

Modular Synthesis of Verified Verifiers of Computation with STV Algorithms

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

Dirk Pattinson
The Australian National University
Canberra, Australia
dirk.pattinson@anu.edu.au

Michael Norrish
Data61, CSIRO, and ANU
Canberra, Australia
Michael.Norrish@data61.csiro.au

Abstract—Single transferable vote (STV) is a family of preferential voting systems, different instances of which are used in binding elections throughout the world. We give a formal specification of this family, from which we derive fully verified tools that verify the computation for various instances of STV vote counting. These tools validate an execution of an STV vote counting algorithm, based on a transcript of the count.

Our framework distils the similarities and differences of various instances of STV and gives a uniform and modular way of synthesising verifiers for its various instances, and provides the flexibility and ease for adapting and extending it to a variety of STV schemes. We minimise the trusted base in correctness of the tools produced by using the HOL4 and CakeML as the technical basis. We first formally specify and verify the tools in HOL4 and then obtain the machine executable versions for the tools by relying on the verified proof translator and the compiler of CakeML. Moreover, proofs that we establish in HOL4 and CakeML are almost completely automated so that new verified instances of STV can be created with no (or minimal) extra proof. Finally, our experimental results with executable code demonstrate feasibility of deploying the framework for verifying real size elections having an STV counting algorithm.

Index Terms—Framework, Modularity, Single Transferable Voting, Algorithms, Variable Computation, Proof Engineering, Code Synthesis, Elections, Trustworthiness, Transparency

I. INTRODUCTION

Single transferable vote (STV) [9] is a preferential voting scheme used in various jurisdictions around the world, including Ireland, Malta, India, Pakistan, Australia, New Zealand and some municipalities of the United States, typically for multi-seat constituencies. A ballot is a preference-ordered list of candidates, typically obtained specifying a numerical preference for each candidate appearing on the ballot paper.

Each specific STV scheme first determines a *quota*, i.e. the (minimal) number of votes that a candidate needs to attract to be elected (often the Droop quota [8]). The quota, mostly but not always, depends on the number of ballots cast, and the number of vacancies to fill. Once the quota has been determined, tallying proceeds as follows:

- 1) count all first preference votes
- 2) if a candidate's first preference votes are above the quota, this candidate is elected
- 3) if an elected candidate has received votes in excess of the quota, the surplus votes are transferred to the next preference

- 4) if there are still vacancies to fill and no candidate is being elected, the candidate with the least number of first preference votes is eliminated, and their votes are transferred to the next preference (possibly resulting in more candidates being elected)
- 5) the process continues until either all vacancies are filled, or the number of elected and continuing (not yet eliminated) candidates matches the number of vacancies.

This informal description is necessarily imprecise, with details differing widely between jurisdictions. For example, there are substantial differences in the details of precisely which excess ballots are being transferred after candidates have been elected, or the order in which surplus ballots are being transferred.

The goal of this paper is to advance formal methods in elections by demonstrating how STV counting can be verified in a modular, transparent, and practical feasible way over a minimal computational trust base: we show how to synthesise efficient and provably correct verifiers for a large class of STV schemes in a modular and almost fully automated way.

We achieve this by first giving a formal definition of STV (called *generic STV*) that encompasses the entire family of STV vote counting schemes. This definition necessarily under-specifies any particular STV vote counting scheme, but fixes both basic data structures and the mode of computation common to all instances of STV. Trustworthiness is achieved by not verifying the *software* that computes the result, but instead by verifying *evidence* that substantiates the correctness of the count. Evidence records and corresponds to the individual steps taken in the computation of winners (and could so also be verified, or spot-checked manually), giving transparency. We minimize the computational trust base by verifying our executables down to the machine level using a combination of HOL [15] and CakeML [4].

We observe that every instance of STV performs the same *actions* (e.g. count first preferences, transfer a surplus, eliminate a candidate, etc.) where the precise interpretation of actions differ. We formalise these actions as rules that advance the state of the count towards a final result. Evidence for an instance of computation then becomes a sequence of rule applications applied to perform this computation. While generic STV abstracts the commonalities, each *instance* of STV formally specifies the pre- and postconditions for each

rule as well as a boolean function that decides whether a rule has been applied correctly.

Executable evidence verifiers are constructed almost automatically: nearly all of the ensuing proof obligations are discharged generically, and we fully automate the translation into a machine-level verified executable using CakeML.

We evaluate our framework by analysing the degree of automation and modularity achieved, and by giving realistic benchmarks results over real-world election data.

In summary, we present a framework that allows us to formally specify instances of STV and synthesise an executable certificate verifier in a modular way that is itself verified down to the machine level.

II. EVIDENCES

Our problem is the following: how can we verify that an execution of an implementation \mathcal{P} of an arbitrary known STV algorithm \mathcal{A} correctly computes the end result according to \mathcal{A} , even if \mathcal{P} 's source code is secret?

To begin assume \mathcal{P} is an implementation of an algorithm \mathcal{A} . We require that each execution of \mathcal{P} on a given input x (of ballots cast in an election) to output both winners y and evidence ω as proof of correctness of the execution. Having evidence available for each execution of \mathcal{P} facilitates checking correctness of the computation carried out *independently* of (the source code of) \mathcal{P} . Therefore the original problem boils down verifying (the correctness of) evidence produced by an *instance of computation* of an algorithm \mathcal{A} .

The evidence as such must be enough informative to provide transparency of tallying and also allow voters, or at least a large pool of scrutineers, to verify for themselves that the tally is authentically processed as per instructions of the counting scheme so that winners truly reflect the will of the voters. Therefore the question becomes what information is enough for establishing transparency and verifiability of tallying and what data structure for recording the information in the evidence we should choose.

A. Characterisation of Evidence

A comprehensive analysis of different STV algorithms reveals the existence of a common underlying abstract data type among STV algorithms as follows.

- Some ballots to deal with (assuming) recorded as cast by voters. Each ballot is a pair consisting of a list expressing the preference of a voter and a fractional transfer value,
- a quota which is the least amount of votes needed to be elected,
- the candidates competing in the election,
- the number of vacancies for which candidates compete,
- the *discrete states of computation* which encapsulate necessary pieces of information needed to transparently know how the tallying has progressed from beginning to end and to be able to independently verify its correctness provided access to the information is granted.

There are two kinds of discrete states of computation. Some of them are final states where tallying has reached a halting state

by finding the winners of the election. The other consist of non-final ones where bits and pieces of data are manipulated according to the counting mechanism of the STV scheme used, to eventually obtain an end result. Each non-final state comprises seven pieces of data:

- **uncounted ballots** which await being counted,
- **tally** for holding information about the amount of votes that each candidate has hitherto attracted
- **pile** to know the ballots allocated to each candidate,
- **backlog of elected** candidates who have exceeded the quota and their surplus votes awaits being transferred,
- **backlog of eliminated** ones whose votes await transfer,
- **elected candidates** who are the already elected ones,
- **continuing candidates** still continuing in the election.

