Coleman, T. and Teague, V., 2007. On the complexity of manipulating elections. In *Theory of Computing 2007. Proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007). January 30 - Febuary 2, 2007, Ballarat, Victoria, Australia, Proceedings*, 25–33. http://crpit.com/abstracts/CRPITV65Coleman.html. (cited on page 76)

35. Coquand, T. and Huet, G. P., 1988. The calculus of constructions. *Inf. Comput.*, 76, 2/3 (1988), 95–120. doi:10.1016/0890-5401(88)90005-3. https://doi.org/10.1016/0890-5401(88)90005-3. (cited on page 14)

36. Cortier, V.; Galindo, D.; Küsters, R.; Müller, J.; and Truderung, T., 2016. Verifiability notions for e-voting protocols. *IACR Cryptology ePrint Archive*, 2016 (2016), 287. (cited on page 152)

37. Culnane, C.; Heather, J.; Joaquim, R.; Ryan, P. Y. A.; Schneider, S.; and Teague, V., 2014. Faster print on demand for prêt à voter. In *2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections, EVT/WOTE '14, San Diego, CA, USA, August 18-19, 2014*. https://www.usenix.org/conference/evtwote14/workshop-program/presentation/culnane. (cited on pages 151 and 152)

38. Culnane, C. and Teague, V. Submission to the parliament of western australia: Inquiry into the administration and management of the 2017 state general election. http://www.parliament.wa.gov.au/parliament/commit.nsf/(InqByName)/D7D9DA3F60350E524825814D00292E48?OpenDocument. (cited on page 153)

39. Dahlweid, M.; Moskal, M.; Santen, T.; Tobies, S.; and Schulte, W., 2009. VCC: contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, 429–430. (cited on page 139)

40. Darcy, R. and McAllister, I., 1990. Ballot position effects. *Electoral Studies*, 9 (1990), 5–17. (cited on page 156)

41. Dawson, J. E.; Goré, R.; and Meumann, T., 2015. Machine-checked reasoning about complex voting schemes using higher-order logic. In *E-Voting and Identity - 5th International Conference, VoteID 2015, Bern, Switzerland, September 2-4, 2015, Proceedings*, 142–158. doi:10.1007/978-3-319-22270-7_9. https://doi.org/10.1007/978-3-319-22270-7_9. (cited on pages 146, 148, and 150)

42. de Nivelle, H. and Piskac, R., 2005. Verification of an off-line checker for priority queues. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, 210–219. doi:10.1109/SEFM.2005.54. https://doi.org/10.1109/SEFM.2005.54. (cited on page 141)

43. Dennis, G. and Yessenov, K. Forge. http://sdg.csail.mit.edu/forge/. (cited on page 147)

44. Dennis, G.; Yessenov, K.; and Jackson, D., 2008. Bounded verification of voting software. In *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, 130–145. doi:10.1007/978-3-540-87873-5_13. https://doi.org/10.1007/978-3-540-87873-5_13. (cited on page 147)

45. DeYoung, H. and Schürmann, C., 2011. Linear logical voting protocols. In *E-Voting and Identity - Third International Conference, VoteID 2011, Tallinn, Estonia, September 28-30, 2011, Revised Selected Papers*, 53–70. doi:10.1007/978-3-642-32747-6_4. https://doi.org/10.1007/978-3-642-32747-6_4. (cited on pages 141, 142, and 150)

46. Droop, H. R., 1881. On methods of electing representatives. *Journal of the Statistical Society of London*, 44, 2 (1881), 141–202. http://www.jstor.org/stable/2339223. (cited on pages 6 and 23)

47. Dummett, M., 1992. *Principles of ELectoral Reform*. Oxford University Press, Oxford. (cited on page 156)

48. Elmasry, A.; Mehlhorn, K.; and Schmidt, J. M., 2012. An o(n+m) certifying triconnnectivity algorithm for hamiltonian graphs. *Algorithmica*, 62, 3-4 (2012), 754–766. doi:10.1007/s00453-010-9481-2. https://doi.org/10.1007/s00453-010-9481-2. (cited on page 139)

