Modular Formalisation and Verification of STV Algorithms

Milad K. Ghale(✉), Rajeev Goré, Dirk Pattinson, and Mukesh Tiwari

The Australian National University, Canberra, Australia
milad.ketabghale@anu.edu.au

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 conditions 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 demonstrated 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 in many countries including Ireland, Malta, India, Nepal, New Zealand and Australia. It is also used to elect moderators in the StackExchange discussion forum [19] and the board of trustees of the John Muir trust [11].

To count an election according to STV, one usually computes a quota dependent on the number of ballots cast (often the Droop quota [7]) 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.

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 STV 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 STV variant used in the upper house of 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.

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. 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 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, 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.

We establish three main properties of this generic version of STV. The first is that each application of any of the generic transitions of STV reduces a complexity measure. The second is that at any non-final state of the count, at least one of the generic transitions is applicable because it satisfies its sanity check requirements. The third is that the overall minimal STV algorithm terminates.

All this is carried out inside the Coq theorem prover [3]. Using Coq's extraction mechanism [14] we can then automatically synthesise a (provably correct) program for STV counting from the termination proof. By construction, the executables are certifying programs which produce an visualised trace of computation upon each execution. The correctness of the certificate can be checked by anyone with minimal technical knowledge, independent of the way it was obtained. That is to say, we provably implement the counted-as-cast aspect [4] 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 features of our work is *modularity*, since sanity checks are local to each rule, and *abstraction*, since the general correctness proof is 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 states: *initial*, *intermediate*, and *final*. An initial state contains the list of all *formal* ballots. Final states are where winners are declared. Each intermediate state consists of seven components:

1. A set of uncounted ballots, which must be counted;
2. A tally function computing the number of votes for each candidate;
3. A pile function computing 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.

We use $\mathcal{C}$ for the set of all candidates participating in an election and use c, c', and c'' for individual candidates from $\mathcal{C}$. We use $\mathsf{List}(\mathcal{C})$ for the set of all lists over $\mathcal{C}$ and use $\mathbb{Q}$ for the set of rational numbers. A ballot is an ordered pair (l, q) where $l \in \mathsf{List}(\mathcal{C})$ is the preference order and $q \in \mathbb{Q}$ is the (possibly fractional) value of this ballot. We write $\mathcal{B} = \mathsf{List}(\mathcal{C}) \times \mathbb{Q}$ for the set of all ballots, i.e. preference ordered lists of candidates, together with a transfer value. We use h and nh for lists of continuing ("$\underline{\text{h}}$opeful") candidates, and e and ne for lists of $\underline{\text{e}}$lected candidates. A $\underline{\text{b}}$ack$\underline{\text{l}}$og is a pair $(l1, l2)$, the lists of elected and eliminated candidates, respectively, both of whose votes await transfer. We use bl, nbl for backlogs, use qu for the quota for being elected and use st for the number of vacant seats. We use t, nt for tallies and use p, np for piles. The prefix "n" always stands for "new", thus ne is the list of elected candidates in the post state, after an action has been applied.

Suppose that $ba \in \mathsf{List}(\mathcal{B})$, and $bl, h, e, w \in \mathsf{List}(\mathcal{C})$ are given. 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 use $\mathsf{initial}(ba)$ for the initial state, use $\mathsf{intermediate}(ba, t, p, bl, e, h)$ for an intermediate state, and use $\mathsf{final}(w)$ for a final one. 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 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 is the union of all possible intermediate and final states that can be constructed from ba and l, together with the initial state initial(ba).*

We now describe the mechanisms to progress an STV count, such as electing all candidates that have reached quota. These steps, formalised as *rules*, are the essence of each particular instance of STV, and are one of the two cornerstones of our generic notion of STV: the other being the properties that rules must 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* ballots 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 elected candidates;
transfer-removed: to transfer the votes of the eliminated candidate;
eliminate: to eliminate the weakest candidate from the process; and
elected win: to terminate counting by declaring the already elected candidates as winners;
hopeful win: to terminate 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 to pre- and post-conditions for the individual counting rules. The pre-condition is an *applicability constraint*: it specifies under what conditions a particular rule is applicable. The post-condition 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, applicability constraints ensure that the count never gets stuck i.e. there is always one applicable rule, while the reducibility constraint 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 recently eliminated candidate; the backlog; and the list of uncounted ballots. Using lexicographic ordering, this allows us to define a complexity measure on the set of machine states in such a way that each rule application reduces this measure.

