Modular Formalisation and Verification of STV Algorithms

No Institute Given

Abstract. We introduce a formal, modular framework that captures a large number of different instances of the Single Transferable Vote (STV) counting scheme in a uniform way. The framework requires that each instance defines the precise mechanism of counting and transferring ballots, electing and eliminating candidates. From formal proofs of basic sanity condition for each mechanism inside the Coq theorem prover, we then synthesise code that implements the given scheme in a provably correct way and produces a universally verifiable certificate of the count. We have applied this to various variations of STV, including several used in Australian parliamentary elections and demonstrate the feasibility of our approach by means of real-world case studies.

1 Introduction

Single Transferable Vote (STV) is a family of vote counting schemes where voters express their preferences for competing candidates by ranking them on a ballot paper. STV is used, directly or indirectly, in many countries including Ireland, Malta, India, Nepal, New Zealand and Australia, but for example also to elect moderators in the StackExchange discussion forum [**?**] or the board of trustees of the John Muir trust [**?**].

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

1. Count all first preferences on ballot papers.
2. Elect all candidates whose first preferences are over the quota.
3. 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, the least preferred candidate is eliminated, and their votes are transferred to the next preference.

While the scheme appears simple and perspicuous, the above 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 election scheme, only the *last parcel* of an

elected candidate (the ballots attributed to the candidate at the last count) of an elected candidate is transferred. In contrast, the variant of STV used to elect the upper house in the Australian state of Vivtoria transfer *all* ballots (at a reduced transfer value). Similar differences also exist for the transfer of votes once a candidate is being eliminated.

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 protocols state that scrutiny shall cease once all vacancies are filled. In this paper, we abstract the commonalities of all different flavours of STV into a set of minimal requirements that we (consequently) call *minimal* STV. It comprises:

- the data (structure) that captures all states of the count
- the requirements that count, elect, eliminate, transfer (and three others) have to 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 uncounted ballots left to be dealt with currently, the tally of each candidate, the pile of ballots which had been counted in their favour, and four lists which specify the list of elected candidates whose votes await being transferred, the list of eliminated candidates whose votes is to be transferred, and the list of elected and continuing candidates. Basically, they record the state of computation up to the current stage.

We realise transitions between states, which correspond to acts of counting, eliminating, transferring, electing, and declaring winners committed by tally officers, as formal rules that relate a pre-state and a post-state. These rules are what varies between different flavours of STV. As a consequence, minimal STV does not define the rules, but rather postulates minimal *conditions* that every rule needs to satisfy. An *instance* of STV is then given by

1. the *definition* of the rules for counting, electing, eliminating, and transfer
2. formal *proofs* that the rules satisfy the respective conditions

We sometimes refer to the conditions that the various rules have to satisfy, somewhat informally, as *sanity checks*. They are the formal counterpart of the legislation that inform counting officers which action to perform, and when. The sanity check for each rule has two parts: the *applicability condition* specifies under what conditions the rule can be applied (for example, the count rule can only be applied if there are uncounted ballots). The *progress condition* specifies the effect of the rule on the state of the count (e.g. all ballots are counted). We establish three main properties about this generic version of STV. The first one asserts that each application of any of the generic transition of STV which satisfies constraints of sanity checks, reduces a complexity measure. The second

property ensures us that at any non-final state of the counting, at least one of the generic transitions is applicable provided that they all satisfy the sanity check requirements. Finally, we prove a termination theorem for this minimal STV algorithm.

All this is carried out inside the Coq theorem prover [**?**]. Using Coq's extraction mechanism [**?**] we can then automatically synthesise a (provably correct) program for STV counting from the termination proof. By construction, the executables are certifying programmes which produce an visualised trace of computation upon each execution. The certificate is checkable *by general member of the public* for correctness independent of the means employed for producing it. That is to say, we provably implement the count-as-cast aspect [**?**] of universal verifiability. Finally, our experimental tests with real elections demonstrate feasibility of our approach for real world applications. Compared with other formalisations of STV, where even small changes in the details of a single rule requires adapting a global correctness proof, the outstanding feature of the work reported here is *modularity*, that is, sanity checks are local to each rule) and *abstraction*, i.e. a general correctness proof based on the local conditions for each rule. It is precisely this simplicity that allows us to capture a large number of variations of STV, including several used in Australian parliamentary elections.

