Modular Formal Verification of STV Algorithms: modular synthesis of STV programs

Milad Ghale, Rajeev Goré, Dirk Pattinson, Mukesh Tiwari
Australian National University

Rajeev.Gore@anu.edu.au

October 4, 2018

Overview

Evidence-based Trust Versus "Trust Me"

Program Synthesis via Coq

Single Transferable Voting (STV) Example

"Last Parcel" Modification

An Abstract State Machine

Sanity Checks and Measure of Progress

Interactive Synthesis of Vote Counting Programs

Results and Features

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?

or the hardware is compromised?

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?

or the hardware is compromised?

Software and Hardware Independence:

Certificates vote-counting programs must produce a tallying script

Proofs if the tallying script is correct then the result is correct

Easy Scrutiny easy to write a program to check tallying script

Synthesising a program for adding two natural numbers

Finite representation via "+": 0 1 2 ...

0 is a natural number

if n is a natural number then so is (n+1)

nothing else is a natural number

Finite representtion via "S": for "successor"

`O` is a natural number

if `n` is a natural number then so is `(S n)`

nothing else is a natural number

Coq: definition of a set called `mynat`

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

`mynat`: `O (S O) (S S O) ...`

Scrutiny: does `mynat` behave as expected?

Synthesising a program for adding two natural numbers

Finite representation via "+": 0 1 2 ...

0 is a natural number

if n is a natural number then so is (n+1)

nothing else is a natural number

Finite representtion via "S": for "successor"

0 is a natural number

if `n` is a natural number then so is `(S n)`

Coq: definition of a set called `mynat`

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

`mynat`: `O (S O) (S S O) ...`

Scrutiny: does `mynat` behave as expected?

Synthesising a program for adding two natural numbers

Finite representation via "+": 0 1 2 ...

0 is a natural number

if n is a natural number then so is (n+1)

nothing else is a natural number

Finite representtion via "S": for "successor"

0 is a natural number

if `n` is a natural number then so is `(S n)`

nothing else is a natural number

Coq: definition of a set called `mynat`

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

`mynat`: `O (S O) (S S O) ...`

Scrutiny: does `mynat` behave as expected?

Synthesising a program for adding two natural numbers

Finite representation via "+": 0 1 2 ...

0 is a natural number

if n is a natural number then so is (n+1)

nothing else is a natural number

Finite representtion via "S": for "successor"

`O` is a natural number

if `n` is a natural number then so is `(S n)`

nothing else is a natural number

Coq: definition of a set called `mynat`

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

`mynat`: `O (S O) (S S O) ...`

Scrutiny: does `mynat` behave as expected?

Specifying addition: from natural language to Coq

Version 1: natural language

add0: adding n to 0 gives n

add1: if adding n to m gives r then adding n to m+1 gives r+1

Version 2: some `equations` and some natural language

add0: `add n 0 = n`

add1: if `add n m = r` then `add n (m+1) = (r+1)`

Version 3: in logic to remove the equations

add0: `add n 0 n` is true

add1: `add n m r` $\rightarrow$ `add n (m+1) (r+1)`

Version 4: in Coq replacing (+1) with `S` and eliding "is true"

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

Scrutiny: does Version 4 captures Version 1 ?

Specifying addition: from natural language to Coq

Version 1: natural language

add0: adding n to 0 gives n

add1: if adding n to m gives r then adding n to m+1 gives r+1

Version 2: some `equations` and some natural language

add0: `add n 0 = n`

add1: if `add n m = r` then `add n (m+1) = (r+1)`

Version 3: in logic to remove the equations

add0: `add n 0 n` is true

add1: `add n m r` $\rightarrow$ `add n (m+1) (r+1)`

Version 4: in Coq replacing (+1) with `S` and eliding "is true"

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

Scrutiny: does Version 4 captures Version 1 ?