The non-final and final states specified above provide us with all the information one needs. We therefore find a perfect characterisation for evidence.

Definition 1: Suppose an STV algorithm \mathcal{A} is given and Q , s and C are respectively the quota, number of vacancies and competing candidates in an election whose counting algorithm is \mathcal{A} . Also assume y is the output of an execution of \mathcal{A} on an input x . By evidence $\hat{\omega}$ for the execution we mean the quadruple (Q, s, C, Ω) , where Ω is the chronological sequence of all states of this instance of computation visited from the initial state where x is input to the final state where y is output.

III. THE GENERIC STV MACHINE

We have already discussed an abstract data type whose values formally represent evidence. However we do not yet possess an operational semantics for manipulating data to validate evidence. On one hand, the semantics must be flexible enough to accommodate variations existing in the counting mechanism of STV algorithms so that we can produce verifiers specifically operating for validation of evidence output by their associated scheme. On the other hand it must facilitate designing and developing the framework modularly while offering automation for synthesising verifiers.

We also identify a general algorithmic mechanism to perform computation on evidence, which realises the *commonalities* among the members of the STV family. This comprises seven general actions (Definition 2) corresponding to actions that tally officers take when counting votes.

Definition 2 (Generic Counting Actions): The following comprise the generic STV counting actions.

- **count** for counting the uncounted ballots,
- **elect** for electing all or some of the electable candidates,
- **transfer-elected** for distributing surplus votes of elected,
- **eliminate** to exclude one or some candidates,
- **transfer-removed** to distribute votes of the eliminated,
- **elected-win** for terminating tallying whenever the vacancies are filled or there is no continuing candidate left,
- **hopeful-win** for terminating tallying whenever the number of already elected and continuing candidates collectively does not exceed the initial vacancies.

Each action consists of preconditions on when and how the action comes into effect, and postconditions dictating what

effect applying the action brings about by describing the new state to which the computation proceeds. In fact, these conditions are the formal counterparts of the legal clauses in the textual specification of the counting scheme which inform tally officers how to proceed with tallying, and they are common between all flavours of STV.

For example, a constant precondition for applying elimination asserts that there must be no uncounted ballot left and that no candidate should be electable at the current state of counting. A postcondition of elimination is that once a candidate is excluded from counting, the list of continuing candidates is updated accordingly so that the excluded candidate no longer received votes. Conjunctions of such declarations define what an action is and what an application of that action means.

The underlying abstract data type and the universal algorithmic pattern found in STV characterise components of a finite state machine which we call *generic STV* (Definition 3). The set of machine states consist of the discrete states of computation \mathcal{S} mentioned earlier. The seven generic actions in Definition 2 form the transition labels \mathcal{T} of the machine. Last, the everywhere present pre- and post-conditions which define (content of) the actions, and constitute a small-step operational semantics for the machine. Note that the states of computation recorded as evidence now become identical with machine states.

Definition 3 (The generic STV machine): Let \mathcal{S} and \mathcal{T} be respectively the set of machine states and transition labels. Also assume for $t \in \mathcal{T}$, $S_t = \bigwedge_{i \leq j_t} \psi_i$ where ψ_i is the formal specification (in higher-order logic) of a pre- or post-condition of t . Then the *generic STV* is the triple $M = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$. There are nonetheless variations in STV algorithms which we recognise by separate *instantiations* into the machine.

A. Instantiations of the Machine with Specific STV schemes

Differences exist between counting algorithms of STV schemes. For example in the STV used for electing the upper house representatives of the Australian State of Victoria (Victoria STV) [19], distributing votes of an eliminated candidate occurs step by step in order of the magnitude of the fractional values that those votes carry. Consequently, a precondition for applying the action is to first assure that upon each instance of applying transfer-removed action (a) exactly those votes which have the same fractional value are transferred and (b) the transfer happens according to the magnitude order. In contrast, for example, the STV used in the lower house elections of the Australian state of Tasmania (Tasmania STV) does not have such precondition simply because votes of an eliminated candidate are transferred in a single application of transfer-removed. There are also variations in postconditions. For example transfer-elected of the ACT STV [1] used in the lower house elections of the Australian Capital Territory requires distributing only the last parcel of surplus votes received by an elected candidate. But Victoria STV specifies transferring all surplus votes of an elected candidate.

We formally realise variations of STV algorithms by instantiation (Definition 4) of the generic machine. Recall that

the semantics of the machine (Definition 3) is formed by conjunctions of formally specified universally appearing pre- and postconditions in STV algorithms. An instantiation of the machine with an STV algorithm \mathcal{A} happens by enriching the generic semantics by adding formal specifications of the legal clauses that are specific to \mathcal{A} . We hence come to define an operational semantics functioning in accordance with instructions of \mathcal{A} .

Definition 4 (Instantiation of the Machine): Assume the generic machine $M = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ as in Definition 3. An instantiation $\hat{\mathcal{A}}$ for an STV algorithm \mathcal{A} of the generic STV machine M is a triple $\langle \mathcal{S}, \mathcal{T}, (S'_t)_{t \in \mathcal{T}} \rangle$, where for each $t \in \mathcal{T}$, $S'_t = S_t \wedge \bigwedge_{i \leq j'_t} \phi_i$. Each ϕ_i is the formal specification of a pre- or postcondition specific to \mathcal{A} .

The operational semantics of each instantiation allows performing operations such as validation on evidence. Definition 5 lays down what verifying evidence amounts to. Informally speaking, to check if evidence is valid one simply needs to inspect if transitions between each two consecutive states of computation appearing in the evidence occur by a legitimate application of a counting action of the scheme used.

Definition 5 (verifier): Assume $\hat{\mathcal{A}} = \langle \mathcal{S}, \mathcal{T}, (S'_t)_{t \in \mathcal{T}} \rangle$ is an instantiation of the STV algorithm \mathcal{A} . A *verifier* for instances of computation of \mathcal{A} is a function \hat{V} mapping $\mathbb{Q} \times \mathbb{N} \times 2^C \times \text{List}(\mathcal{S})$ to $\{0, 1\}$ such that for any putative evidence $\hat{\omega} = (Q, s, C, \Omega)$ where $\Omega = \langle \Omega_1, \dots, \Omega_n \rangle$, $\hat{V}(\hat{\omega}) = 1$ if and only if the following holds:

$\forall i \in \{1, \dots, n-1\}. \exists t \in \mathcal{T}. (\Omega_{i-1}, \Omega_i) \in t \wedge \vdash S'_t[\Omega_{i-1}, \Omega_i]$ where \mathbb{Q} and \mathbb{N} respectively represent the set of rational and natural numbers, 2^C is the set of all subsets of C , and $\text{List}(\mathcal{S})$ is the set of all possible lists of machine states. The state Ω_{i-1} is a pre-state and Ω_i is a post-state visited in the execution. The last line means that pre- and post-conditions of applying t to move from Ω_{i-1} to Ω_i are logically true.