2 The Generic STV Machine

We begin by describing the components of minimal STV before discussing their implementation in the Coq theorem prover, together with examples.

2.1 The Machine States and Transitions

The best way to think of minimal STV and its instances is in terms of an abstract machine. The states can be thought of as snapshots of the hand counting procedure, where there is e.g. a current tally, and a set of uncounted ballots at every stage. Tallying is then formalised as transition between these states.

There are three types of machine sates: *initial*, *intermediate*, and *final*. An initial state specifies the list of all *formal* ballots. Final states of the machine are accepting stages where winners of an election are announced. Each intermediate state consists of six components:

1. A set of uncounted ballots, which must be counted
2. A tally function computing the amount of vote for each candidate
3. A pile function computing which ballots are assigned to which candidate
4. A list of already elected candidate whose votes awaits being transferred
5. A list of the eliminated candidates whose votes should be deal with
6. A list of elected candidates
7. A list of continuing candidates

Our mathematical formalisation uses the following terminology. We write $\mathcal{C}$ for the set of candidates participating in an election. Individual candidates are denoted by c, c', and c''. The set of ballots $\mathcal{B}$, is a shorthand for $(\mathsf{List}\ \mathcal{C})\times\mathbb{Q}$, where

$\mathbb{Q}$ is the set of rational numbers. Therefore a ballot ba is a pair (l, q) where $l \in \mathsf{List}(\mathcal{C})$ and $q \in \mathbb{Q}$. The characters h and nh are reserved for lists of continuing candidates ("hopefuls"), e and ne for lists of elected candidates, and bl, nbl for backlogs. A backlog bl is a pair $(l1, l2)$, where $l1$ contains the list of elected candidate whose votes should be transferred, and l is list containing the eliminated candidates whose votes are to be transferred.The quota of election and number of seats are symbolised by qu and st, respectively. Finally, tallies are denoted by t, nt and piles by p, np. The prefix "n" in the above stands for "new", e.g. ne denotes the elected candidates in the post state, after an action has been applied.

Suppose $ba \in \mathcal{B}$, and $bl, h, e, w \in \mathsf{List}(\mathcal{C})$ are given. Also assume t is a function from $\mathcal{C}$ into $\mathbb{Q}$, and p is a function from $\mathcal{C}$ into $\mathsf{List}(\mathcal{B})$. We write $\mathsf{initial}(ba)$ for the initial state, denote an intermediate state by $\mathsf{intermediate}(ba, t, p, bl, e, h)$, and a final one by $\mathsf{final}(w)$. Having established terminology and necessary representations, we can mathematically define the states of the generic STV machine.

Definition 1 (machine states). *Suppose ba is the initial list of ballots cast to be counted, and l is the list of all of candidates competing in the election. Then the set $\mathcal{S}$ of states of the generic STV is the union of all possible intermediate and final states that can be constructed from on ba and l, together with the initial state $\mathsf{initial}(ba)$.*

We now describe the mechanisms to progress an STV count such as electing all candidates that have reached the quota. These steps, formalised as *rules*, are the essence of each particular instances of STV, and are the one cornerstone of our generic notion of STV (the other being the properties that rules need to satisfy). We stipulate that each instance of STV needs to implement the following mechanisms that we formulate as rules relating a pre-state and a post-state:

start. to determine the *formal* votes and valid initial states.
count. for counting the uncounted ballots,
elect. to elect one or more candidates who have reached or exceeded the quota,
transfer-elected. for transferring surplus votes of already the elected,
transfer-removed. to transfer the votes of the eliminated candidate.
eliminate. to eliminate the weakest candidate from the process, and
elected win. to finish the counting by announcing the already elected candidates as winners.
hopeful win. to finish the counting by declaring the list of elected and continuing candidates as winners.

For the moment, we treat the above as transition labels only, and provide semantical meaning in the next section.

