Modular Formalisation and Verification of STV Algorithms

Milad K. Ghale Dirk Pattinson

Research School of Computer Science, ANU, Canberra

Abstract. We introduce a framework which offers uniform formalisation and verification of some properties for various Single Transferable Voting (STV) algorithms. These algorithms, although different from one another in some ways, have key properties in common which make them STV instances. By abstracting away common features of particular STV schemes, we obtain minimum requirements to accept a given, arbitrary scheme as a legitimate STV instance. We formally prove that any STV scheme which meets the expectations of the above conditions satisfies some mathematical properties, such as termination. We demonstrate extensibility and concision of our framework by modular formalisation of a divergent range of STV algorithms in the theorem prover Coq. Then, by the built-in mechanisms of Coq, for each of the modules, we extract a certifying executable programme into the Haskell programming language. The certificate produced upon each execution, is a visualisation of the trace of the computation carried out to obtain an end result for an election instance. It provides us with an independently checkable proof of tallying correctness to establish count-as-recorded subproperty of the universal verifiability quality. Finally we show effectiveness of our approach by evaluating the executables on some real-size elections.

1 Introduction

Single Transferable Voting is a family of proportional representative vote counting schemes where voters express their preferences for competing candidates by ordering them in ballots. In many countries, such as Australia and Ireland, some variants of STV are adapted by government and private sectors as the counting scheme for electing parliament and senate members, board of trustees, and community representatives. There are noticeable differences among these versions of STV. However, most of them share a common underlying structure which pins down the central characteristics of STV algorithms.

When one analyses variants of STV protocols, they recognise some familiar pieces of information. They all have competing candidates, a quota of election, and vacancies. Moreover, they all specify the protocol in a language which assumes or implies that counting advances step by step in a discrete manner. The steps are taken by invariably appearing names such as *eliminate* or *transfer*. These names act as transtions from one step into another to update the current step, for example by removing someone from the counting. Furthermore, there are constraints described in these protocols that dictate when a tally officer is allowed to take which one of the transitions. Again, similarities strike us in that some of the conditions for applying these transitions constantly emerge. For example, all of STV protocols state that when we have come to fill all of the vacancies, the counting process must terminate by declaring winners of the election. These conditions operate in two levels. First they assign a particular function to each transition act. These expectations required by the protocols from each transition, serves as the meaning of performing that individual act. Second, they impose an algorithmic skeleton in how transitions collectively behave, which consistently appears in most of STV schemes.

We abstract the above data and algorithmic structure away to obtain a minimal STV. In particular, we formally understand each single discrete steps of counting mentioned above as a mathematical object which comprises some data. Based on the kind of data that such an object encapsulates, we codify them into three sets. The first set groups the initial states of the counting. They essentially hold the original ballots cast to be counted. The second set consists of intermediate stages of the process. Each intermediate state carries seven pieces of information; the list of uncounted ballots left to be dealt with currently, the tally amount 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 keep record of the computation up to the current stage. The last class of states are the final ones. They inform us of the end result of tallying.

We realise transitions, which correspond to acts of counting, eliminating, transferring, electing, and declaring winners committed by tally officers, as mathematical functions which take an input state of the computation and output another state. These functions are defined generically, which means they are void of content. Such formalisation enables us to cover as much STV protocols as possible. However, we bless them with some meaning *sanity checks*. In essence, sanity checks are the formal counterpart of the minimal, constantly appearing conditions explained above which instruct the tally officer which transition to apply and when.

The formally specified states of the computation, the transition labels and the applicability constraints imposed on them by sanity checks, constitute the minimal, generic, STV algorithm. 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. The content of this theorem, embodies the algorithmic skeleton explained above. It determines how the transitions of the generic STV, and instances of STV as well, collectively behave. Finally, we prove a termination theorem for this minimal STV algorithm.