We shall next provide an example to concretely illustrate a piece of evidence and exemplify how STV schemes work.

B. An Example of a Concrete Piece of Evidence

Figure 1 shows evidence of a small election whose counting algorithm is the ACT STV with three candidates a , b and c , and initially recorded ballots containing $b_1 = ([a, c], 1/1)$, $b_2 = ([a, b, c], 1/1)$, $b_3 = ([a, c, b], 1/1)$, $b_4 = ([b, a], 1/1)$, $b_5 = ([c, b, a], 1/1)$. The first component of each ballot is a preference-ordered list of candidates, and the second is the fractional transfer value (initially set to $1/1$). The evidence consists of a *header* that specifies the quota (computed according to the ACT STV instructions), the number of vacancies, and the list of competing candidates. The fourth line is the election result, and the remainder of the evidence are non-final states visited to compute this outcome. Each non-final state shows uncounted ballots, tallies of candidates, piles of votes attracted, backlog of the elected, backlog of the eliminated, list of elected candidates, and list of continuing candidates.

First preferences for each candidate are computed. Ballots counted in favour of particular candidates are placed onto that candidate's pile. In this election, a is the first preference on

b_1 , b_2 , and b_3 , b receives b_4 , and c receives b_5 . Tallies are updated so that tally of a becomes 3, and b and c each reach $1/1$. As a exceeds the quota, (s)he is elected and the fractional value of his surplus is updated according to the ACT scheme. Then a 's surplus is transferred to continuing candidates b and c and votes are counted according to next preferences shown on a 's surplus votes. Candidate b and c each attract one vote. However neither of them reaches the quota. Hence the weakest one b is eliminated. As the number of elected and continuing candidates is equal to the initial vacancies, the election terminates by declaring a and c to be winners.

IV. THE FRAMEWORK ARCHITECTURE

We progress through three macro-level phases to develop the framework. For the first phase, we use HOL4 to formally specify, implement and verify the generic STV, its instantiations, verifiers, and every auxiliary assertion needed for these ends. The second step relies on the verified proof translation tool of CakeML to provably correctly translate implementations of the first phase into equivalent CakeML implementations. The last phase uses the ecosystem of CakeML and its verified compiler to generate machine executable evidence verifiers.

Figure 2 is a simplified schematic illustration of the framework modules and their dependencies. We explain the purpose of each and then describe how they collectively function.

Auxiliary: The module contains specification, implementation and verification of the generic STV. It consists of subdivisions for the formal specification of the machine and its components, implementation of functions meant to be the computational counterparts of the specifications and verification of the implementations by proving that they logically satisfy their respective specifications. Moreover we include specification, implementation and verification of helper assertions which appear in many well-known STV schemes here so that Auxiliary also serves as a comprehensive library for STV.

Instantiations: Instantiations of the generic STV happen in separate modules which we have schematically shown only two, namely STV_1 and STV_2 . An instantiation module STV_i for an algorithm \mathcal{A} consists of a subdivision for specification $\hat{\mathcal{A}}_{spec}$ of the scheme and another one its implementation $\hat{\mathcal{A}}_{dec}$.

Proofs: We prove that for an algorithm \mathcal{A} in the above modules, $\hat{\mathcal{A}}_{dec}$ is logically equivalent to $\hat{\mathcal{A}}_{spec}$. We automate proofs so that they work for various instantiations.

Verifier: This module formally realises the notion of verifier in Definition 5. However, it is developed in a way that automatically operates regardless of which STV algorithm is

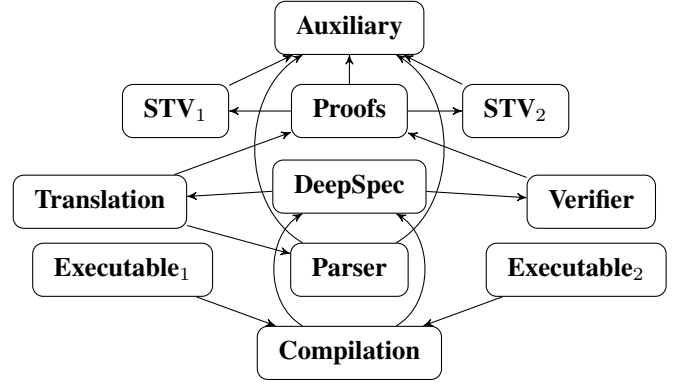


Fig. 2. Architecture of the Framework, direction of arrows represent module dependencies

instantiated into the machine. It has three subdivisions. A specification $\hat{\mathcal{V}}_{spec}$ defining what a verifier is, the implementation $\hat{\mathcal{V}}_{dec}$ and proofs of equivalence of $\hat{\mathcal{V}}_{spec}$ and $\hat{\mathcal{V}}_{dec}$.

Translation: We translate each implementation \hat{f}_{imp} in HOL4 to a proven equivalent implementation f_τ in CakeML's environment by using the verified proof translator of CakeML. For example, an implementation $\hat{\mathcal{A}}_{imp}$ of a scheme \mathcal{A} and its verifier $\hat{\mathcal{V}}_{imp}$ are respectively translated into \mathcal{A}_τ and \mathcal{V}_τ .

DeepSpec: For any translated verifier \mathcal{V}_τ and the translated evidence parser \mathcal{P}_τ , we obtain logically equivalent deep CakeML embeddings \mathcal{V}^* and \mathcal{P}^* , respectively. We then use them to establish end-to-end desired properties, based on CakeML's I/O model, about the interaction of the executable verifier with its hosting operating system.

Compilation: Instantiations of CakeML's compiler for generating machine executable verifiers happen in this module. For any deeply embedded \mathcal{V}_i^* , corresponding to $\hat{\mathcal{V}}_{dec}$ in STV_i , using proofs established in **DeepSpec**, we instantiate the compiler to synthesise executable verifier \mathcal{V}_i .

V. GENERIC STV FORMALISED

We proceed through three steps, namely specification, implementation and verification of implementations in HOL4 to formally verify the machine and its components.