49. Farrell, B., 1985. *Representatives of the People?*, chap. Ireland: From Firends and Neighbours to Clients and Partisans. Gower, Aldershot. (cited on page 156)

50. Farrell, B., 1988. *Introduction: Western Europe Cabinets in Comparative Perspective*, chap. Irland. Palgrave Macmillan, London. (cited on page 155)

51. Farrell, D., 1994. *How Partise Organize: Adaptation and Change in Party Organizations in Western Democracies*, chap. Ireland: Centralization, Prosessionalization and Campaign Pressures. Sage, London. (cited on page 156)

52. Farrell, D. and McAllister, I., 2000. *Elections in Australia, Ireland and Malta Under the Single Transferable Vote*, chap. Through a Glass Darkly: Understanding the World of STV. University of Michigan Press, Ann Arbor. (cited on page 155)

53. Farrell, D. M., 2011. *Electoral systems : a comparative introduction*. Palgrave Macmillan, Houndmills, Basingstoke, Hampshire, UK ; New York, 2nd edn. (cited on pages 1, 23, 112, 141, 147, 155, and 156)

54. Farrell, D. M. and McAllister, I., 2005. *The Australian Electoral System: Origins, Variations and Consequences*. University of New South Wales, Sydney. (cited on pages 1, 22, 23, 118, 147, and 155)

55. Flanagan, C.; Leino, K. R. M.; Lillibridge, M.; Nelson, G.; Saxe, J. B.; and Stata, R., 2013. PLDI 2002: Extended static checking for java. *SIGPLAN Notices*, 48, 4S (2013), 22–33. doi:10.1145/2502508.2502520. https://doi.org/10.1145/2502508.2502520. (cited on page 147)

56. Ford, B., 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 111–122. doi:10.1145/964001.964011. https://doi.org/10.1145/964001.964011. (cited on page 16)

57. Gallagher, M. Disproportionality in a proportional system: The irish experience. *Political Studies*, 23, 50 . (cited on page 155)

58. Ghale, M. K.; Goré, R.; and Pattinson, D., 2017. A formally verified single transferable voting scheme with fractional values. In *Electronic Voting - Second International Joint Conference, E-Vote-ID 2017, Bregenz, Austria*, 163–182. (cited on pages 141, 149, and 150)

59. Ghale, M. K.; Goré, R.; Pattinson, D.; and Tiwari, M., 2018. Modular formalisation and verification of STV algorithms. In *Electronic Voting - Third International Joint Conference, E-Vote-ID 2018, Bregenz, Austria, October 2-5, 2018, Proceedings*, 51–66. doi:10.1007/978-3-030-00419-4_4. https://doi.org/10.1007/978-3-030-00419-4_4. (cited on pages 149 and 150)

60. Ghale, M. K.; Pattinson, D.; Kumar, R.; and Norrish, M., 2018. Verified certificate checking for counting votes. In *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, 69–87. doi:10.1007/978-3-030-03592-1_5. https://doi.org/10.1007/978-3-030-03592-1_5. (cited on pages 141 and 142)

61. Ghale, M. K.; Pattinson, D.; and Norrish, M., 2019. Modular synthesis of verified verifiers for computation with stv algorithms. In *Forthcoming in FormaliSE: 7th International Conference on Formal Methods in Software Engineering - 10th International Conference, Montreal, Canada, May 27, 2019*. https://www.formalise.org/program. (cited on pages 141 and 142)

62. Girard, J., 1987. Linear logic. *Theor. Comput. Sci.*, 50 (1987), 1–102. doi:10.1016/0304-3975(87)90045-4. https://doi.org/10.1016/0304-3975(87)90045-4. (cited on page 141)

63. Goldreich, O., 2001. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press. ISBN 0-521-79172-3. (cited on pages 138 and 152)