Definition 2 (machine transitions). *The set $\mathcal{T}$ consisting of the labels **count**, **elect**, **transfer-elected**, **transfer-removed**, **eliminate**, **hopeful win**, and **elected win**, is the set of transition labels of the generic STV.*

2.2 The Small-step Semantics

The textual description of STV is usually in terms of clauses that specify what actions are to be undertaken, under what conditions. In our formulation, this corresponds pre and postconditions for the individual counting rules. The precondition is an *applicability constraint*: it specifies under what conditions a particular rule is applicable. The postcondition is a *reducibility constraint*: it specifies how applying a rule progresses the count. Taken together, they form the *sanity check* for an individual rule. Technically, the applicability constraints ensure that they count never gets stuck, i.e. there is always one applicable rule, and reducibility guarantees termination.

Reducibility. A careful examination of STV protocols shows that each rule reduces the size of at least one of the following four objects: the list of continuing candidates, the number of ballots in the pile of the most recent eliminated candidate, the backlog, or the list of uncounted ballots. Using lexicographic ordering, this allows us to define a a complexity measure on the set of machine states in such a way that each rule application reduces this measure.

Local Rule Applicability. Ever incarnation of STV needs to, and indeed does, impose restrictions on when rules can and must be applied. These depend on the particular protocol, however some of them are uniform: for example, all STV algorithms require three properties to hold in order for elimination rule to apply: there must be empty seats to fill, there must not be any surplus votes awaiting transfer, and no candidate should have reached or exceeded the quota. We constrain each of the counting rules in this way to guarantee that at least one rule can always be applied.

To formulate the sanity checks for each transition step, we first define a lexicographic ordering on the set $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and impose it on non-final states of the generic machine.

Definition 3. *Let* $\{s : \mathcal{S} \mid s \text{ not final}\}$ *be the set of non-final machine states. We define a function* $\mathsf{Measure} : \mathcal{S} \to \mathbb{N}^5$ *as follows. We let* $\mathit{Measure}\ (\mathit{initial}\ ba) = (1,0,0,0,0)$*. Suppose* $bl = (l_1, l_2)$*, for some lists* l_1 *and* l_2*, and for a given candidate* c*,* $\mathsf{flat}\ (p\ c) = l_c$*. Then*

$$\mathit{Measure}\ (\mathit{state}\ (ba, t, p, bl, e, h)) = (0, \mathsf{length}\ h, \textstyle\sum_{d \in l_2} \mathsf{length}\ l_c, \mathsf{length}\ l_1, \mathsf{length}\ ba).$$

Note that the first component of the codomain of the measure function just serves to reduce measure from the initial state to any intermediate state. In the following, we describe the sanity checks for transfer and elect in detail, and leave it to the reader to follow those for other rules on the basis of the formal Coq development.

Transfer-elected check. A rule that decribes the tansfer of surplus of an elected candidate (labelled *transfer-elect*) needs to satisfy two conditions. The applicability constraint asserts that transfer-elect is applicable to any intermediate machine state input of the form $\mathsf{state}([],t,p,bl,e,h)$ where there are no uncounted ballots left, if there are vacancies to fill, i.e. $\mathsf{length}(e) < st$, there are surpluses awaiting transfer, i.e. $bl \neq []$, 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 reducibility condition requires that any application of *transfer-elected* reduces the length of the backlog *bl* while elected and continuing candidates remain unchanged. Mathematically, this takes the following form:

Definition 4 (transfer-elected sanity check). *A rule $R \subseteq \mathcal{S} \times \mathcal{S}$ satisfy the* transfer-elected sanity check *if and only if the following hold:*

applicability *for any state* $\mathsf{input} = \mathsf{state}([], t, p, bl, e, h)$ *that satisfies (*$\mathsf{length}\ e$*)* $< st$ *(there are still seats to fill),* $bl \neq []$ *(there are votes to be transferred) and* $\forall c.\ (c \in h \rightarrow (t\ c < qu))$ *(no candidate has reached the quota), there exists a post-state* output *such that* input R output*.*

reducibility *for any machine states* input *and* output*, if* input$R$output *then* input *is of the form* input = state*(*[]*,t,p,bl,e,h), *output *is of the form* output = state*(nba,t,np,nbl,e,h) and* length*(nbl) <* length*(bl) (i.e. the backlog is reduced).*

The following is immediate from the definition of measure (Definition **??**):

Theorem 1. *Suppose R is a transition label which satisfies the second condition stated in definition 4. Also assume* input*,* output $\in \mathcal{S}$ *and* input R output*. Then the complexity of* output *is less than the* input*, i.e.* Measure(input) *is lexicographically smaller than* Measure(output)