Specification: Recall that we require minimising the trusted computing base for computing with generated verifiers. The specification module consists of properties which we can use to verify that our implementations of STV algorithms in HOL4 indeed match with the description of the counting methods used. In light of Definition 3 these properties actually form the semantics of the generic STV machine.

Implementation: Our final objective for creating the framework is to produce means for validating concrete items of evidence. To this end, we need decision procedures used for deciding whether or not given evidence is valid. Therefore we define boolean-valued functions in the implementation submodule to computationally realise logical declarations in the specification module. In particular, we implement decision procedures whose computational content provably realises the specification of the machine semantics.

	8/3
	2
	[a, b, c]
	[a, c]
hwin	[b ₄ , ([a, b, c], 1/9)]; a {3/1} b {10/9} c {11/9}; a {1} b {1} c {11b ₅ , ([c], 1/9), ([c, b], 1/9), ([c], 1/9)}]; [1; [a]; [c]
elim	[1; a {3/1} b {10/9} c {11/9}, a {1} b {11b ₄ , ([a, b, c], 1/9)}]; c {11b ₅ , ([a, c], 1/9), ([a, c, b], 1/9)}]; [1; [a]; [b, c]
count	[([a, c], 1/9), ([a, b, c], 1/9), ([a, c, b], 1/9); a {3/1} b {1/1} c {1/1}; a {1} b {11b ₄ }] c {11b ₅ }]]; [1; [a]; [b, c]
tr-elect	[1; a {3/1} b {1/1} c {1/1}; a {11b ₄ , ([a, c], 1/9), ([a, b, c], 1/9), ([a, c, b], 1/9)}]; b {11b ₄ }] c {11b ₅ }]]; [a]; [a]; [b, c]
elect	[1; a {3/1} b {1/1} c {1/1}; a {11b ₁ , b ₂ , b ₃ }] b {11b ₄ }] c {11b ₅ }]]; [1; [1; [a, b, c]
count	ba; a {0/1} b {0/1} c {0/1}; a [1] b [1] c [1]; [1; [a, b, c]

Fig. 1. An example of Evidence

Verification: Here we formally prove that the implementations meet the expectations of their respective specifications. Therefore through verification we come to eliminate a trusted layer required to lay in our framework.

We also have another motivation for developing the auxiliary module through this three-part process of specification, implementation and then verification. Familiarity, but no expertise, with our framework and its general purpose is needed for extending it to synthesise an executable verifier for one's desired STV scheme. On the one hand, an average user may lack skills in functional programming which we rely on for implementing verifiers in HOL4. However, the same user most probably has had exposure to formulations of mathematical properties in first-order logic syntax. Including purely descriptive logical assertions as our specifications, proven equivalent to their implementations, facilitates the users with means to understand the functionality of components, modules and the system as a whole simply by inspecting the specifications instead of implementations.

A. Data Structure of Machine States

We choose the data structure given in Definition 6 for implementing the abstract data type underlying the generic STV. We have four reasons for this choice of data structure.

- As the abstract syntactic representation of evidence closely represents concrete evidence, designing the parser and inspecting its correctness is less challenging.
- HOL4 has well-developed libraries containing verified assertions of operations on lists. By using lists to represent the data, both us and users of this framework can exploit the (already verified) tools to formalise the framework and avoid re-proving assertions from scratch.
- HOL4 also has well-developed tactics for discharging proof obligations on assertions involving list structure and operations on it. This notably eases the verification phase.
- Understanding the data structure used in the framework implementation is critical for ease of its usability and extensibility by third parties. The type *judgement* closely models concrete evidence such as the one in Figure 1. Therefore one can sensibly perceive the abstraction step taken for modelling concrete data which in turn enhances understandability of the framework and its mechanism.

Definition 6: We formalise machine states as an inductive type *judgement* whose constructors are *NonFinal* and *Final*. A *Final* (*w*) judgement declares *w* as winners of the election. A *NonFinal* (*ba*, *t*, *p*, *bl*₁, *bl*₂, *e*, *h*) judgement consists of uncounted ballots *ba*, tally *t*, pile *p*, *bl*₁ the backlog of elected candidates, *bl*₂ the backlog of eliminated candidates, and *e* and *h* for the list of elected and continuing (hopeful) candidates.

judgement =
 NonFinal (*ballots* × *tallies* × *piles* ×
cand list × *cand list* × *cand list* × *cand list*)
 | Final (*cand list*)

The types *ballots*, *tallies* and *piles* are respectively abbreviations for $((\text{cand list}) \times \text{rat})\text{list}$, $(\text{cand} \times \text{rat})\text{list}$ and

$(\text{cand} \times (\text{ballots list})\text{list})\text{list}$ where *rat* is the HOL4 type of fractional numbers.

Recall that objects whose type is *piles* serve as containers for recording the ballots allocated to each candidate. Our choice for the type of piles allows us to store the ballots received by each candidate in chunks of lists rather than one single list containing all of them. We have two reasons for designing the type of piles as it stands instead of $(\text{cand} \times (\text{ballots list})\text{list})\text{list}$;

(a): Some STV schemes such as the lower house ACT and Tasmania STV, employ a notion called "last parcel". In short, the last parcel of a candidate is the collection of those ballots received by the candidate at the last round of application of the count transition which caused the tally of the candidate to reach or exceed the quota and therefore be elected. Only the last parcel, instead of all ballots received, of an elected candidate is distributed, according to next preferences shown on the ballots in the last parcel, to the continuing candidates. Also they update the fractional transfer value, that each ballot in the last parcel awaiting transfer carries, based on the length of the last parcel. To accommodate this notion and its effect, we need identifying the ballots constituting the last parcel of an elected. We hence formalise the pile to record, for each continuing candidate, a list consisting of lists of ballots each received upon chronological applications of count transition. Once a candidate is elected their pile may be manipulated differently from the continuing candidates.

(b): We noted earlier that the STV used for the upper house elections in the Victoria state transfers votes of an eliminated candidate chunk by chunk in several applications of transfer-removed, rather than in a single action, in order of the magnitude of the fractional value that the chunks carry. We therefore need to rearrange the pile of ballots of an eliminated candidate into lists based on equality of the fractional value of ballots and then distribute them in the proper order as the scheme requires. This means that the type of piles has to be as it is defined above. Implementing piles in this way enables us to tailor the semantics of the transfer and elect transitions and their instantiations in such a way to modularly formalise several STV schemes which use the last parcel or stepwise distribution of votes from eliminated candidates or the combination of both.

There is also a reason for choosing the type *rat* instead of e.g. floating numbers. Elections with an STV counting scheme have a small margin of victory especially for the last vacancy left to fill [2]. This margin may happen to be less than the magnitude of error caused by the accumulation of rounding errors as it is with floating points. Hence one must sensibly avoid the high cost of electing a wrong candidate by falling into traps of imprecise calculations due to unintelligent choices. Using exact fractions allows safe handling of calculations.