All of the above steps, are formalised in the base of our framework. On the other hand, however, STV protocols have distinctive features that scape being pinned down by sanity checks, which are merely common properties among STV cases. We formally accommodate concrete instances of STV used in real elections in separate modules. This happens in four phases. First we instantiate each generic, fleshless transition with its legal specification according to the instance of the STV being formalised. More conditions for when and how a transition may apply are added in this step. Therefore, we come to honour individualistic side of each STV. Then we discharge proof obligations of sanity checks for each of the instantiated transitions. By satisfying sanity checks, we are able to obtain for each STV instance, the same computational and mathematical properties proved earlier for the generic version. Also we verify functions used for computation with the instantiated transition at this same step. Hence, we ascertain that we compute in accordance with the generic STV skeleton and that every instance of such computation proceeds correctly.

Finally, by the automatic programme extractions of Coq, we extract execute executables pertinent to each modules. The extraction mechanism provides us with a satisfactory level of verification that the executable behaves in agreement with its formalised counterpart in Coq. Moreover, 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. Therefore, we come to provide the public with tools for personally verifying the tallying count. Consequently, our approach respects and meets the count-as-cast subproperty of the universal verifiability quality. Finally, our experimental tests with real elections demonstrate feasibility of our approach for real world application so that our work extends beyond mere academic fruitfulness.

2 The Generic STV Machine

There are numerous elections which use STV for their tallying method. Despite the apparent differences observable in the particular version of STV which is invoked, there are fundamental data and algorithmic structure common among most of them. By abstracting these underlying features away, we obtain an abstract data structure and minimal constraints which form a generic STV. We prove some mathematical properties, such as termination, about the generic version.

In this section, we elaborate on what the components of the generic version are and how they function as a whole. Shortly afterwards, we demonstrate how the properties are established in the theorem prover Coq and then modularly illustrate how they extend to particular STV cases. For this purpose, we describe our generic STV design by employing a pedagogical language from theoretical computer science, namely automata theory and programming semantics.

2.1 The Machine States and Transitions

In tallying process of an election, there are some pieces of information which are necessary to know in order to handle the computation. Moreover, this kind of data invariably appears throughout the tallying process so that tally officers would have access to current state of the procedure. For example, in hand counting methods, officers must know what are the uncounted ballots and what is the current tally amount for each candidate. Since computations are local phenomena, tallying process is divided into stages of counting each of which comprises such vital data. These encapsulated pieces of information, form the sates of our generic STV machine.

There are three types of machine sates; *initial*, *intermediate*, and *final*. An initial state specifies the list of all *formal* ballots cast to be tallied. On the other hand, final states of the machine are accepting stages where winners of an election are announced. Last, 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

One could think of the pile and tally functions as abstraction of the action performed by tally officers when they respectively assign ballots and their values to candidates.

To express machine states in a mathematically enough precise language, we use symbols each of which stands for a notion introduced above. A set of candidates participating in an election is represented by $\mathcal{C}$ and members of this set are illustrated by c, c', and c''. The set of ballots $\mathcal{B}$, is a short hand 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, 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. The quota of election and number of seats are symbolised by qu and st, respectively. Finally, tallies are shown by t, nt and piles by p, np.

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 $\mathsf{List}(\mathcal{B})$. Then we illustrate an initial state of the machine by $\mathsf{initial}(ba)$, 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 equals to all of possible intermediate and final states formed based on ba and l, together with the initial state* initial*(ba).*

There is also a mechanism devised to advance the counting process by updating the current state of the count with necessary changes. For example, if the counting comes to a stage where some candidate has received enough votes to be elected, a particular rule for electing permits making the transition from this state into a new one where the candidate has been elected.

These steps which are an integral part of each STV and perform updating the information locally, are named counting rules. A specific set of counting rules consistently comes into sight when looking into instances of STV :
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.

Each of these counting rules accept a machine state as input and output another state. At the moment, we treat them merely as transition labels of the generic STV. However, in the next section, we specify a semantics for each and explain their computational content.

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

STV protocols are composed of clauses which primarily textually describe the expectations of the protocol from each counting rule. Protocol clauses informally specify when and to what kind of state a counting rule applies, and how it must update this state by making a transition to another. From the algorithmic perspective, STV protocols carry two common consistent properties. The first is some invariant requirements in order for a counting rule to be applicable, which we formalise in this section. The second involves invariant order of rule applications, which is discussed in the next section.