64. Goré, R. and Lebedeva, E., 2016. Simulating STV hand-counting by computers considered harmful: A.C.T. In *Electronic Voting - First International Joint Conference, E-Vote-ID 2016, Bregenz, Austria, October 18-21, 2016, Proceedings*, 144–163. doi:10.1007/978-3-319-52240-1_9. https://doi.org/10.1007/978-3-319-52240-1_9. (cited on page 75)

65. Greenaway, D., 2014. *Automated proof-producing abstraction of c code*. Ph.D. thesis. University of New South Wales. (cited on page 149)

66. Greenaway, D.; Andronick, J.; and Klein, G., 2012. Bridging the gap: Automatic verified abstraction of C. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA*, 99–115. (cited on page 140)

67. Guéneau, A.; Myreen, M. O.; Kumar, R.; and Norrish, M., 2017. Verified characteristic formulae for CakeML. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden*, 584–610. (cited on page 126)

68. Haftmann, F., 2009. *Code generation from specifications in higher-order logic*. Ph.D. thesis, Technical University Munich. http://mediatum2.ub.tum.de/node?id=886023. (cited on page 150)

69. Halderman, J. A. and Teague, V., 2015. The new south wales ivote system: Security failures and verification flaws in a live online election. In *E-Voting and Identity - 5th International Conference, VoteID 2015, Bern, Switzerland, September 2-4, 2015, Proceedings*, 35–53. doi:10.1007/978-3-319-22270-7_3. https://doi.org/10.1007/978-3-319-22270-7_3. (cited on pages 151 and 153)

70. Heather, J., 2007. Implementing STV securely in pret a voter. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, 157–169. doi:10.1109/CSF.2007.22. https://doi.org/10.1109/CSF.2007.22. (cited on pages 151 and 152)

71. Hill, I. D. Tie breaking in stv. *Voting Matters*, 12, 5–6. www.votingmatters.org.uk. (cited on page 159)

72. Hill, I. D.; Wichmann, B. A.; and Woodal, D., 1987. Algorithm 123: Single transferable vote by meek's method. *The Computer J.*, 30 (1987), 277–281. (cited on page 163)

73. Hindley, J. R. and Seldin, J. P., 1986. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press. (cited on page 14)

74. Hinman, P. G., 2005. *Fundamentals of Mathematical Logic*. Talyor and Francis. (cited on page 47)

75. Jackson, D., 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11, 2 (2002), 256–290. doi:10.1145/505145.505149. https://doi.org/10.1145/505145.505149. (cited on page 147)

76. Jakobsson, M.; Juels, A.; and Rivest, R. L., 2002. Making mix nets robust for electronic voting by randomized partial checking. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, 339–353. http://www.usenix.org/publications/library/proceedings/sec02/jakobsson.html. (cited on page 152)

77. Jonker, H.; Mauw, S.; and Pang, J., 2013. Privacy and verifiability in voting systems: Methods, developments and trends. *Computer Science Review*, 10 (2013), 1–30. doi:10.1016/j.cosrev.2013.08.002. https://doi.org/10.1016/j.cosrev.2013.08.002. (cited on page 152)

78. Juels, A.; Catalano, D.; and Jakobsson, M., 2010. Coercion-resistant electronic elections. In *Towards Trustworthy Elections, New Directions in Electronic Voting*, 37–63. (cited on page 153)

79. Katz, R. S., 1980. *A Theory of Parties and Electoral Systems*. John Hopkins University Press, Baltimore. (cited on page 156)

80. Kelley, J. and McAllster, I., 1984. Ballot paper cues and the vote in australia and britain: Alphabetic voting, sex, and title. *Public Opinion Quarterly*, 48 (1984), 52–66. (cited on page 156)

81. Kiniry, J. R.; Cochran, D.; and Tierney, P. E., 2007. Verification-centric realization of electronic vote counting. In *2007 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'07, Boston, MA, USA, August 6, 2007*. https://www.usenix.org/conference/evt-07/verification-centric-realization-electronic-vote-counting. (cited on page 147)