B. The Semantics and its Auxiliary Components

The semantics of each machine transition label consists of conjunctions of formally specified general pre- and postconditions across STV schemes which enforce when and how to take a tallying action and what the immediate effect is. To

demonstrate the process of specification, implementation and verification of the machine transitions and their semantics, we discuss the transfer-elected transition.

STV schemes *explicitly* declare four conditions that must be satisfied for any legitimate application of transfer-elected;

- there are still vacancies to fill
- There are no uncounted ballots to deal with
- there is no vote from any eliminated candidate still awaiting distribution, and
- there are surplus votes of elected candidates to transfer.

Also there are some *implicit* conditions present in legal documents describing transfer-elected. These constraints come to attention either as the result of a straightforward understanding of the explicit conditions above or as auxiliary components that are taken for granted by legislators but are necessary for proper functioning of the explicit constraints;

- every candidate in the pre- and post-state of transfer-elected has a unique tally and pile
- no one is elected or eliminated by applying transfer-elect
- no candidate attracts any new vote by transfer-elect and therefore tallies remain the same
- candidates whose names appear in the backlog of elected are indeed among elected candidates
- any elected candidate is no longer a continuing candidate so that they do not receive votes any further
- the list of competing candidates in the election is not empty and has no duplication of names

Conjunctions of formal declarations of the above explicit and implicit conditions form the semantics of the transition label transfer-elected. The predicate TransferAuxSpec defines the semantics of this transition.

$$\begin{aligned} \text{TransferAuxSpec } (qu, st, l) \text{ } ba \text{ } t \text{ } t' \text{ } p \text{ } p' \text{ } bl \text{ } bl_2 \text{ } bl'_2 \text{ } e \text{ } e' \text{ } h \text{ } h' \iff \\ bl_2 = [] \wedge bl \neq [] \wedge ba = [] \wedge bl'_2 = [] \wedge t' = t \wedge e' = e \wedge \\ h' = h \wedge \text{length } e < st \wedge (\forall d. \text{mem } d (h \# e) \Rightarrow \text{mem } d l) \wedge \\ (\forall d. \text{mem } d bl \Rightarrow \text{mem } d l) \wedge \text{distinct } (h \# e) \wedge \\ \text{Valid_PileTally } t \text{ } l \wedge \text{Valid_PileTally } p \text{ } l \wedge \text{Valid_PileTally } p' \text{ } l \wedge \\ \text{ValidInitCandList } l \wedge \text{distinct } (\text{map fst } t) \wedge \text{distinct } (\text{map fst } p) \end{aligned}$$

Transfer-elected, similar to other formalised transitions, is parametrised by the quota qu , the number of initial vacancies st and the list l of all candidates competing in the election. The semantics of the transition declares that given the ballots ba , tally t , pile p , backlog of elected bl , backlog of eliminated bl_2 and the list of elected e and continuing candidates h in the pre-state of transfer-elected, and their respective counterparts in the post-state characterised by having a prime symbol, some conditions as specified above are satisfied by the transition. Note that mem in TransferAuxSpec declares list membership.

$$\text{ValidInitCandList } l \iff l \neq [] \wedge \text{distinct } l$$

The predicate ValidInitCandList which realises item (6) asserts that each candidate has a pile and a tally. We also declare that elements in the first component of the tally t and pile p are distinct. Therefore we come to satisfy item (1).

$$\text{Valid_PileTally } t \text{ } l \iff \forall c. \text{mem } c \text{ } l \iff \text{mem } c (\text{map fst } t)$$

We implement for each of the above declarations a counterpart decision procedure that is later translated and then extracted

as part of the verifier for actual computation. To illustrate how this phase proceeds, we provide some instances. The following two functions together implement Valid_PileTally.

$$\begin{aligned} \text{Valid_PileTally_dec1 } [] \text{ } l &\iff \text{true} \\ \text{Valid_PileTally_dec1 } (h::t) \text{ } l &\iff \\ \text{mem } (\text{fst } h) \text{ } l \wedge \text{Valid_PileTally_dec1 } t \text{ } l \end{aligned}$$

$$\begin{aligned} \text{Valid_PileTally_dec2 } t \text{ } [] &\iff \text{true} \\ \text{Valid_PileTally_dec2 } t \text{ } (l_0::ls) &\iff \\ \text{if mem } l_0 (\text{map fst } t) \text{ then Valid_PileTally_dec2 } t \text{ } ls \text{ else false} \end{aligned}$$

We prove that the implementations and specification match:

$$\begin{aligned} \text{Valid_PileTally } t \text{ } l &\iff \\ \text{Valid_PileTally_dec1 } t \text{ } l \wedge \text{Valid_PileTally_dec2 } t \text{ } l \end{aligned}$$

Conjunctions of the computational implementations define a computational twin TransferAuxDec for the specification of the transfer-elected semantics TransferAuxSpec.

$$\begin{aligned} \text{TransferAuxDec } (qu, st, l) \text{ } ba \text{ } t \text{ } t' \text{ } p \text{ } p' \text{ } bl \text{ } bl_2 \text{ } bl'_2 \text{ } e \text{ } e' \text{ } h \text{ } h' &\iff \\ \text{null } bl_2 \wedge e' = e \wedge h' = h \wedge t' = t \wedge \text{length } e < st \wedge \\ \text{list_MEM_dec } (h \# e) \text{ } l \wedge \text{list_MEM_dec } bl \text{ } l \wedge \\ \text{distinct } (h \# e) \wedge \text{Valid_PileTally_dec1 } t \text{ } l \wedge \\ \text{Valid_PileTally_dec2 } t \text{ } l \wedge \text{Valid_PileTally_dec1 } p \text{ } l \wedge \\ \text{Valid_PileTally_dec2 } p \text{ } l \wedge \text{Valid_PileTally_dec1 } p' \text{ } l \wedge \\ \text{Valid_PileTally_dec2 } p' \text{ } l \wedge \neg \text{null } l \wedge \text{distinct } l \wedge \neg \text{null } bl'_2 \\ \text{distinct } (\text{map fst } t) \wedge \text{distinct } (\text{map fst } p) \wedge \neg \text{null } bl \wedge \text{null } ba \end{aligned}$$

Similarly we obtain specification and computational implementation for other machine transitions as well. Then the specification (implementation) of the machine semantics is comprised of the collection of specification (resp. implementation) of the individual transitions. We formally prove each transition implementation matches with its specification.