Various STV algorithms differ in details of what conditions must be met before a specific transition step can apply to a given state of the machine. For example, the lower house ACT STV transfers only *the last parcel* of an elected candidate. However, some other STV schemes such as the one used in the upper house Victoria state of Australia, transfer all of the surplus votes rather than merely the last parcel received.

On the other hand, there are conditions that appear invariably among different STV schemes. These conditions, each of which correspond to a transition label, comprise the small-step semantics for the generic STV machine. They are also used to check if a given arbitrary STV can legitimately be categorised as an STV instance. We obtain the conditions by singling out the key invariant properties existing in each STV scheme and name them *sanity checks.* The checks are formed by conjunction of the reducibility and local rule applicability properties.

Reducibility. A careful examination of STV protocols illustrates that each rule application at least reduces the length of one the following four objects: the list of continuing candidates, the length of pile of the most recent eliminated candidate, the backlog, or the list of uncounted ballots. The astute observer quickly grasps that by the correct choice of ordering among the above quantities, a complexity measure can be imposed on the set of machine states in such a way that each rule application reduces the measure. This measure persists to apply and function across various STV algorithms and to each transition label.

Local Rule Applicability. In order to legally correctly apply a counting rule, the protocol declares some restrictions to be met first. Many of the constraints depend on the particular protocol, however some of them consistently come to attention. For example, all of 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. Each one of the counting rules is constrained to their distinct conditions that constantly apply, regardless of the specifics of the STV protocol invoked.

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 $\{s : \mathcal{S} |\ s \text{ not final}\}$ be the set of non-final machine states. Then* Measure (initial *ba*) = (1,0,0,0,0). *Suppose for a given backlog bl, $(l1, l2)$ equals to bl, for some candidate lists $l1$ and $l2$. Then*

$$\text{Measure (state } (ba, t, p, bl, e, h)) = (0, \text{ length } h, \textstyle\sum_{c \in l2} \text{ length } (p\ c), \text{length } bl, \text{length } ba).$$

We detail the checks for transfer and elect rules and do not attend to the rest of them due to space limitations. Each sanity check is a conjunction of two properties which concern the reducibility and local applicability.

Transfer-elected check. For a given transition label R to be formally approved as a legal transfer-elected transition, satisfaction of two criteria is needed. The first one asserts that for any current intermediate machine state input of the form state($[]$,t,p,bl,e,h) where there are no uncounted ballots left, if there are vacancies to fill, i.e. length(e) $< st$, there are surpluses awaiting transfer, i.e. $bl \neq []$, and no continuing candidate has reached or exceeded the quota, then there must exist another intermediate machine state output which is reachable from input by one step through R, i.e. input R output. The second condition requires that :

- the input of R be an intermediate state where there is no ballot left to count, i.e. $ba = []$,
- the output to be an intermediate state as well,
- any application of R reduces the length of the backlog.
- the list of uncounted ballots, the pile and the backlog are updated. However, the lists of elected and continuing candidates remain the same as before.

Definition 4 (transfer-elected sanity check). *Assume R is given. It is accepted as a legal transfer-elected transition label if and only if the followings hold.*

- *for any machine state* input*, tally t, pile p, and lists of candidates bl, e, and h, if* input = state([]*,t,p,bl,e,h), and*
  1. *(length e) $< st$*
  2. *$bl \neq []$*
  3. *$\forall c.\ (c \in h \rightarrow (t\ c < qu))$*

  then exists a machine state output *such that* input *R* output*.*
- *for any machine states* input *and* output*, if we can move from* input *state to the* output *by R, then there exist a tally t, piles p and np, backlogs bl and nbl, a list of continuing candidates h, and a list of elected candidates e such that* input = state([]*,t,p,bl,e,h),* length(nbl) < length(bl)*, and* output = state(nba,t,np,nbl,e,h)*.*

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*.*

Elect check. A transition label R must meet two expectations to legitimately be an STV elect transition. First, for an arbitrary machine state input of the form state([],t,p,bl,e,h), if there exist a continuing candidate c which has reached or exceeded the quota and if adding c to the list of vacancies does not cause electing more than the number of empty seats, then there is a new machine state output which we can move to by R. Second, the input and output of R must comply with the following constraints :

- the input must an intermediate machine state where there is no uncounted ballot left
- length of the update list of continuing candidates nh is shorter than h
- the length of the updated list of elected candidates is shorter than e
- the output is another intermediate state where there are no ballots to count
- the pile function is updated to np, list of the backlog, continuing and elected candidates are updated to nbl, nh, and ne, respectively.