Elect check. The action of electing a candidate, also formalised as a rule in our framework, is subject to the following constraints: in the pre-state input = state([],*t*,*p*,*bl*,*e*,*h*), there must be a continuing candidate that has reached the quota, and there must be an unfilled vacancy. If this is the case, there must exist a post-state output where the set of continuing candidates is smaller, there are still no uncounted ballots, and the piles and backlog for candidates may be updated. Mathematically, this takes the following form:

Definition 5 (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* $\mathsf{input} = \mathsf{state}([], t, p, bl, e, h)$ *and any continuing candidate* $c \in h$*, if* $t(c) \geq qu$ *(candidate c has reached the quota) and* $\mathsf{length}(e) < st$ *(there are still vacancies), there exists a post-state* output *such that* input$R$output*.*

reducibility *for any states* input *and* output*, if* input R output*, then* input *is of the form* $\mathsf{input} = \mathsf{state}([], t, p, bl, e, h)$*,* output *is of the form* $\mathsf{output} = \mathsf{state}([], nt, np, nbl, ne, nh)$ *and both* $\mathsf{length}(nh) < \mathsf{length}(h)$*,* $\mathsf{length}(ne) > \mathsf{length}(e)$*.*

Analogous to Theorem 1 we have the following:

Theorem 2. *Assume a transition rule R meets the second condition of definition 5. Then any application of the transition R reduces the complexity measure.*

Similarly we define sanity checks corresponding to other transition labels, namely start, count, eliminate, hopeful win, and elected win. Additionally, for other sanity checks, we establish theorems such as Theorems 1 and 2. Then by drawing on them, we obtain a corollary on the measure reduction for the generic STV machine.

Corollary 1. *Any transition R corresponding to a machine transition in $\mathcal{T}$ that satisfies the corresponding sanity check reduces the complexity measure.*

2.3 The Generic STV Machine

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 correspoinding sanity check, can therefore be seen as a small-step semantics for STV counting. We capture this mathematically as a generic machine.

Definition 6 (The generic STV machine). *Let $\mathcal{S}$ be the set of STV states (Definition 1) and $\mathcal{T}$ be the set of transition labels (Definition 2). The* generic STV machine *is $M = \langle \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ where S_t is the santity check condition associated with transition $t \in \mathcal{T}$. An* instance *of the generic STV machine 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.*

In the sequel, we show that each instance of the generic STV machine in fact produces an election result, present a formalisation, and several concrete instances.

2.4 Progress via Applicability Condition

One specific "sanity check", in fact the sanity check that did inspire the very term, is the ability to in fact always be able to progress the count. That is, the rules are such that one (more) rule is applicable at every stage of the count, and there are no "dead ends". We have seen that each rule comes with specific applicability conditions, e.g. no rule other than count may apply if there are uncounted ballots, or that all elected candidates shall be declared winners if the number of candidates marked elected equals the number of seats to be filled.

Here, this is captured by the various applicability conditions that are part of the sanity checks outlined above. The key insight is that if the sanity check 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
- if all vacancies are filled by elected canidates, then scrutiny shall cease
- if elected and continuing candidates fill all vacancies, scrutiny shall cease
- uncounted ballots shall be counted
- candidates that exceed the quota shall be elected
- the surplus of elected candidates shall be transferred
- ballots of eliminated candidates shall be transferred

– the weakest candidate shall be eliminated

We realise this order of rule application in the proof of the rule applicability theorem. We draw upon the local rule applicability property present in the sanity checks, which is satisfied by the generic STV model, to guide the theorem prover Coq to finalise 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 3 (Rule Applicability). *Let $I = \langle \mathcal{T}, (R_t)_{t \in \mathcal{T}} \rangle$ be an instance of the generic STV machine. If* input *is a non-final state of the STV model of computation, then there is a transition label t in $\mathcal{T}$ such that we can move from* input *to a new state* output *through R_t, i.e.* input R_t output.