Specifying addition: from natural language to Coq

Version 1: natural language

add0: adding n to 0 gives n

add1: if adding n to m gives r then adding n to m+1 gives r+1

Version 2: some `equations` and some natural language

add0: `add n 0 = n`

add1: if `add n m = r` then `add n (m+1) = (r+1)`

Version 3: in logic to remove the equations

add0: `add n 0 n` is true

add1: `add n m r` $\rightarrow$ `add n (m+1) (r+1)`

Version 4: in Coq replacing (+1) with `S` and eliding "is true"

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

Scrutiny: does Version 4 captures Version 1 ?

Specifying addition: from natural language to Coq

Version 1: natural language

add0: adding n to 0 gives n

add1: if adding n to m gives r then adding n to m+1 gives r+1

Version 2: some `equations` and some natural language

add0: `add n 0 = n`

add1: if `add n m = r` then `add n (m+1) = (r+1)`

Version 3: in logic to remove the equations

add0: `add n 0 n` is true

add1: `add n m r` $\rightarrow$ `add n (m+1) (r+1)`

Version 4: in Coq replacing $(+1)$ with `S` and eliding "is true"

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

Scrutiny: does Version 4 captures Version 1 ?

Extracting a correct implementation myplus via Coq

Theorem For all n and m, there is an r such that $add\ n\ m\ r$ holds.

```
Theorem myplus: forall n m, { r | add n m r }.
Proof. ........ Defined.
Extraction "myplus.ml" myplus.
```

Extracted OCaml program (* my comments *) in file myplus.ml

```
let rec myplus n = function
| O -> n                          (* case when m is O *)
| S n0 -> S (myplus n n0)         (* case when m is S . *)
```

Can compile and run this program

```
myplus (S O) (S (S (S O)));;
 - : Myplus.mynat = S (S (S (S O)))
```

Certificate the program needs to print its trace

Extracting a correct implementation myplus via Coq

Theorem For all n and m, there is an r such that $add\ n\ m\ r$ holds.

```
Theorem myplus: forall n m, { r | add n m r }.
Proof. ........ Defined.
Extraction "myplus.ml" myplus.
```

Extracted OCaml program (* my comments *) in file myplus.ml

```
let rec myplus n = function
| O -> n                          (* case when m is O *)
| S n0 -> S (myplus n n0)         (* case when m is S . *)
```

Can compile and run this program

```
myplus (S O) (S (S (S O)));;
 - : Myplus.mynat = S (S (S (S O)))
```

Certificate the program needs to print its trace

Extracting a correct implementation myplus via Coq

Theorem For all n and m, there is an r such that $add\ n\ m\ r$ holds.

```
Theorem myplus: forall n m, { r | add n m r }.
Proof. ........ Defined.
Extraction "myplus.ml" myplus.
```

Extracted OCaml program (* my comments *) in file myplus.ml

```
let rec myplus n = function
| O -> n                          (* case when m is O *)
| S n0 -> S (myplus n n0)         (* case when m is S . *)
```

Can compile and run this program

```
myplus (S O) (S (S (S O)));;
 - : Myplus.mynat = S (S (S (S O)))
```

Certificate the program needs to print its trace

Extracting a correct implementation myplus via Coq

Theorem For all *n* and *m*, there is an *r* such that *add n m r* holds.

```
Theorem myplus: forall n m, { r | add n m r }.
Proof. ........ Defined.
Extraction "myplus.ml" myplus.
```

Extracted OCaml program (* my comments *) in file myplus.ml

```
let rec myplus n = function
| O -> n                          (* case when m is O *)
| S n0 -> S (myplus n n0)         (* case when m is S . *)
```

Can compile and run this program

```
myplus (S O) (S (S (S O)));;
 - : Myplus.mynat = S (S (S (S O)))
```