Definition 5 (elect sanity check). *A transition label R is a legal STV elect transition whenever the following conditions hold.*

- *for any bl, e, $h \in$ List($\mathcal{C}$), the tally t and pile p, and any* input $\in \mathcal{S}$*, if* input = state([]*,t,p,bl,e,h) and if there exists a continuing candidate c such that* length(e) $+1 < st$ *and $qu \leq (t\ c)$ then there is a machine state* output *where* input *R* output*.*

– *for any* input, $\mathsf{output} \in \mathcal{S}$, *if* input *R* output *then for some tally t, piles p and np, backlogs bl and nbl, candidate lists e, ne, h, and nh, it is the case that* $\mathsf{input} = \mathsf{state}([], t, p, bl, e, h)$, $\mathsf{length}(nh) < \mathsf{length}(h)$, $\mathsf{length}(e) < \mathsf{length}(ne)$, *and* $\mathsf{output} = \mathsf{state}([], t, np, nbl, ne, nh)$.

Theorem 2. *Assume a transition label 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 Theorem 2. Then by drawing on them, we obtain a corollary on the measure reduction for the generic STV machine.

Corollary 1. *Any transition label R which satisfies a check condition reduces the complexity measure.*

The sanity checks specify what is the computational content of an execution of a transition label on a given input. Therefore, the set of sanity checks $\mathcal{SC}$ operates as a small-step semantics for the generic STV machine.

Definition 6 (The generic STV machine). *Let* $\mathcal{S}$ *be the set specified in definition 1 and* $\mathcal{T}$ *be the set of transition labels given in definition 2. Then by the generic STV model of computation we mean the structure* $< \mathcal{S}, \mathcal{T} >$, *where each transition in* $\mathcal{T}$ *satisfies the pertinent sanity check in* $\mathcal{SC}$.

2.3 The Invariant Terminating Structure in Executions

There is one last ubiquitous property which numerous STV algorithms respect and appears in the way counting rules must collectively behave. Many STV algorithms specify a precedence of rule applications so that one knows which individual rule must be applied next. This order determines the overall structure of the computation. For example, STV protocols require that if there are uncounted ballots, no rule other than the count rule should apply. As another example, they assert that when all of the vacancies have been filled, only the rule elected win must apply. Definition 7 lays down this invariant structure existing in several STV algorithms.

Definition 7. *Suppose the current state of computation is* input. *Then the following pseudo algorithm specifies which transition should be used.*

```
1. Is input an initial state or an intermediate one?
  • if input is an initial state then apply the rule start
  • if input is an intermediate state(ba,t,p,bl,e,h), then
    2. Is all vacancies filled, i.e. length(e) = st ?
      • if yes then apply elected win and declare winners
      • if not then
        3. Is length (e)+ length(h) less than or equal to st?
          • if yes then apply hopeful win and declare winners
          • if not then,
4. Is there uncounted ballots, i.e. $ba \neq []$ ?
 • if yes then apply count transition
 • if not then,
   5. Has any candidate reached or exceeded the quota?
    • if yes then elect them by elect transition
    • if not then,
      6. Are there votes awaiting transfer, i.e. $\textsf{fst}\ bl \neq []$?
       • if yes then,
         ⋆ if $\textsf{snd}\ bl = []$, then apply transfer-elected
         ⋆ if $\textsf{snd}\ bl \neq []$ then apply transfer-removed
       • if not then, eliminate the weakest candidate.
```

We formally realise the content of definition 7 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). *Assume* input *is a current non-final state of the STV model of computation. Then there is always a transition label R in $\mathcal{T}$ such that we can move from* input *to a new state* output *through R, in accordance to the algorithm in definition 7.*

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 the generic STV machine on any initial* input *state 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. Out formalisation consists of a base for our framework and some separate modules which call the base to avoid duplication of encoding. The base 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

```
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

carry computation for instances of STV. The modules, on the other hand, mainly host encoding of particular STV schemes. They essentially include four parts :

1. instantiation of the generic counting transitions defined in the base, with concrete instances of counting rules of a particular STV schemes
2. discharging proofs which establish sanity checks for the instantiated transition rules
3. possibly defining particular functions for evaluating specifically with the STV instance defined in the module
4. instantiation of the termination theorem with the concrete transition rules, and proofs of the sanity checks for these transitions.

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 label. 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 later define a concrete elect transition for 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)`) ∈ `STV_record`.