Theorem 1: Assume $M^{\text{spec}} = \langle \mathcal{S}, \mathcal{T}, (S_t^{\text{spec}})_{t \in \mathcal{T}} \rangle$ is the specification of the machine and $M^{\text{dec}} = \langle \mathcal{S}, \mathcal{T}, (S_t^{\text{dec}})_{t \in \mathcal{T}} \rangle$ is its computational implementation. Then for any $t \in \mathcal{T}$, $S_t^{\text{spec}} \iff S_t^{\text{dec}}$.

One can already proceed to synthesise an executable verifier from the underspecified generic machine. Such a verifier can correctly decide if given evidence ω claimed to have been produced by an algorithm of the STV family, instead of e.g. First-Past-The-Post voting [9] where voters choose whom they prefer most and the candidate with the most amount of votes received is elected. However we wish to generate verifiers that can recognise and validate according to which specific instances of STV ω has been output. Hence we need to enrich the computational content of the machine semantics with more pre- and postconditions which are particular to individual STV schemes. We refer to this enrichment process as instantiation of the machine and discuss it further below.

VI. INSTANTIATIONS OF THE MACHINE

We exemplify how an instantiation of the transfer-elected succeeds for transferring surplus of elected candidates based on the ACT STV. Instantiation of other machine transitions proceeds in a similar manner. Under the section 'Step 3' and 'transfer surplus from elected candidates' the protocol explains

under what conditions and how to distribute surplus votes. We summarise and rephrase these sections as follows.

- ₁ no candidate exceeds the quota
- ₂ the parcel of an elected with surplus is not empty.
- ₃ the distribution of the surplus of an elected candidate proceeds in one single step.
- ₄ the surplus of elected candidates is distributed one at a time beginning with those who are elected earlier.
- ₅ the pile of the candidate whose surplus is transferred is emptied in the post-state of transfer-elected.
- ₆ only the last parcel of votes received (which resulted in a surplus) is transferred. It may be that the last parcel is the only parcel in a candidate's pile (if only one application of the count action has occurred), or more parcels exist (if several actions of count have happened as a result of an earlier elect, transfer or eliminate actions)
- ₇ the pile of any candidate other than the one whose surplus is transferred at this stage remains the same.
- ₈ the fractional transfer value is subsequently computed depending on whether or not the last parcel is the only parcel of ballots in the pile of the elected candidate.

$\text{TransferActSpec } (qu, st, l) \ j_1 \ j_2 \iff$
 $\exists ba \ ba' \ t \ t' \ p \ p' \ bl \ bl' \ bl_2 \ bl'_2 \ e \ e' \ h \ h'.$
 $j_1 = \text{NonFinal } (ba, t, p, bl, bl_2, e, h) \wedge$
 $\text{TransferAuxSpec } (qu, st, l) \ ba \ t \ t' \ p \ p' \ bl \ bl_2 \ bl'_2 \ e \ e' \ h \ h' \wedge$
 $(\forall c'. \text{mem } c' \ h \Rightarrow \exists x. \text{mem } (c', x) \ t \wedge x < qu) \wedge$
 $\exists l_1 \ c. (bl = c::l_1 \wedge \text{mem } c \ l \wedge (\forall l_2. \text{mem } (c, l_2) \ p \Rightarrow l_2 \neq []))$
 $\wedge bl' = l_1 \wedge ba' = \text{last } (\text{getCandPile } c \ p) \wedge \text{mem } (c, []) \ p' \wedge$
 $\star \forall d. d \neq c \Rightarrow \forall l_3. (\text{mem } (d, l_3) \ p \Rightarrow \text{mem } (d, l_3) \ p') \wedge$
 $\star (\text{mem } (d, l_3) \ p' \Rightarrow \text{mem } (d, l_3) \ p) \wedge$
 $j_2 = \text{NonFinal } (nba, t', p', bl', bl'_2, e', h')$

We augment the formal counterparts of the •_i conditions to the clauses given in Section V-B to obtain the specification TransferActSpec for transfer-elected of ACT STV. The lines denoted by the star symbol (★) in TransferActSpec formalise the Clause •₇ above. Also note that we place the last parcel of the pile of the candidate c whose votes are transferred first into the list of uncounted ballots so that in the subsequent transition count deals with distributing the votes and “recalculating” candidates’ tallies and piles. Finally, because of efficiency purposes our system is designed to update the fractional transfer value of the surplus in the formalisation of the elect transition instead of the transfer-elected transition.

We next define computational twins for the components of TransferActSpec and use them to implement a computational counterpart for the semantics of the ACT transfer-elected. For example, the function getCandTally looks through a tally list t and finds the tally of an input candidate name c . We then verify this function computes the tally of candidates correctly and that indeed every candidate is assigned only one tally (item 1 of the implicit machine conditions).

$\text{distinct } (\text{map fst } t) \wedge \text{mem } (c, x) \ t \Rightarrow \text{getCandTally } c \ t = x$

Using this function, we implement another function lessThanQuota which checks if the tally of every candidate in a given list ls is below the quota (Item •₁).

$\text{lessThanQuota } qu \ l \ ls \iff \text{every } (\lambda h. \text{getCandTally } h \ l < qu) \ ls$ ¹see https://github.com/MiladKetabGhale/Modular_Checker

We show that the function lessThanQuota is a computational realisation of its specification:

$(\forall c. \text{mem } c \ h \Rightarrow \exists x. \text{mem } (c, x) \ t \wedge x < qu) \wedge \text{distinct } (\text{map fst } t) \wedge$
 $(\forall c''. \text{mem } c'' \ h \Rightarrow \text{mem } c'' \ (\text{map fst } t)) \Rightarrow \text{lessThanQuota } qu \ t \ h$

Moreover lessThanQuota enforces its specification.

$\text{lessThanQuota } qu \ (t_0::t_1) \ h \wedge \text{Valid_PileTally_dec2 } (t_0::t_1) \ h \Rightarrow$
 $\forall c. \text{mem } c \ h \Rightarrow \exists x. \text{mem } (c, x) \ (t_0::t_1) \wedge x < qu$

In the same manner we define and verify other functions that computationally implement the rest of components of the transfer-elected specification. Conjunctions of the implementations constitute a computational semantics for transfer-elected.