Local Rule Applicability. Each instance of STV must, and indeed does, impose restrictions on when rules can, and must, be applied. Most depend on the particular instance, but some are universal. For example, all STV algorithms require three constraints for the elimination rule to apply: there must be vacant seats; there must be no 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, we define a lexicographic ordering on the set $\mathbb{N}^5$ 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* Measure : $\mathcal{S} \rightarrow \mathbb{N}^5$ *as follows. We let Measure (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, flat (p c) =* l_c *where for a list l of lists,* flat *l is the concatenation (flattening) of all elements of l. Then*

$$\mathit{Measure}\ (\mathit{state}\ (ba, t, p, bl, e, h)) = (0,\ \mathit{length}\ h, \sum\nolimits_{d \in l_2}\ \mathit{length}\ l_d,\ \mathit{length}\ l_1,\ \mathit{length}\ ba).$$

Note that the first component of the co-domain of the measure function simply reduces measure from the initial state to any intermediate state, and the third component is the sum of the length of the ballots cast in favour of eliminated candidates that await transfer. In the following, we describe the sanity checks for **transfer** and **elect** in detail, and leave it to the reader to reconstruct those for the other rules from the formal Coq development.

Transfer-Elected Check. The *transfer-elected* rule that decribes 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 $\mathsf{state}([], t, p, bl, e, h)$, where the list of uncounted ballots is empty, if there are vacancies to fill ($\mathsf{length}(e) < st$), there are surpluses awaiting transfer, ($bl \neq []$), 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* $R \subseteq \mathcal{S} \times \mathcal{S}$ *satisfies 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* $\forall c.\ (c \in h \to (t\ c < qu))$ *(no continuing candidate has reached the quota), there exists a post-state* output *such that* $\mathsf{input}\ R\ \mathsf{output}$*.*

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

The following is immediate from the definition of measure (Definition 3):

Theorem 1. *If transition label R obeys the reducability condition of Definition 4,* $\mathsf{input} \in \mathcal{S}$*,* $\mathsf{output} \in \mathcal{S}$ *and* $\mathsf{input}\ R\ \mathsf{output}$*, then the* output *complexity* $\mathsf{Measure}(\mathsf{output})$ *is lexicographically smaller than the* input *complexity* $\mathsf{Measure}(\mathsf{input})$*.*

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

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

Analogous to Theorem 1 we have the following:

Theorem 2. *If a transition rule R meets the reducibility 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**. For all such sanity checks, we establish analogues of 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 corresponding 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 $\mathcal{S}$ and $\mathcal{T}$ be the sets of STV states (Definition 1) and transition labels (Definition 2), respectively. The* generic STV machine *is $M = \langle \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ where S_t is the santity 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.*

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 the Applicability Conditions

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. As an example, no rule other than count may apply if there are uncounted ballots (and indeed count must be applicable in this situation), or that all elected candidates shall be declared winners if the number of candidates marked elected equals the number of seats to be filled. 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 3 (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* input R_t output.

Corollary 1 shows that every applicable transition R_t, for $t \in \mathcal{T}$, reduces the complexity measure. Theorem 3 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 4 (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 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.

We now briefly discuss the framework base and explain some design decisions. In the next section, we give modular formalisations of three STV algorithms.

We encode machine states as an inductive type (Fig. 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, a pile function, and lists of backlogs, elected and continuing candidates, respectively.