82. Kiniry, J. R.; Morkan, A. E.; Cochran, D.; Fairmichael, F.; Chalin, P.; Oostdijk, M.; and Hubbers, E., 2006. The KOA remote voting system: A summary of work to date. In *Trustworthy Global Computing, Second Symposium,*

TGC 2006, Lucca, Italy, November 7-9, 2006, Revised Selected Papers, 244–262. doi:10.1007/978-3-540-75336-0_16. https://doi.org/10.1007/978-3-540-75336-0_16. (cited on page 147)

83. Kiniry, J. R.; Morkan, A. E.; Cochran, D.; Oostdijk, M.; and Hubbers, E., 2006. Formal techniques in a remote voting system. *ACM SIGSOFT Software Engineering Notes*, 31, 6 (2006), 1–2. doi:10.1145/1218776.1218793. https://doi.org/10.1145/1218776.1218793. (cited on page 147)

84. Klein, G.; Andronick, J.; Elphinstone, K.; Heiser, G.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2010. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53, 6 (2010), 107–115. doi:10.1145/1743546.1743574. https://doi.org/10.1145/1743546.1743574. (cited on page 140)

85. Klein, G.; Andronick, J.; Elphinstone, K.; Murray, T. C.; Sewell, T.; Kolanski, R.; and Heiser, G., 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32, 1 (2014), 2:1–2:70. doi:10.1145/2560537. https://doi.org/10.1145/2560537. (cited on page 133)

86. Kratsch, D.; McConnell, R. M.; Mehlhorn, K.; and Spinrad, J. P., 2006. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM J. Comput.*, 36, 2 (2006), 326–353. doi:10.1137/S0097539703437855. https://doi.org/10.1137/S0097539703437855. (cited on page 139)

87. Küçük, K. A.; Paverd, A.; Martin, A. C.; Asokan, N.; Simpson, A.; and Ankele, R., 2016. Exploring the use of intel SGX for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*, 5:1–5:6. doi:10.1145/3007788.3007793. https://doi.org/10.1145/3007788.3007793. (cited on page 167)

88. Kumar, R.; Mullen, E.; Tatlock, Z.; and Myreen, M. O., 2018. Software verification with itps should use binary code extraction to reduce the TCB - (short paper). In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, 362–369. doi:10.1007/978-3-319-94821-8_21. https://doi.org/10.1007/978-3-319-94821-8_21. (cited on page 133)

89. Leavens, G. T.; Schmitt, P. H.; and Yi, J., 2013. The java modeling language (JML) (NII shonan meeting 2013-3). *NII Shonan Meet. Rep.*, 2013 (2013). http://shonan.nii.ac.jp/shonan/wp-content/uploads/2011/09/No.2013-3.pdf. (cited on page 147)

90. Letouzey, P., 2004. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00150912. (cited on page 149)

91. Letouzey, P., 2008. Extraction in coq: An overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, 359–369. doi:10.1007/978-3-540-69407-6_39. https://doi.org/10.1007/978-3-540-69407-6_39. (cited on pages 8, 84, and 149)

92. Lijphart, A., 1986. *Electoral Laws and Their Political Consequences*, chap. Degrees of Proportionality of Proportional Representation Formulas. Agathon Press, New York. (cited on page 155)

93. Lijphart, A., 1994. *Electoral Systems and Party Systems: A Study of Twenty Seven Democracies, 1945-1990*. Oxford University Press, Oxford. (cited on page 155)

94. Lundell, J. Random tie breaking in stv. *Voting Matters*, 22, 1–6. www.votingmatters.org.uk. (cited on page 159)

95. Lundie, R., 2013. The disputed 2013 wa senate election. *Parliament of Australia*, (11 2013). Available online at https://www.aph.gov.au/About_Parliament/Parliamentary_Departments/Parliamentary_Library/FlagPost/2013/November/The_disputed_2013_WA_Senate_election. (cited on pages 1 and 2)