Corollary 1 asserts that whenever a transition $R \in \mathcal{T}$ applies, it reduces the complexity measure. Theorem 3 establishes the fact that for any non-final machine state, indeed a transition from the set $\mathcal{T}$ is applicable. Therefore we accomplish a termination property for the generic STV model. Expressed in the programming semantics terminology, it 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 4 (Termination). *Each execution of every instance of the generic STV machine on any initial state* input *terminates at a final state* output*, along with constructing the sequence of computations taken from* input *to reach to* output*.*

3 Formalisation of The Generic Machine in Coq

We have formalised each notion introduced in the previous section in the theorem prover Coq. Our formalisation consists of a base layer, with instances defined in separate modules. The base layer contains the generic inductive types, definitions of sanity checks, parametric transition labels, specification of the STV machine, functions which are used to formulate the generic STV machine, and theorems proved about the generic STV model. It also includes functions which are commonly called by the modules to carry computation for instances of STV. Instances consist of four parts:

1. instantiations of the generic counting conditions defined in the base, with concrete instances of counting rules of a particular STV schemes
2. proofs which establish sanity checks for the instantiated transition rules
3. possibly auxilary fuctions specific to the particular instance of STV
4. an instantiation of the termination theorem which allows us to synthesise a provably correct, and certifiable, vote counting implementation

```
Inductive STV_States :=
   | initial: list ballot -> STV_States
   | state:   list ballot
          * list (cand -> Q)
          * (cand -> list (list ballot))
          * (list cand) * (list cand)
          * {elected: list cand | length elected <= st}
          * {hopeful: list cand | NoDup hopeful} -> STV_States
   | winners: list cand -> STV_States.
```

Fig. 1. inductive definition of STV machine states

Here we briefly discuss the base of the framework and the reasons for some of our design decisions. In the next section, we demonstrate modular formalisation of three different STV algorithms.

We encode machine states as an inductive type (figure 1) with three constructors; `initial`, `state`, and `final`. The constructor `state` has six value fields which parametrise the list of uncounted ballots, a list of tallies, pile function, and lists of backlog, elected and continuing candidates, respectively.

Tie breaking. To accommodate formalisation of some tie breaking methods used in some STV schemes, we encode tallies as a list of tallies so that we can keep track of previous amount of votes which each candidate had received. This allows us to realise one popular tie breaking decision procedure. In this method, whenever two or more candidates are tied together at the weakest amount of vote, 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 had received the least amount of votes compared to others with whom he is tied currently. Then we update the current state of the counting by eliminating this candidate.

Last parcel. 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 so that they are elected. As a result, we choose to formalise the pile function to assign a list containing some other lists of ballots. Each of these lists of ballots, 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.

Parameters. We formalise the notions of candidates, the quota, and transition labels parametrically. The parameters are later specified in the modules for each particular STV. For example, each transition label is generically specified to be a function of type `STV_States -> STV_States -> Prop`.

Sanity checks. Corresponding to each generic transition label, there is a formal definition of the sanity checking. Sanity checks are constraints which are expected of every instance of STV to successfully pass in order to be classified as an STV scheme. Here we illustrate the encoding of the sanity checks for the elect transition. Items (1) and (2) in the Figure 2 respectively match with the first and the second items given in Definition 5. Note that the check loosens the constraint so that in order for elect rule to apply, we need an electable continuing candidate and electing them would not exceed the number of vacancies. This allows us to define a concrete elect transition for e.g. CADE STV [**?**] which elects only one candidate who has reached or exceeded the quota, rather than electing all of the electable candidates together. Moreover, we are able to formalise other instances of elect transitions which do elect all of the eligible candidates in one step.

Generic STV record. We bundle the generic quota, transition labels and the evidences that the generic transitions satisfy the sanity checks, in one record type named `STV_record`. For example, one record field of `STV_record` is the requirement that the generic elect transition meets the constraints of the elect sanity check, which technically means `(Elect_sanity_check (elect))` $\in$ `STV_record`.

Finally, we formally prove all of the mathematical properties discussed under the previous section for any `stv` of type `STV_record`. In particular, we demonstrate the termination property. The termination theorem is instantiated in separate modules with particular `STV_record` values, such as ACT STV and CADE STV, to obtain termination property for them as well and carry provably correct computations upon programme extraction into Haskell.