$\text{TransferActDec } (qu, st, l) \ (\text{NonFinal } (ba, t, p, bl, bl_2, e, h))$
 $(\text{NonFinal } (ba', t', p', bl', bl'_2, e', h')) \iff$
 $\text{TransferAuxDec } (qu, st, l) \ ba \ t \ t' \ p \ p' \ bl \ bl_2 \ bl'_2 \ e \ e' \ h \ h' \wedge$
 $\text{lessThanQuota } qu \ t \ h \wedge$
 $\text{dtcase } bl \text{ of}$
 $[\] \Rightarrow \text{false}$
 $| \ hbl::tbl \Rightarrow (\text{let } gcp = \text{getCandPile } hbl \ p \text{ in}$
 $\neg \text{null } gcp \wedge \text{mem } hbl \ l \wedge bl' = tbl \wedge ba' = \text{last } gcp \wedge$
 $\text{mem } (hbl, []) \ p' \wedge \text{subpile1 } hbl \ p \ p' \wedge \text{subpile2 } hbl \ p' \ p)$
 $\text{TransferActDec } v_0 \ (\text{Final } v_1) \ v_2 \iff \text{false}$
 $\text{TransferActDec } v_3 \ (\text{NonFinal } v_4) \ (\text{Final } v_5) \iff \text{false}$

The functions subpile1 and subpile2 together implement the lines marked by the star symbol in Definition of the TransferActSpec . We next demonstrate how the framework modularly extends to instantiations with various STV algorithms.

A. Variations in Instantiation

To exemplify the flexibility of our framework, we discuss an instantiation of the machine with the CADE STV as well. We refer the reader to our source code¹ for details of five other STV instances for which we have successfully completed the verifiers’ synthesis process. As we have already illustrated how a specification and the corresponding implementation of an algorithm advance, we therefore only elaborate on how CADE STV varies from ACT STV in textual descriptions of its semantics and subsequently its implementations.

CADE STV is used for electing the board of trustees of the Conference on Automated Deduction. The algorithm is radically different from “standard” STV algorithms. In particular, to the best of our knowledge, every “normal” STV algorithm at least respects Clause •₁. However CADE violates not only this condition but also all of the ACT’s transfer-elected semantics components except Clause •₂. This unorthodox behaviour of CADE permeates to the semantics of other transitions as well to an extent where the algorithm is sometimes questioned to be a true member of the STV family. But our framework flexibly accommodates even the extraordinary ones. We catalogue CADE’s transfer-elected informal semantics conditions as follows.

- ★₁ the backlog of elected candidates contains one element.
- ★₂ parcel of the element in the backlog of elected candidates is not empty.

- *₃ backlog of the elected candidates is emptied in the post-state of transfer-elected.
- *₄ the election restarts after each round of transfer-elected.

The Clause *₄ itself consists of the following sub-clauses.

- *_{4a} pile of all of the candidates is emptied in the post-state.
- *_{4b} all ballots in the piles of all candidates are placed back into the list of uncounted ballots with the name of already elected candidate removed from those ballots.
- *_{4c} the eliminated candidates are “resurrected” meaning they start to be continuing candidates in the post-state (implemented by the ListDiff function below).

```

TransferCadeDec (qu,st,l) (NonFinal (ba,t,p,bl,bl2,e,h))
  (NonFinal (ba',t',p',bl',bl'2,e',h'))  $\iff$ 
  TransferAuxDec (qu,st,l) ba t t' p p' bl bl2 bl'2 e e' h h' l
  dtcase bl of
  []  $\Rightarrow$  false
  | hbl::tbl  $\Rightarrow$  tbl = []  $\wedge$  bl' = []  $\wedge$   $\neg$ null (getCandPile hbl p)  $\wedge$ 
    ba' = APPEND_ALL p  $\wedge$  ALL_EMPTY l p'  $\wedge$  ListDiff e' h' l
TransferCadeDec v0 (Final v1) v2  $\iff$  false
TransferCadeDec v3 (NonFinal v9) (Final v5)  $\iff$  false

```

Remark 1: In this section, when we instantiated transitions of the machine, particularly transfer-elected, we named instantiations differently. For example, we represented the instantiation of transfer-elected with the ACT STV by TransferActDec. This choice was out of pedagogical purposes to assist the reader with understanding the work. This is different in the actual proof engineering development. We carry out different instantiations of the machine in separate modules but uniformly name instantiations of the transitions. For example, for instantiation of the machine with the ACT STV, we have a module accordingly named and inside the module we refer, for instance, to the instantiation of transfer-elected as TransferDec or TransferSpec without the infix Act.

Remark 2: The Core calculus of HOL4 uses *term rewriting* to manipulate assertions expressed in higher-order logic and ML programming style. Since HOL4 is a rewriting system, what appears as the name of an assertion on the left of \iff is therefore secondary to its definitional content on the right side. In light of the previous remark, we can uniformly refer to *names* of instantiated transitions regardless of the algorithm used. Therefore we can formulate evidence verifier based on the names of the transitions refer to the desired instantiation module to equip the names with the semantics of the algorithm for which we are constructing a verifier. Consequently the Verifier, Translation, DeepSpec and Compilation modules in Figure 2 which all depend on instantiation modules as their parents are developed once and for all.

B. Automating Verification of Instantiations

Once an instantiation of the machine completes we verify logical equivalence of the specification of each transition semantics with its implementation. Based on our experience, proving such equivalence roughly requires 200 lines of HOL4 code. We desire to automate them in a way that they practically approximate the following desideratum.

Desideratum 1: Assume \mathcal{A} is an arbitrary STV algorithm, $\hat{\mathcal{A}}_{spec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{spec})_{t \in \mathcal{T}} \rangle$ is a specification of \mathcal{A} ’s instantiation into the machine and $\hat{\mathcal{A}}_{dec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{dec})_{t \in \mathcal{T}} \rangle$ is its implementation. Then for any $t \in \mathcal{T}$ the framework automatically proves $S_t^{dec} \iff S_t^{spec}$.

We engineer the framework in a way that the proofs module calls instantiation modules as its parents one by one. Considering Remark 1, we uniformly declare the desired equivalence between the specification and implementation of an instantiated transition and discharge the proof in the module.

Engineered this way, the desideratum is met under the following two scenarios for the proofs module. However for a third scenario, proofs in the module may break so that proving the equivalences becomes a semi-automatic interactive procedure. The proofs nonetheless remain reusable after proper refinements so that one needs not re-encoding 200 lines.

Scenario one. We have already formalised and verified instantiations of the machine with five different STV algorithms. If the algorithm \mathcal{A} already exists in our framework then the desideratum is satisfied. No effort beyond following simple instructions to execute on the command line is required to synthesise an executable verifier.

Scenario two. Another possibility is that \mathcal{A} as a whole does not literally match with any of the already existing instantiations of the machine. However, clauses describing pre- and post-conditions of transitions which are components of the semantics of \mathcal{A} do exist in the auxiliary module. Then all one needs doing is to call the formal specifications of the clauses and their respective implementations into the specification and implementation of transitions, respectively, to formally obtain an instantiation of the algorithm. In this case, the proofs also succeed in satisfying the desideratum.