Tie Breaking. To formalise some tie breaking methods used in some STV schemes, we encode tallies into a chronological list so we can trace the number of votes which each candidate received in previous rounds. This allows us to realise 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 less votes than the tied candidates. 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

```
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

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 of lists of ballots to every candidate: the element of this list are the ballots that have been counted in favour of the candidate at the successive counts. This allows us to identify precisely the set of ballots that 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 where last parcel is being used.

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 associated with a relation, that is, 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 Fig. 2 respectively match with the first and

```
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

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 [2] 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 evidence that the generic transitions satisfy the sanity checks in one record type named `STV_record`. For example, one 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 program extraction into Haskell.

4 Modular Formalisation of Some STV Systems

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 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 describe two of the real-world STV schemes we have formalised.

4.1 Victoria STV

The Australian state of Victoria employs a version of STV [20] for electing 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 cof the Victorian STV protocol which specify the elect rule. We only explain lines 5, 6, and 7 of Fig. 3.

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 by the current value of every ballot in the pile of the elected candidate c to update the pile of this candidate.

```
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

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 `ne` of elected candidates, and insist that the list `l` of elected candidates in this state and the old list `e` of elected candidates together form a permutation of `ne`. Insisting that the new (combined) lists of winners are a permutation of `l` and `e` combined also imply that no new candidates are introduced, no existing candidates are deleted, and there is no duplication.

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 Fig. 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 Fig. 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 computed 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

Lower house elections in the Australian Capital Territory (ACT) use a version of STV [1] which 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 Fig. 4 except for lines 6 and 7, which are replaced by the following:

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

Moreover, the ACT version of transfer-elected is as in Fig. 4 except that the fist conjunct in Line 5 is replaced 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 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 issues with the transfer value formula used in the ACT STV [10]. For example, it is possible for the calculated fractional value of a surplus vote to exceed 1, which is clearly a flaw of the algorithm. As a result, the software used by the ACT election commission which implements the algorithm [18], makes explicit modifications to ensure no surplus votes exceeds 1. We adapt this corrected version in our formalisation. Nonetheless, nothing would restrict us from selecting the defective original formula of ACT STV, if we chose to.

5 Certifying Extracted Programs and Experiments

We use the built-in mechanisms of Coq to extract executable Haskell programs for each module. The automatic extraction method provides very high assurance that the executable behaves in accordance to its Coq formalisation. Correctness proofs established in the Coq therefore give functional correctness of the executables. However, each execution of the extracted program generates a run-time certificate, providing independently checkable evidence of the underlying computation.

Theorem 4 guarantees each run of the program produces a *formal* certificate, i.e. a sequence of states of the count that are linked by rules, as an element of an inductive data type. (This contrasts with what one may call an *concrete* certificate which would be a file that comprises a textual representation of the formal certificate.) Moreover, the theorem guarantees that the formal certificate is the sequence of computation performed in the execution to obtain the final result. To produce a concrete certificate from an execution of extracted Haskell program, we need to agree on textual representations for the elements of the data types concerned.

The certificate generated for each input witnesses the correctness of the count. Note that it is trivial to demonstrate that the existence of a correct certificate implies the correctness of the result, as the latter is defined precisely as being obtained through a sequence of correct rule applications. Certificate correctness