4 Modular Formalisation of Some STV Machines

We already have discussed some points where STV schemes diverge from one another. They mainly vary in their specification of formal votes, quota, what is the surplus of an elected candidate, how many candidates to elect out of all of

```
Definition Elect_Sanity_Check (R:STV_States -> STV_States-> Prop)
:=
1. (∀ input t p bl e h, input = state([],t,p,bl,e,h) ->
   ∃ (c: cand),
    length (proj1_sig e) +1 ≤ st
    ∧ In c (proj1_sig h) ∧ (quota ≤ (hd nty t) c) ->
       ∃ output, R input output) ∧
2. (∀ input output, R input output -> ∃ t p np bl nbl e ne h nh,
   input = state([],t,p,bl,e,h)
   ∧ length(proj1_sig e) < length(proj1_sig ne)
   ∧ length(proj1_sig nh) < length(proj1_sig h)
   ∧ output = state([],t,np,nbl,ne,nh))
```

Fig. 2. Sanity check for elect transition

those who are electable, how to update the transfer value of votes of an elected candidate, how to transfer the surpluses, or how to eliminate a candidate and then distribute their votes among other continuing candidates.

We have formalised some STV schemes used in real election, out of which we describe two in detail.

4.1 Victoria STV

The Victoria state of Australia employs a version of STV [**?**] for electing the upper house representatives. Figure 3 depicts the instantiation of the generic elect transition label with our formulation of the Victoria STV elect rule. Each line of `Victoria_Elect` embodies some clauses of Victoria state's counting protocol which specify the elect rule. We only explain lines 5, 6, and 7 of Figure 3 to show how they accommodate some of the protocol's clauses.

The counting protocol of Victoria STV, defines surpuls votes to be "*the number, if any, of votes in excess of the quota of each elected candidate*". Moreover it dictates, under Section 17, Subsection 7 Clause (a), that "*the number of surplus votes of the elected candidate is to be divided by the number of first preference votes received by the elected candidate and the resulting fraction is the transfer value*". In lines 6 and 7, we compute the surplus vote and the fractional transfer value accordingly and multiply it at the current value of every ballot in the pile of the elected candidate c to update the pile of this candidate.

The protocol further states under subsection (8), and (13) that "*Any continuing candidate who has received a number of votes equal to or greater than the quota on the completion of any transfer under subsection (7), or on the completion of a transfer of votes of an excluded candidate under subsection (12) or (16), is to be declared elected*". The definition requires electing candidate(s) no matter how they have obtained enough votes. We therefore implement clauses (8) and (13) in Line 5, where we elect everyone over or equal to the quota, place them in the update list of elected candidates `ne`, and insist that the list of elected candidates in this state, namely `l` and the old list of elected candidates `e` together form a permutation of `ne`. The choice for permutation carries further details which realise other parts of the protocol, which we leave to the reader.

```
Definition Victoria_Elect input output : Prop :=
∃ t p np bl nbl nh h e ne,
1. input = state([],t,p,bl,e,h) ∧
2. ∃ l, length (proj1_sig e) + length(l) ≤ st
3. ∧ ∀ c, In c l ->(In c (proj1_sig h) ∧(quota ≤ hd nty t (c)))
4. ∧ ordered (hd nty t) l∧ Permutation l(proj1_sig nh)(proj1_sig h)
5. ∧ Permutation l(proj1_sig e)(proj1_sig ne)∧ (nbl= bl ++ l)
6. ∧ ∀ c, In c l -> (np (c) = map(map (fun b ⇒
7. (fst b, (snd b)× ((hd nty t (c))-quota)/((hd [] t)c))(p c)
8. ∧ output = state([],t,np,nbl,ne,nh)
```

Fig. 3. Victoria STV elect transition

Next, we describe how the updated pile of an elected candidate in `Victoria_Elect` is transferred by Victoria's transfer-elect transition. Figure 4 illustrates the instantiation of the generic transfer-elected rule with a concrete case used by Victoria STV. Notice that in the first conjunct of Line 4 in Figure 3, we order the list of elected candidates according to the tally amount. When it comes to transferring elected surplus, as we see in Line 4 of Figure 4, the biggest surplus is dealt with first which belongs to candidate c. Furthermore, Line 5 specifies that *all of this candidate's surplus is distributed* at the fractional value decided in `Victoria_Elect`.

```
Definition Victoria_TransferElected input output :=
∃ nba t p np bl nbl h e,
1. input = state([],t,p,bl,e,h) ∧
2. length(proj1_sig e) < st ∧ output = state([],t,np,nbl,ne,nh)
3. ∧ ∀ c, In c (proj1_sig h) -> ((hd nty t) c < quota)
4. ∧ ∃ l c, (bl= (c::l,[]) ∧ (nbl= (l,[])) ∧ (np(c) = [])
5. ∧ (nba= flat(fun x => x)(p c) ∧ (∀ d, d≠c -> (np c)=(p d))
```

