A Formally Verified Single Transferable Vote Scheme with Fractional Values

Milad Ghale, Rajeev Goré, Dirk Pattinson
Australian National University

Rajeev.Gore@anu.edu.au

March 2, 2021

Overview

E2E Verifiabiity Needs Program Verification

Single Transferable Voting (STV) scheme ?

Why is it hard to tally ballots according to STV?

Current computer counting in Australia

Where is the scrutiny and trust ?

Interactive Synthesis of Vote Counting Programs

Results, Features, Further Work, Caveats and Conclusion

E2E Verifiability Needs Program Verification

Cast as intended: voters verify that electronic ballot is correct

Recorded as cast: ballot was not tampered with in transit

Tallied as recorded: voter can verify that ballot was tallied

E2E Verifiability Needs Program Verification

Cast as intended: voters verify that electronic ballot is correct

Recorded as cast: ballot was not tampered with in transit

Tallied as recorded: voter can verify that ballot was tallied

But ... what if the vote-counting program contains bugs?

E2E Verifiability Needs Program Verification

Cast as intended: voters verify that electronic ballot is correct

Recorded as cast: ballot was not tampered with in transit

Tallied as recorded: voter can verify that ballot was tallied

But ... what if the vote-counting program contains bugs?

Software independence:

Idea 1: vote-counting programs must produce a tallying script

Idea 2: if the tallying script is correct then the result is correct

Idea 3: it is trivial to write a program to check tallying script

E2E Verifiability Needs Program Verification

Cast as intended: voters verify that electronic ballot is correct

Recorded as cast: ballot was not tampered with in transit

Tallied as recorded: voter can verify that ballot was tallied

But ... what if the vote-counting program contains bugs?

Software independence:

Idea 1: vote-counting programs must produce a tallying script

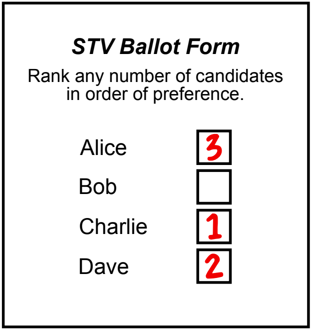
Idea 2: if the tallying script is correct then the result is correct

Idea 3: it is trivial to write a program to check tallying script

That is: provide easily-checkable evidence that this run is correct

What do we mean by voting scheme?

A method for setting out, filling in and counting ballots



Setting out: order of candidates fixed or Robson rotated ?

Filling in: write all numbers from 1 to N or only ones you want ?

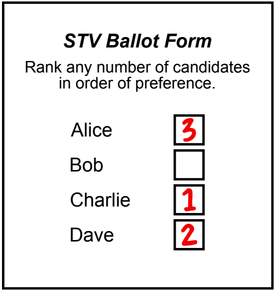
Counting: quota required to be elected;
who is weakest candidate ;
how to break ties;
how to transfer a vote;
when to stop counting

Nothing to do with electronic voting . . . yet

In particular, nothing to do with security aspects of e-voting

Single Transferable Vote Counting is Non-trivial

Vacancies: number of candidates that we need to elect
Candidates: number of people standing for election
Quota: how many votes are required to elect a candidate
Ballot: is a vote for highest ranked continuing candidate
Counting: proceeds in rounds
Surplus: ballots are transferred to next continuing candidate
Transfer Value: current value of ballot (possibly ≤ 1)
Eliminate Weakest: but how to break ties



Rounds: repeat until all seats filled
Tally: all highest preferences
Elected: All candidates with “quota” are elected
Eliminated: If nobody elected this round then eliminate weakest candidate
Transfer: compute new transfer values
Autofill: If can seat all remaining cands., do so

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

Seats: 2

Ballots: 5

$$A > B > D$$
$$A > B > D$$
$$A > B > D$$
$$D > C$$
$$C > D$$

Assume no fractional transfers and no autofill

Example   Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

Seats: 2

Ballots: 5

$$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$$

$$A > B > D$$
$$A > B > D$$
$$A > B > D$$
$$D > C$$
$$C > D$$

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$

Seats: 2

Ballots: 5

$A > B > D$	$votes(A) = 1$
$A > B > D$	$votes(A) = 2$
$A > B > D$	$votes(A) = 3$
$D > C$	$votes(D) = 1$
$C > D$	$votes(C) = 1$

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$$

Seats: 2

Ballots: 5

$$\begin{array}{ll} A > B > D & \\ A > B > D & \\ A > B > D & \\ D > C & votes(D) = 1 \\ C > D & votes(C) = 1 \end{array}$$