Certificate the program needs to print its trace

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

Candidates: A, B, C

Seats: 1

Ballots: 4

$A > C > B$	1/1
$B > C > A$	1/1
$C > A$	1/1
$C > B > A$	1/1

Elected: none

Eliminated: none

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

Candidates: A, B, C

Seats: 1

Ballots: 4

$$Q = \left\lfloor \frac{4}{1+1} \right\rfloor + 1 = 3$$

$A > C > B$	1/1
$B > C > A$	1/1
$C > A$	1/1
$C > B > A$	1/1

Elected: none

Eliminated: none

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

Candidates: A, B, C

Seats: 1

Ballots: 4

$$Q = \left\lfloor \frac{4}{1+1} \right\rfloor + 1 = 3$$

$A > C > B$	$1/1$	$votes(A) = 1$
$B > C > A$	$1/1$	$votes(B) = 1$
$C > A$	$1/1$	$votes(C) = 1$
$C > B > A$	$1/1$	$votes(C) = 2$

Elected: none

Eliminated: none

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

Candidates: A, B, C

Seats: 1

Ballots: 4

$$Q = \left\lfloor \frac{4}{1+1} \right\rfloor + 1 = 3$$

$A > C > B$	$1/1$	$votes(A) = 1$
$B > C > A$	$1/1$	$votes(B) = 1$
$C > A$	$1/1$	$votes(C) = 1$
$C > B > A$	$1/1$	$votes(C) = 2$

Elected: none: we have to eliminate weakest candidate

Eliminated: none

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

Candidates: A, B, C

Seats: 1

Ballots: 4

$$Q = \left\lfloor \frac{4}{1+1} \right\rfloor + 1 = 3$$

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

Elected: none

Eliminated: A but ballot retains full value

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

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

$$Q = \left\lfloor \frac{4}{1+1} \right\rfloor + 1 = 3$$

Seats: 1

Ballots: 4

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

Elected: C

Eliminated: A but ballot retains full value

Last Parcel Transfer Rule Example: ACT

Last Parcel Simplification: suppose we needed to transfer C's ballots, we would not consider the third and fourth ballots because they are not part of C's "last parcel"

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 O n)
| addS: forall n m r, add n m r -> add n (S m) (S r).
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{add (S O) O (S O)}}\,\texttt{add0}}{\texttt{add (S O) (S O) (S S O)}}\,\texttt{addS}}{\texttt{add (S O) (S S O) (S S S O)}}\,\texttt{addS}}{\texttt{add (S O) (S S S O) (S S S S O)}}\,\texttt{addS}$$

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 O n)
| addS: forall n m r, add n m r -> add n (S m) (S r).
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{add (S O) O (S O)}}\,\text{add0}}{\texttt{add (S O) (S O) (S S O)}}\,\text{addS}}{\texttt{add (S O) (S S O) (S S S O)}}\,\text{addS}}{\texttt{add (S O) (S S S O) (S S S S O)}}\,\text{addS}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{initial [([a,c,b],1/1),([b,c,a],1/1),([c,a],1/1),([c,b,a],1/1)]}}}{\text{state [([a,c,b],1/1),([b,c,a],1/1),([c,a],1/1),([c,b,a],1/1)]; a[0/1] b[0/1] c[0/1]; a[] b[] c[]; ([],[]); []; [a,b,c]}}\,\text{start}}{\text{state []; a[1/1] b[1/1] c[2/1]; a[[([a,c,b],1/1)]] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[]); []; [a,b,c]}}\,\text{count}}{\text{state []; a[1/1] b[1/1] c[2/1]; a[[(a,c,b],1/1)]] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[a]); []; [b,c]}}\,\text{eliminate}}{\text{state [([a,c,b],1/1)]; a[1/1] b[1/1] c[2/1]; a[] b[[([b,c,a],1/1)]] c[[([c,a],1/1),([c,b,a],1/1)]]; ([],[a]); []; [b,c]}}\,\text{transfer-removed}}{\text{state []; a[1/1] b[1/1] c[3/1], a[] b[[([b,c,a],1/1)]] c[[(a,c,b],0/1)]]; ([c],[a]); [c]; [b]}}\,\text{count}}{\text{winners [c]}}\,\text{elect win}$$

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