Fig. 4. Victoria STV transfer-elected transition

4.2 Australian Capital Territory STV

Government of the Australian Capital Territory uses a version of STV [**?**] for election of the lower house. This STV stands out for some of its characteristics, including transfer of the last parcel of votes and the formulation of transfer value. The specification of the elect transition of ACT STV is similar to the one in Figure 4 except for lines 6 and 7, which are replaced by the following.

$$\texttt{np(c)= map(fun b =>}\ \frac{\texttt{(fst b, (snd b)} \times \texttt{((hd nty t(c))- quota)}}{\texttt{(Sum snd(last(p c))}}\texttt{) (last(p c))}$$

Moreover, the ACT version of transfer-elected is as in Figure 4 except that the fist conjunct in Line 5 is substituted by a different proposition and reads `nba = last (p c)`. The two variations together tell us that we only transfer the last parcel of the elected candidate and the transfer value equals to the surplus votes of this candidate divided by the sum of fractional values of this last parcel, rather than the tally of the elected candidate.

There are obvious possible issues with the transfer value formula used in the ACT STV. It may come to a situation where the fractional value of a surplus vote is bigger than 1, which is essentially a flaw of the algorithm. As a result, the software used by authorities of ACT which implements the algorithm [**?**], makes modifications to ensure no surplus votes become more than 1. We adapt this corrected version in our implementation. Nonetheless, nothing would restrict us from selecting the defective original formula of ACT STV, if we chose to.

5 Certifying Extracted Programmes and Experiments

We use the built-in mechanisms of Coq to extract executable programme for each module into the Haskell language. The automatic extraction method provides a sufficient degree of verification that the executable behaves in accordance to its Coq formalisation. Correctness proofs established in the Coq therefore extend to the executables. However, upon each execution of the programs, we generate a run-time certificate which functions as an independently checkable evidence for reliability of the computation carried out.

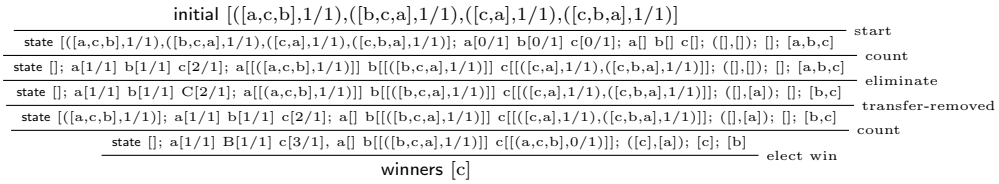

Fig. 5. example of a certificate

Theorem 4 guarantees that for each run of the program a formal certificate exists. Moreover, the theorem elaborates that the formal certificate is the sequence of computation performed in the execution to obtain the final result. To produce a concrete certificate, upon each execution of extracted Haskell programmes, we visualise this formal sequence of machine states and transitions which we have encountered from the input to reach to the end result.

The certificate generated for each input proves correctness of the count to the general public, without need to trust in the skilfulness of the programmer, their well grounded understanding of legal texts, the extraction means used, and any hardware malfunctioning. Checkability of the visualised certificate by *any scrutineers* witnesses that our tallying technique satisfies the count-as-recorded subproperty of the universal verifiability quality. Consequently, any election protocol designed for STV schemes which requires proof of tallying correctness can utilize our tool.

Figure 5 illustrates an example of a concrete certificate, where candidates a, b, and c are competing for one seat. We discuss certification only briefly as it is analogous to [**?**]. We use exact fractions for computations to avoids issues explained in literature [**?**] about rounding numbers. Every line shows six component each of which corresponds to an abstract data representation of the intermediate states of the abstract machine: the list of uncounted ballots, the tallies of candidates, each candidate's pile, the backlog and the list of elected and continuing candidates.