```
initial [([a,c,b],1/1),([b,c,a],1/1),([c,a],1/1),([c,b,a],1/1)]
------------------------------------------------------------------------------------------------------- start
state [([a,c,b],1/1),([b,c,a],1/1),([c,a],1/1),([c,b,a],1/1)]; a[0/1] b[0/1] c[0/1]; a[] b[] c[]; ([],[]); []; [a,b,c]
------------------------------------------------------------------------------------------------------- count
state []; a[1/1] b[1/1] c[2/1]; a[[([a,c,b],1/1)]] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[]); []; [a,b,c]
------------------------------------------------------------------------------------------------------- eliminate
state []; a[1/1] b[1/1] C[2/1]; a[[(a,c,b],1/1)]] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[a]); []; [b,c]
------------------------------------------------------------------------------------------------------- transfer-removed
state [([a,c,b],1/1)]; a[1/1] b[1/1] c[2/1]; a[] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[a]); []; [b,c]
------------------------------------------------------------------------------------------------------- count
state []; a[1/1] B[1/1] c[3/1], a[] b[[([b,c,a],1/1)]] c[[(a,c,b],0/1)]]; ([c],[a]); [c]; [b]
------------------------------------------------------------------------------------------------------- elect win
winners [c]
```

Fig. 5. Example of a certificate

can be checked by anyone, without *any* trust in the means that were used in the production of the certificate, or the underlying hardware. The fact that concrete certificates can be checked by scrutineers means that our tallying technique satisfies the count-as-recorded property of universal verifiability. Thus any election protocol designed for STV schemes which requires a proof of tallying correctness can use 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 described elsewhere [8]. We use exact fractions for computations to avoid the rounding issues explained in [10]. Every line shows six components, each corresponding 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 lists 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 2008 and 2012 (Fig. 6). The Molonglo electorate of ACT is the biggest lower house electorate in Australia, both in the number of vacancies and the number of voters. The extracted program computes the result in just 22 min.

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

6 A Technical Discussion

We have introduced a framework for formalisation, verification, and provably correct computation with various STV algorithms. In the design decisions that we made, we have been balancing different aspects for designing a framework. The modular design allows for a much simpler realisation than made possibly by other frameworks (e.g. [8]) 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 satisfying the sanity checks enjoys the rule applicability and termination properties established in Theorems 3 and 4. Therefore, for an instance of STV to be verified, we simply need to establish that the sanity checks hold, rather than duplicate the whole proof process. These checks offer an abstraction on the algorithmic side which helps us avoid duplication of code. Unlike previous work, users do not need to know how the application and termination theorems have been proved in order to show termination of their particular instance. Additionally, separation into modules further improves usability. Anyone seeking a verified implementation of their preferred flavour of STV can simply use our framework and instantiate as appropriate.

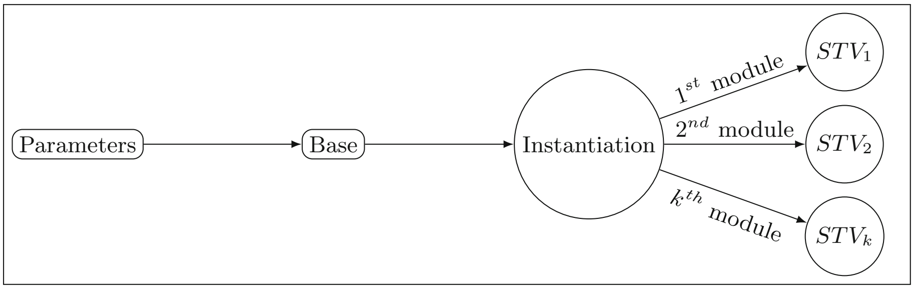

Fig. 7. System description

Figure 7 illustrates the framework architecture. The parameters component includes type level declaration of candidates, vacancies, and the quota. The base comprises the encoding of the generic STV model along with functions commonly called by the dependent modules. The instantiation component consists of instantiating types specified in the parameters file and automatically discharging of required proofs. Finally, the instantiated generic model is called into each module for discharging sanity checks and consequently extracting provably correct Hakell programs.

The ability to just instantiate is significant as many aspects are dealt with once and for all in the base layer of 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 the previous platform, demands 25000 lines of encoding, along with overcoming numerous technicalities.

Related Work. DeYoung and Schürmann [6] formally specify an STV scheme as a linear logic [9] program and then discharge the required correctness proofs inside the logical framework Celf [17]. Celf is capable of executing the specification but their linear logic program does not scale to real-world elections.