Elected: A

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$

Seats: 2

Ballots: 5

~~$A > B > D$~~

~~$A > B > D$~~

$A > B > D$

$D > C$ $\quad$ $votes(D) = 1$

$C > D$ $\quad$ $votes(C) = 1$

Elected: A

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$

Seats: 2

Ballots: 5

~~$A > B > D$~~

~~$A > B > D$~~

~~A~~ $> B > D$ $\quad votes(B) = 1$

$D > C$ $\quad votes(D) = 1$

$C > D$ $\quad votes(C) = 1$

Elected: A

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$

Seats: 2

Ballots: 5

~~$A > B > D$~~

~~$A > B > D$~~

$\not{A} > \not{B} > D \qquad votes(D) = 2$

$D > C$

$C > D \qquad votes(C) = 1$

Elected: A

Eliminated: B

Example Droop Quota: $Q = \left\lfloor \frac{totalnumberofballots}{seats+1} \right\rfloor + 1$

Candidates: $A, \quad B, \quad C, \quad D$

$Q = \left\lfloor \frac{5}{2+1} \right\rfloor + 1 = 2$

Seats: 2

Ballots: 5

~~$A > B > D$~~

~~$A > B > D$~~

$\not{A} > \not{B} > D$ $\quad votes(D) = 2$

$D > C$

$C > D$ $\quad votes(C) = 1$

Elected: $A, \quad D$

Eliminated: B

Existing Electronic Vote-counting in Australia

Australian Electoral Commission: proprietary code; not available for scrutiny; FOI request to publish code denied on grounds of "security" and "commercial in confidence"

Victorian Electoral Commission: proprietary code; available for scrutiny; no formal scrutiny to my knowledge

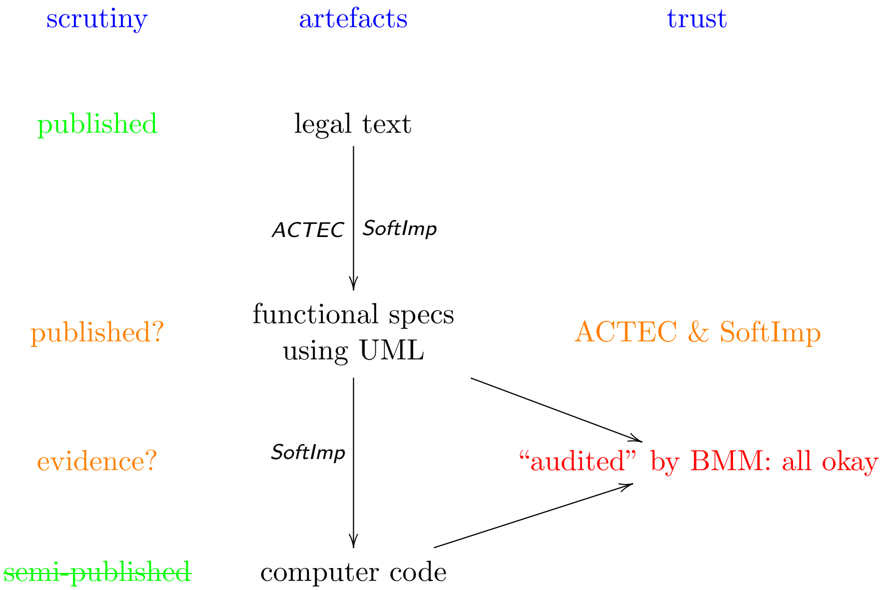
Australian Capital Territory: eVACS^{TM}

- developed by Software Improvements Pty Ltd. using C++
- used since 2001 to count four elections
- counting code used to be available from ACTEC website
- full code available if you sign a non-disclosure agreement

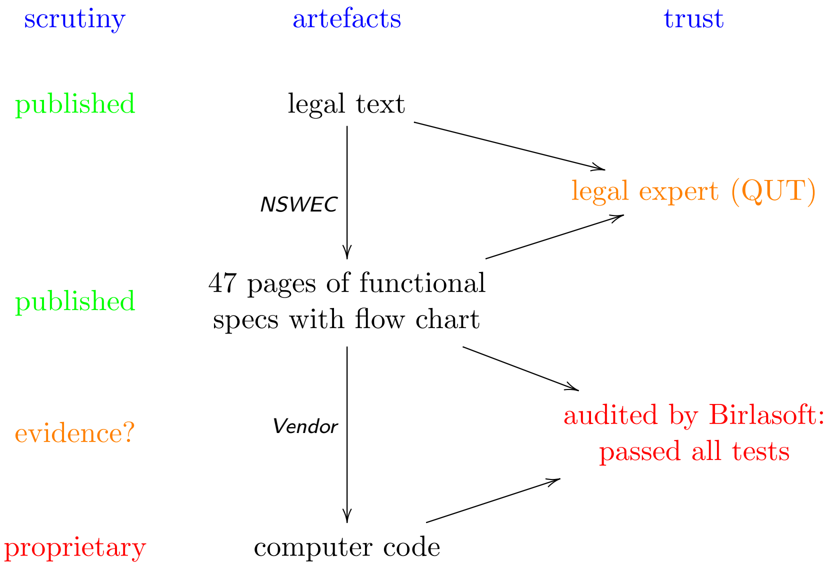
New South Wales Electoral Commission: detailed functional requirements publicly available; found to comply with legislation by legal expert from QUT; certified by Birlasoft as passing all tests; proprietary code; code not available for scrutiny

^{TM}eVACS is a trademark of Software Improvements Pty Ltd.