We have evaluated the efficiency of our approach by testing the extracted module for the lower house ACT STV on some real elections held in 2006 and 2008 (Figure 6). The Molonglo electorate of ACT is the biggest lower house electorate in Australia, both in terms of the number of candidates who are elected

and the number of voters in the district. The extracted programme computes the result in just 22 minutes.

6 A Technical Discussion

We have introduced a framework for formalisation, verification, and provably correct computation with various STV algorithms. Our software is the outcome of macro level design decisions which we made prior to the implementation. In the architecture, we have been attentive to balancing out between different standards for designing a framework. The modular design allows for a much simpler implementation than made possibly by other frameworks (e.g. [**?**]) as we only need to discharge proofs at a per-rule basis which is also reflected in the fact that (a) we capture realistic voting protocols, and (b) we can accommodate a larger number of protocols with ease.

Previous work emphasises data structures and certification, and showcases this by means of monolithic specifications and proofs. Our work adds modularity, and we distil the algorithmic essence of STV into what we call *sanity checks*.

Every instance of STV which satisfies the sanity checks enjoys rule applicability and termination property established in Theorems 3 and 4. Therefore, for an instance of STV to be verified, we simply need to discharge the sanity checks rather than duplicating the whole proof process again. These checks offer an abstraction on the algorithmic side which helps us avoid duplication of encoding. Moreover, unlike the previous work, users do not need to know how the application theorem and termination have been proven in order to show termination of the particular instance into which they are interested. Additionally, separation of into modules further improves usability. Everyone after a correct computation of their preferred flavouer of ST, can simply use the framework and instantiate as appropriate.

The ability to just instantiate is significant, as a large number of aspects are dealt with once and for in the base layer that comprises roughly 25000 lines of code. Each module already formalised is less than 500 lines. Therefore, an interested user has to just carry out formalisation and discharging sanity checks in about 500 lines to acquire a verified executable implementation of their favourite STV. On the other hand, accomplishing the same goal by using

electoral	ballots	vacancies	candidates	time (sec)	certificate size (MB)	year
Brindabella	63334	5	19	116	80.6	2008
Ginninderra	60049	5	27	332	128.9	2008
Molonglo	88266	7	40	1395	336.1	2008
Brindabella	63562	5	20	205	94.3	2012
Ginninderra	66076	5	28	289	126.1	2012
Molonglo	91534	7	27	664	208.4	2012

Fig. 6. ACT Legislative Assembly 2008 and 2012

the previous platform, demands 25000 lines of encoding, along with overcoming numerous technicalities.

Related work. DeYoung and Schurmann [**?**] use Linear Logic [**?**] to formally specify a STV scheme and then discharge proofs inside the logical framework of Celf [**?**]. Technical knowledge of linear logic is required to understand how the textual description of the protocol matches with the formal one. On the other hand, Pattinson and Shurmann [**?**], and Verity and Pattinson [**?**] formalise a simple version of STV and First-past-The-Post elections in Coq. Their approach reduces the gap between the informal protocol and the encoded counting rules. Also they prove some properties such existence of winners in every formal execution. Then they extract certifying executable in Haskell which can compute large size elections. Pattinson and Tiwari [**?**] tackle verification of Schultz method by the similar approach of specifying the algorithm in Coq, discharging proofs and extracting executables in Haskell and OCaml. Their extracted executable outputs performs effectively and certificates output in each run of the programme offer a unique feature. Besides evidence supporting why the winner has been elected, it also visualises non-existence of any path between each defeated candidate and the winner. Dawson et al [**?**] formalise a version of Hare-Clark in the theorem prover HOL [**?**]. Their specification is expressed in the Higher-Order-Logic. Moreover, they encode computational definition inside HOL and prove its correctness against the HOL specification of the protocol. Then to actually compute, they manually transliterate the computational definitions from HOL into the syntax of Standard ML functional language. A significant issue with this approach is existence of no guarantee that the transliterate encoding would behave according to the original HOL definitions, simply because of semantic differences of HOL and SML. Therefore, their verification does not legitimately extend beyond HOL to the SML programme.

7 Conclusion

We have designed a framework for modular formalisation, verification of, and provably correct computation with STV algorithms. Our work is fully formalised and provides an encoding and provably correct executables for various flavours of STV.