96. Maher, P., 2007. Explication defended. *Studia Logica*, 86, 2 (2007), 331–341. (cited on pages 19 and 134)

97. Martin-Lof, P., 1972. An intuitionistic theory of types. In *Twenty-Five Years of Constructive Type Theory* (Eds. G. Sambine and J. Smith). Oxford University Press. (cited on page 14)

98. Martin-Lof, P., 1996. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1, 1 (1996), 11–60. (cited on page 14)

99. McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P., 2011. Certifying algorithms. *Computer Science Review*, 5, 2 (2011), 119–161. (cited on page 139)

100. McIver, A.; Rabehaja, T. M.; Wen, R.; and Morgan, C., 2017. Privacy in elections: How small is "small"? *J. Inf. Sec. Appl.*, 36 (2017), 112–126. doi:10.1016/j.jisa.2017.08.003. https://doi.org/10.1016/j.jisa.2017.08.003. (cited on page 153)

101. Mehlhorn, K. and Näher, S., 1989. LEDA: A library of efficient data types and algorithms. In *Mathematical Foundations of Computer Science 1989, MFCS'89, Porabka-Kozubnik, Poland, August 28 - September 1, 1989, Proceedings*, 88–106. doi:10.1007/3-540-51486-4_58. https://doi.org/10.1007/3-540-51486-4_58. (cited on page 139)

102. Mehlhorn, K. and Näher, S., 1998. From algorithms to working programs on the use of program checking in LEDA. In *Fundamentals - Foundations of Computer Science, IFIP World Computer Congress 1998, August 31 - September 4, 1998, Vienna/Austria and Budapest/Hungary*, 81–88. (cited on page 139)

103. Mehlhorn, K.; Näher, S.; Seel, M.; Seidel, R.; Schilz, T.; Schirra, S.; and Uhrig, C., 1999. Checking geometric programs or verification of geometric structures. *Comput. Geom.*, 12, 1-2 (1999), 85–103. doi:10.1016/S0925-7721(98)00036-4. https://doi.org/10.1016/S0925-7721(98)00036-4. (cited on page 139)

104. Murray, T. C.; Matichuk, D.; Brassil, M.; Gammie, P.; and Klein, G., 2012. Noninterference for operating system kernels. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, 126–142. doi:10.1007/978-3-642-35308-6_12. https://doi.org/10.1007/978-3-642-35308-6_12. (cited on page 133)

105. Myreen, M. O. and Owens, S., 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24, 2-3 (may 2014), 284–315. doi:10.1017/S0956796813000282. (cited on pages 125 and 150)

106. Naish, L., 2013. Partial disclosure of votes in stv elections. *Voting Matters*, 30 (2013), 9–13. www.votingmatters.org.uk. (cited on pages 11, 152, and 153)

107. Necula, G. C., 1997. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, 106–119. doi:10.1145/263699.263712. https://doi.org/10.1145/263699.263712. (cited on page 138)

108. Nipkow, T.; Paulson, L. C.; and Wenzel, M., 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-43376-7. doi:10.1007/3-540-45949-9. https://doi.org/10.1007/3-540-45949-9. (cited on page 139)

109. Norrish, M., 1998. *C formalised in HOL*. Ph.D. thesis. University of Cambridge. (cited on page 140)

110. Noschinski, L.; Rizkallah, C.; and Mehlhorn, K., 2014. Verification of certifying computations through AutoCorres and Simpl. In *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA*, 46–61. (cited on pages 140, 148, 150, and 151)

111. Nurmi, H., 1997. It's not just the lack of monotonicity. *Representation*, 34 (1997), 48–52. (cited on page 156)

112. Ockenden, W., 2015. Nsw election 2015: ivote flaw 'allowed vote to be changed'; electoral commission fixes vulnerability. *ABC News*, (3 2015). Avaliable online at http://www.abc.net.au/news/2015-03-23/ivote-security-hack-allowed-change-of-vote-security-expert-says/6340168. (cited on page 4)