Dawson et al. encode an ML program for STV counting into the HOL4 theorem prover [5] and prove various correctness properties of the program, including termination. HOL4 is able to execute the ML encoding within reasonable time bounds for small elections, but not for large ones. But there is a gap between the HOL4 semantics of ML and those used by ML compilers. This gap could be closed by using the proof-producing synthesis [13] of CakeML code from the HOL assertions, then using the verified CakeML compiler [12] to produce the machine code. However, this has not been done to date.

Pattinson and Schürmann [15], and Verity and Pattinson [21] formalise a simple version of STV and first-past-the-post elections in Coq and prove properties such as termination and the existence of winners. Then they extract certifying executables in Haskell which can handle real-world elections. Their crucial contribution is that their executable code produces a certificate for every run, which can be idependently verified.

Pattinson and Tiwari [16] extend this method to tackle the Schultz method. Their extracted code handles real-world election and also outputs a certificate for every run. The certificate not only witnesses how the winner was elected, but also provides concrete evidence that each losing candidate is a “loser”.

7 Conclusion

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

References

1. ACT Electoral Commission: https://www.elections.act.gov.au/education/act_electoral_commission_fact_sheets/fact_sheets_-_general_html/elections_act_factsheet_hare-clark_electoral_system
2. Beckert, B., Goré, R., Schürmann, C.: Analysing vote counting algorithms via logic - and its application to the CADE election scheme. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 135–144. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_9
3. Bertot, Y., Castéran, P., Huet, G., Paulin-Mohring, C.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Cortier, V., Galindo, D., Küsters, R., Müller, J., Truderung, T.: Verifiability notions for e-voting protocols. IACR Cryptology ePrint Archive 2016, 287 (2016)
5. Dawson, J.E., Goré, R., Meumann, T.: Machine-checked reasoning about complex voting schemes using higher-order logic. In: Proceedings of EVote-ID 2015, pp. 142–158 (2015)

6. DeYoung, H., Schürmann, C.: Linear logical voting protocols. In: Kiayias, A., Lipmaa, H. (eds.) Vote-ID 2011. LNCS, vol. 7187, pp. 53–70. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32747-6_4
7. Droop, H.R.: On methods of electing representatives. J. Stat. Soc. Lond. **44**(2), 141–202 (1881)
8. Ghale, M.K., Goré, R., Pattinson, D.: A formally verified single transferable voting scheme with fractional values. In: Krimmer, R., Volkamer, M., Braun Binder, N., Kersting, N., Pereira, O., Schürmann, C. (eds.) E-Vote-ID 2017. LNCS, vol. 10615, pp. 163–182. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68687-5_10
9. Girard, J.: On the unity of logic. Ann. Pure Appl. Logic **59**(3), 201–217 (1993)
10. Goré, R., Lebedeva, E.: Simulating STV hand-counting by computers considered harmful: A.C.T. In: Krimmer, R., et al. (eds.) E-Vote-ID 2016. LNCS, vol. 10141, pp. 144–163. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52240-1_9
11. John Muir Trust: Apply to be a trustee. https://www.johnmuirtrust.org/assets/000/002/860/How_to_apply_to_be_a_Trustee_Jan_2018_original.pdf. Accessed 15 May 2018
12. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL). ACM, January 2014
13. Magnus, M.O., Scott, O.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014)
14. Letouzey, P.: A new extraction for CoQ. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
15. Pattinson, D., Schürmann, C.: Vote counting as mathematical proof. In: Pfahringer, B., Renz, J. (eds.) AI 2015. LNCS (LNAI), vol. 9457, pp. 464–475. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26350-2_41
16. Pattinson, D., Tiwari, M.: Schulze voting as evidence carrying computation. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 410–426. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_26
17. Schack-Nielsen, A., Schürmann, C.: Celf - a logical framework for deductive and concurrent systems (system description). In: Proceedings of IJCAR 2008, pp. 320–326 (2008)
18. Software Improvements: Electronic and voting and counting sytems. http://www.softimp.com.au/evacs/index.html. Accessed 12 May 2015
19. StackExchange: Moderator elections (2018). https://math.stackexchange.com/election/6?tab=election. Accessed 15 May 2018
20. The Parliament of Victoria: Electoral act 2002. http://www.legislation.vic.gov.au/domino/web_notes/ldms/pubstatbook.nsf/f932b66241ecf1b7ca256e92000e23be/3264bf1de203c08aca256e5b00213ffb/%24FILE/02-023a.pdf
21. Verity, F., Pattinson, D.: Formally verified invariants of vote counting schemes. In: Proceedings of ACSW 2017, pp. 31:1–31:10 (2017)