ACTEC and SoftImp Approach



NSWEC Approach



Bugs in ACT and NSW Counting Modules

ANU logic group: found three bugs in eVACS

programming error: simple for-loop bounds error

ambiguous legal text: break weakest candidate ties by inspecting previous round where *"all candidates have an unequal number of votes"*

programming error: un-initialised boolean: different compilers give different results

how bad: for every bug, we could generate an election in which the code gave the wrong result

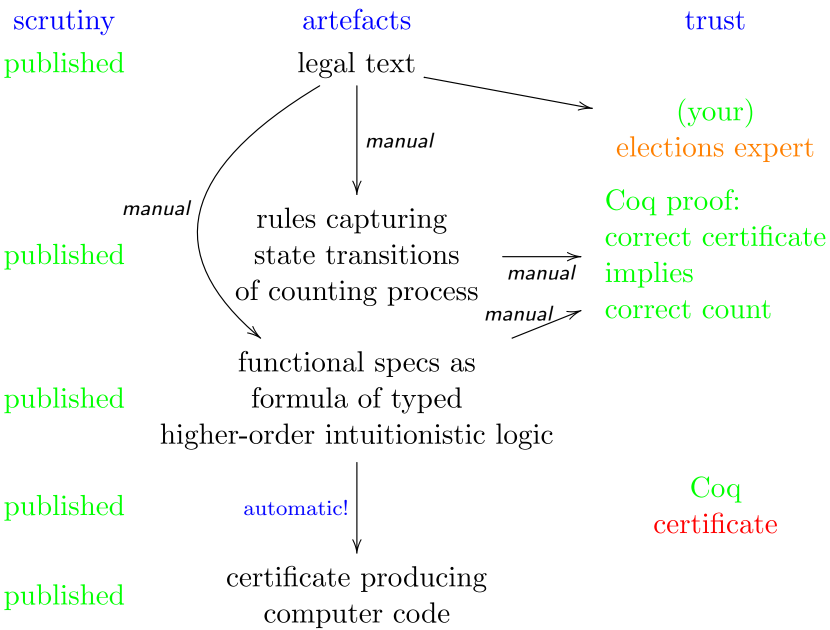
UniMelb group: found bug in NSWEC code whereby one candidate's chances of winning were reduced from 90% to 10% and she lost the 2015 election! No recourse as the three month period for a legal challenge had passed.

"Simplifications" in ACT Legislation Are Harmful

ANU logic group: we showed that

- Rounding (fractions): errors can become significant
- Point of declaring winners: can be significant
- "Last parcel" simplication: is just silly
- How bad: for every "simplification", there is an election where legislation gave the wrong result w.r.t. Vanilla STV
- And ... these cases do happen in real elections e.g. Brindabella

Efficient Interactive Synthesis Via Mathematical Proof



Minimal STV: Abstract Machine

Three types of states: initial states (all ballots uncounted); final states (election winners are declared); intermediate states

Data "carried" by non-initial states: 7 items

1 list of currently uncounted ballots;

2-3 tally t and pile p of ballots "for" each candidate;

4-5 elected/eliminated candidate lists (bl_1, bl_2) requiring transfer;

6-7 lists of elected e and continuing h candidates

State Transitions: correspond to counting, eliminating, transferring, electing, and declaring winners as formal rules that relate a pre-state and a post-state via conditions

Variations: so minimal STV does not define the rules, but rather postulates minimal conditions that every rule needs to satisfy

Inductive definition of STV machine states in Coq

```
Inductive mynat : Set :=
  | O : mynat                (* O is a mynat *)
  | S : mynat -> mynat.      (* S of a mynat is a mynat *)
```

```
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.
```

Inductive definition of STV machine states in Coq

```
Inductive mynat : Set :=
  | O : mynat                (* O is a mynat *)
  | S : mynat -> mynat.      (* S of a mynat is a mynat *)

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.
```

Minimal STV: an instance

An instance: of STV is then given by

- definitions: rules for counting, electing, eliminating, transfering
- proofs: that rules satisfy the respective conditions

Conditions: consist of two parts

- applicability: conditions for when the rule is applicable
- progress: how the rule changes the state