113. Parker, A. J., 1983. Localism and bailiwicks: The galway west constituency in the 1977 general election. In *Proceedings of the Royal Irish Academy 83:C2*, 17–37. (cited on page 156)

114. Pattinson, D. and Schürmann, C., 2015. Vote counting as mathematical proof. In *AI 2015: Advances in Artificial Intelligence - 28th Australasian Joint Conference*, 464–475. (cited on pages 141, 143, 149, and 150)

115. Pattinson, D. and Tiwari, M., 2017. Schulze vote as evidence carrying computation. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, 410–426. doi:10.1007/978-3-319-66107-0_26. https://doi.org/10.1007/978-3-319-66107-0_26. (cited on pages 141, 144, 150, and 168)

116. Paulin-Mohring, C., 2011. Introduction to the coq proof-assistant for practical software verification. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, 45–95. doi:10.1007/978-3-642-35746-6_3. https://doi.org/10.1007/978-3-642-35746-6_3. (cited on page 7)

117. Pierce, B. C., 2002. *Types and programming languages*. MIT Press. (cited on page 14)

118. Prabhakaran, M. and Sahai, A. (Eds.), 2013. *Secure Multi-Party Computation*, vol. 10 of *Cryptology and Information Security Series*. IOS Press. ISBN 978-1-61499-168-7. (cited on page 167)

119. Pressman, R. S., 2005. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, New York, 6th international edition edn. (cited on pages 7, 41, and 42)

120. Rae, D., 1967. *The Political Consequences of Electoral Laws*. Yale University Press, New York. (cited on page 155)

121. Rinard, M., 1990. Credible compilers. In *Technical Report MIT/LCS/TR-776*. (cited on page 140)

122. Robson, C. and Walsh, B., 1974. The importance of positional voting bias in the irish general election of 1973. *Political Studies*, 22 (1974), 191–203. (cited on page 156)

123. Ryan, P. Y. A., 2005. A variant of the chaum voter-verifiable scheme. In *Proceeding WITS '05 Proceedings of the 2005 workshop on Issues in the theory of security*, 81–88. (cited on pages 151 and 152)

124. Ryan, P. Y. A.; Bismark, D.; Heather, J.; Schneider, S.; and Xia, Z., 2009. Prêt à voter: a voter-verifiable voting system. *IEEE Trans. Information Forensics and Security*, 4, 4 (2009), 662–673. doi:10.1109/TIFS.2009.2033233. https://doi.org/10.1109/TIFS.2009.2033233. (cited on pages 11, 151, and 152)

125. Ryan, P. Y. A. and Schneider, S. A., 2006. Prêt à voter with re-encryption mixes. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, 313–326. doi:10.1007/11863908_20. https://doi.org/10.1007/11863908_20. (cited on pages 151 and 152)

126. Ryan, P. Y. A.; Schneider, S. A.; and Teague, V., 2015. End-to-end verifiability in voting systems, from theory to practice. *IEEE Security & Privacy*, 13, 3 (2015), 59–62. doi:10.1109/MSP.2015.54. https://doi.org/10.1109/MSP.2015.54. (cited on pages 2 and 151)

127. Schack-Nielsen, A. and Schürmann, C., 2008. Celf - A logical framework for deductive and concurrent systems (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, 320–326. doi:10.1007/978-3-540-71070-7_28. https://doi.org/10.1007/978-3-540-71070-7_28. (cited on page 141)

128. Schirmer, N., 2006. *Verification of sequential imperative programs in Isabelle/HOL*. Ph.D. thesis. Technische Universitat Munchen. (cited on page 140)

129. Schulze, M., 2011. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Coice and Welfare*, 36, 2 (2011), 267–303. (cited on page 144)