```
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

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 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. For the sake of space limitation, we mention two of them here. First, for each of them we describe some key aspects which distinguish their way of electing and transferring elected votes from the other STV cases mentioned under this section.

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*

```
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

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*". Recall that theorem 3 respects the structure of definition 7. The definition requires electing the electable candidate(s) no matter how they have obtained enough votes. We therefore, meet the clauses (8) and (13) by 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, nonetheless we need to ignore mentioning them.

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 you 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 ACT 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. Logical specification of the elect transition of ACT STV matches with the one in figure 4 except for lines 6 and 7, which is replaced by the following.

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

Moreover, the ACT version of transfer-elected accords with figure 4 except that the fist conjunct in line 5 is substituted by a different proposition :

$$\texttt{nba=}\ \textsf{last}\ \ \texttt{(p c)}$$

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.

```
                         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

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 concrete certificate generated for each input instance, relieves the general public of laying 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 over one seat. We only shortly explain it, as we have detailed concrete certificates elsewhere[]. We use exact fractions for computations to avoids issues explained in literature[] about rounding numbers. Every intermediate line depicts six component each of which corresponds to an abstract data representation of the intermediate states of the abstract machine. They are separate by semi-colon symbol. The first one is the list of uncounted ballots. The Second one is tallies of candidates. The third part shows piles of each candidate. The last three are the backlog, list of elected, and list of continuing candidates.
Experiments. We have evaluated efficiency of our approach by testing the extracted module for the lower house ACT STV on some real elections held in 2006 and 2008. [**?**]

6 A Technical Discussion

Here we introduce 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. As a result, the final product surpasses the earlier attempt made in formalisation of STV algorithms[].

Our previous work emphasises excessively on the data structure of STV algorithms and pays little attention to the algorithmic dimension. In short, it first formalises a generic specification of STV scheme, where the algorithm is encoded in terms of types to be instantiated later. The highly generically realised STV is bereft of algorithmic substance to the point that we do not perceive any specificity of STV. Consequently, each time one aims at instantiating the generic types with parameters to actually formalise an instance of STV, they

electoral	ballots	vacancies	candidates	time (sec)	certificate size (MB)	year
Brindabella	63334	5	19		84.0	2008
Ginninderra	60049	5	27		124.8	2008
Molonglo	88266	7	40		324.0	2008
Brindabella	63562	5	20		95.8	2012
Ginninderra	66076	5	28		131.5	2012
Molonglo	91534	7	27		213.7	2012

Fig. 6. ACT Legislative Assembly 2008 and 2012

have go through countless duplication of data and algorithmic structure simply to establish an instantiated version of the generic application and termination theorems.

Here we have distanced from generic programming and given weight to the algorithmic character of STV. The essence of STV algorithms is encoded in sanity checks which operate on any instance of STV. Any STV which satisfies 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. Therefore, the 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. Thus we hide information which are not necessary for any user to be bothered with for using the software. Additionally, separation of encoding into modules besides promoting the information hiding, usability and abstraction further, makes our product extensible for future applications. Any user interested in correct computation with a preferred STV, would enjoy calling the base of the framework in and draw upon the existing functionalities to achieve the goal.

The above choice is worthwhile, as the base of the framework which is dealt with once and for all comprises roughly 25000 lines of encoding. Each module already formalised is less than 500 lines. Therefore, an interested user has to just carry out formalisation and discharging sanity checks for 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.

7 Conclusion

We have designed a framework for modular formalisation, verification of, and provably correct computation with STV algorithms. The product hosts formalised and verified encoding of key characteristics of STV schemes in a base.