Prove: three theorems

- reduction: every applicable transition reduces "complexity"
- liveness: at least one transition from each non-final state
- termination: minimal STV terminates

Code Extraction and Certificates

Encoding: into Coq which is based on intuitionistic logic

Constructive proofs: of theorems of the form $\forall x \exists y, \varphi(x, y)$ correspond to lambda-terms

Code Extraction: automatically extract Haskell code

Certificates: the theorems stated so the extracted code produces a run of the state machine as evidence that the result is correct

Claim: it is easy to write a program to check that the certificate is correct wrt the rules

Code Extraction and Certificates

Encoding: into Coq which is based on intuitionistic logic

Constructive proofs: of theorems of the form $\forall x \exists y, \varphi(x, y)$ correspond to lambda-terms

Code Extraction: automatically extract Haskell code

Certificates: the theorems stated so the extracted code produces a run of the state machine as evidence that the result is correct

Claim: it is easy to write a program to check that the certificate is correct wrt the rules

Example: certificates and checking

```
Inductive add: mynat -> mynat -> mynat -> Prop :=
| add0: forall n, (add n 0 n)
| addS: forall n m r, add n m r -> add n (S m) (S r).
```

```
---------------------------------- add0
      add (S 0) 0 (S 0)
------------------------------------- addS
   add (S 0) (S 0) (S S 0)
---------------------------------------- addS
 add (S 0) (S S 0) (S S S 0)
------------------------------------------- addS
add (S 0) (S S S 0) (S S S S 0)
```

```
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]
```

Checking: simple pattern matching on rule definitions

Example: certificates and checking

```
Inductive add: mynat -> mynat -> mynat -> Prop :=
| add0: forall n, (add n 0 n)
| addS: forall n m r, add n m r -> add n (S m) (S r).
```

```
 ---------------------------------- add0
      add (S 0) 0 (S 0)
 ---------------------------------- addS
   add (S 0) (S 0) (S S 0)
 ---------------------------------- addS
   add (S 0) (S S 0) (S S S 0)
 ---------------------------------- addS
 add (S 0) (S S S 0) (S S S S 0)
```

```
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]
```

Checking: simple pattern matching on rule definitions

Features and Further Work

Completed: STV vote-counting and Schulze Method

Exact fractions: our code for STV manipulates fractions exactly

Efficiency: can (STV) count up to 10 million votes with 40 candidates and 20 vacancies in 20 minutes

Certificate: our code produces a (plain text) certificate that vouches for the correctness of the count

Scrutiny: program to check the certificate is correct w.r.t. published rules and published ballots is just pattern matching

Trust: you don't even need to trust the hardware or software since a correct certificate implies a correct count

Caveat: have to publish all ballots

Further Work: can we extend to STV counting of encrypted ballots

Features and Further Work

Completed: STV vote-counting and Schulze Method

Exact fractions: our code for STV manipulates fractions exactly

Efficiency: can (STV) count up to 10 million votes with 40 candidates and 20 vacancies in 20 minutes

Certificate: our code produces a (plain text) certificate that vouches for the correctness of the count

Scrutiny: program to check the certificate is correct w.r.t. published rules and published ballots is just pattern matching

Trust: you don't even need to trust the hardware or software since a correct certificate implies a correct count

Caveat: have to publish all ballots

Further Work: can we extend to STV counting of encrypted ballots

Features and Further Work

Completed: STV vote-counting and Schulze Method

Exact fractions: our code for STV manipulates fractions exactly

Efficiency: can (STV) count up to 10 million votes with 40 candidates and 20 vacancies in 20 minutes

Certificate: our code produces a (plain text) certificate that vouches for the correctness of the count

Scrutiny: program to check the certificate is correct w.r.t. published rules and published ballots is just pattern matching

Trust: you don't even need to trust the hardware or software since a correct certificate implies a correct count

Caveat: have to publish all ballots

Further Work: can we extend to STV counting of encrypted ballots

Why Should We Trust Machine-checked Proof?

scrutiny	artefacts	trust
published	1930s Alonzo Church's typed λ-calculus	peer review
	↓ *INRIA*	logic community
published	Coq theorem prover: 50K lines of OCaml code	Coq development team
	↕ *person*	
published	proof checked by Coq	OCaml compiler and hardware

Further Work, Caveats and Conclusions:

Verified Certificate Checker: using CakeML to verify our certificate checker against a formal model of the semantics of C

Other flavours of STV: cover all STV schemes used in Australia

Effort: approximately 4 person-months of work by a Coq novice

Caveat: relies on EMB publishing the ballots in clear text so it is vulnerable to the Sicilian Attack

Shufflesum: currently trying to synthesise the code

Conclusion: verified synthesis possible for complex e-counting