130. Schürmann, C., 2009. Electronic elections: Trust through engineering. In *First International Workshop on Requirements Engineering for e-Voting Systems, RE-VOTE 2009, Atlanta, Georgia, USA, August 31, 2009*, 38–46. doi:10.1109/RE-VOTE.2009.4. https://doi.org/10.1109/RE-VOTE.2009.4. (cited on page 141)

131. Slind, K. and Norrish, M., 2008. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, vol. 5170 of *LNCS*. Springer. (cited on page 9)

132. Stump, A., 2013. *Programming Language Foundation*. Jon Wiley. (cited on page 8)

133. Sullivan, G. F. and Masson, G. M., 1990. Using certification trails to achieve software fault tolerance. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing, FTCS 1990, Newcastle Upon Tyne, UK, 26-28 June, 1990*, 423–431. doi:10.1109/FTCS.1990.89397. https://doi.org/10.1109/FTCS.1990.89397. (cited on page 139)

134. Sullivan, G. F. and Masson, G. M., 1991. Certitication trails for data structures. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing, Montreal, Canada*, 240–247. doi:10.1109/FTCS.1991.146668. https://doi.org/10.1109/FTCS.1991.146668. (cited on page 139)

135. Taagepera, R. and Shugart, M., 1989. *Seats and Votes: The Effects and Determinants of Electoral Systems*. Yale University Press, New Haven. (cited on page 155)

136. Tan, Y. K.; Myreen, M. O.; Kumar, R.; Fox, A. C. J.; Owens, S.; and Norrish, M., 2016. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, 60–73. doi:10.1145/2951913.2951924. http://doi.acm.org/10.1145/2951913.2951924. (cited on pages 9 and 128)

137. Thiemann, R. and Sternagel, C., 2009. Certification of termination proofs using ceta. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, 452–468. doi:10.1007/978-3-642-03359-9_31. https://doi.org/10.1007/978-3-642-03359-9_31. (cited on page 142)

138. Troelstra, A. S. and Schwichtenber, H., 2000. *Basic Proof Theory*. Cambridge University Press, UK, 2nd edn. (cited on page 141)

139. Verity, F. and Pattinson, D., 2017. Formally verified invariants of vote counting schemes. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2017, Geelong, Australia, January 31 - February 3, 2017*, 31:1–31:10. doi:10.1145/3014812.3014845. http://doi.acm.org/10.1145/3014812.3014845. (cited on pages 141, 149, and 150)

140. Wadler, P., 2000. Proofs are programs: 19th century logic and 21st century computing. doi:10.1007/978-3-030-00419-4_4. https://homepages.inf.ed.ac.uk/wadler/papers/frege/frege.pdf. (cited on page 15)

141. Walsh, T., 2009. Manipulability of single transferable vot. In *Proceedings of the CARE'09 Interna-tional Workshop on Collaborative Agents—Research and Development*. (cited on page 156)

142. Wen, R., 2010. *Online Elections in Terra Australis*. Ph.D. thesis, University of New South Wales, Sydney, Australia. http://handle.unsw.edu.au/1959.4/51698. (cited on pages 2, 154, and 159)

143. Wen, R. and Buckland, R., 2009. Masked ballot voting for receipt-free online elections. In *E-Voting and Identity, Second International Conference, VOTE-ID 2009, Luxembourg, September 7-8, 2009. Proceedings*, 18–36. doi:10.1007/978-3-642-04135-8_2. https://doi.org/10.1007/978-3-642-04135-8_2. (cited on pages 153 and 154)

144. Wen, R.; McIver, A.; and Morgan, C., 2014. Towards a formal analysis of information leakage for signature attacks in preferential elections. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, 595–610. doi:10.1007/978-3-319-06410-9_40. https://doi.org/10.1007/978-3-319-06410-9_40. (cited on page 153)

145. Wichmann, B. Tie breaking in stv. *Voting Matters*, 19, 1–5. www.votingmatters.org.uk. (cited on page 159)

146. Wikipedia-contributors. Schulze method. https://en.wikipedia.org/w/index.php?title=Schulze_method&oldid=886102228. (cited on page 144)