Scenario three. Suppose there is a clause in the description of \mathcal{A} whose specification, and therefore implementation, does not exist in the auxiliary module. Then one trivially has to extend the auxiliary module by specifying and implementing that clause and then verifying the implementation correct against the specification. If one was overconfident in their implementation, then they can take the implementation as its specification in which case no verification is required. Note that as including the generic formal machine transitions in the semantics of each transition instantiation is mandatory there are few numbers of such clauses and consequently few lines of encoding needed to formalise them.

Once the proofs of equivalence between the specification and implementation of a machine instantiation either automatically or interactively succeed, the rest of verifier synthesis process for the instantiated algorithm completely automatically follows to eventually obtain an executable verifier.

VII. MODULAR SYNTHESIS OF VERIFIERS

According to Definition 5 the notion of verifier depends on instantiation of the machine with an algorithm \mathcal{A} . In view of Remark 1, we only need to develop the verifier module once and simply vary the parent instantiation module to adapt the verifier definition for a different instantiation. So let’s assume

$\hat{A}_{spec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{spec})_{t \in \mathcal{T}} \rangle$ and $\hat{A}_{dec} = \langle \mathcal{S}, \mathcal{T}, (S_t^{dec})_{t \in \mathcal{T}} \rangle$ are the specification and implementation of an STV algorithm \mathcal{A} where for any $t \in \mathcal{T}$, $S_t^{dec} \Leftrightarrow S_t^{spec}$. Then we define \mathcal{V}_{spec} as the specification of the verifier and implement \mathcal{V}_{dec} as its computational counterpart. Drawing on the match between S_t^{dec} and S_t^{spec} for each t , we formally prove Theorem 2.

Theorem 2: Suppose the algorithm \mathcal{A} is instantiated in the machine as above. Then for any piece of evidence $\hat{\omega} = (Q, s, C, \Omega)$ produced by an execution of \mathcal{A}_{dec} , $\mathcal{V}_{spec}(Q, s, C) \Omega \Leftrightarrow \mathcal{V}_{dec}(Q, s, C) \Omega$.

As the specification and implementation are equivalent, we shall only discuss the formalisation of the implementation. We first define the decision procedure `Valid_Step` which for given two machine states j_0 and j_1 decides if a transition from the former to the latter occurs by an application of a transition.

```
Valid_Step params j0 j1  $\Leftarrow$ 
  HwinDec params j0 j1  $\vee$  EwinDec params j0 j1  $\vee$ 
  CountDec params j0 j1  $\vee$  TransferDec params j0 j1  $\vee$ 
  ElectDec params j0 j1  $\vee$  TransferExcludedDec params j0 j1  $\vee$ 
  EXISTS ( $\lambda c$ . ElimCandDec c params j0 j1) (snd (snd params))
```

Then we implement the function `valid_judgements_dec` which recursively calls `Valid_Step` on a list of machine states.

```
valid_judgements_dec v0 []  $\Leftarrow$  false
valid_judgements_dec v1 [Final v2]  $\Leftarrow$  true
valid_judgements_dec v3 [NonFinal v10]  $\Leftarrow$  false
valid_judgements_dec params (j0::j1::js)  $\Leftarrow$ 
  Valid_Step params j0 j1  $\wedge$  valid_judgements_dec params (j1::js)
```

The verifier in HOL4 is eventually defined as a follows. The function `initial_Judgement_dec` decides if the first element of input evidence is a machine state correctly recording data of the initial state of tallying votes where no one has yet attracted any vote, the backlogs of elected and eliminated and the list of elected are all empty and every candidate is continuing.

```
Check_Parsed_Certificate params []  $\Leftarrow$  false
Check_Parsed_Certificate params (fst_state::rest_states)  $\Leftarrow$ 
  Initial_Judgement_dec (snd (snd params)) fst_state  $\wedge$ 
  valid_judgements_dec params (fst_state::rest_states)
```

One can use `Check_Parsed_Certificate` to run small sample elections manually encoded in HOL4's environment. However real evidence such as Figure 1 are stored in a file in an operating system which needs to be read, parsed, and processed for validation. Therefore we need an executable verifier in an operating system's environment. On the other hand, `Check_Parsed_Certificate` despite its infeasibility for computation is proven to behave correctly as the specification \mathcal{V}_{spec} expects. How can we synthesise from it an executable verifier \mathcal{V} that is both efficient for actual computation and provably correct with respect to \mathcal{V}_{spec} and thus trustworthy? To this end, we invoke the verified CakeML's proof translator tool [14], its ecosystem and the compiler [21].

Using the verified CakeML's translator we obtain a translated version of the verifier \mathcal{V}_τ . Thanks to the translator, we are guaranteed that every property proven for `Check_Parsed_Certificate` and its components also holds for \mathcal{V}_τ and its components. Therefore correctness of the verifier in

Electorate	Ballots	Candidates	Seats	year
Brindabella	63562	20	5	2012
Ginnindra	66076	27	5	2012
Molonglo	88266	40	7	2008

Fig. 3. Parameters of the ACT Lower house elections 2008/2012

HOL4 provably extends to that of \mathcal{V}_τ . The translated verifier \mathcal{V}_τ is a pure function operating in CakeML's environment. To synthesise an executable verifier that actually opens files, parses evidence lines and validates them according to \mathcal{V}_{spec} , we implement a deeply embedded function called `check_count` in CakeML's ecosystem. CakeML has libraries developed for modelling and verifying properties about I/O semantics of the executable versions of a deeply embedded impure function. Using this modelling we specify and prove that the specified and compiled `check_count` behaves as follows.

The function `check_count` accepts a file as input on the command line, opens the file consisting of evidence which we intend to validate, parses the header of the evidence consisting of the quota, number of seats and competing candidates in the election. If the header is well-formed, then `check_count` proceeds to parse two judgement lines at a time and if the lines successfully parse into values of the type judgement (the type of the machine states) then checks if the transition happening between them *corresponds with the specification of the verifier in HOL4* (\mathcal{V}_{spec}). This process of reading evidence lines, parsing and checking them continues until either the evidence is accepted as valid or `check_count` encounters a malformed judgement line or an invalid transition step from one parsed judgement (machine state) to another in which case it returns an error message informing us where it occurs.

VIII. EXPERIMENTAL RESULTS

Figure 3 illustrates the lower house election electorates of the Australian Capital Territory with the size of their respective input valid ballots and parameters (seats and candidates) for the elections held in years 2008 and 2012. Figure 4 shows some of the experimental results performed on real historical data of these districts. For each election, we construct evidence by executing a Haskell program² computing winners of each election according to ACT STV. Then the verifier synthesised for validating instances of computation with the ACT STV verifies each evidence as valid³. Note that certifying evidence validity is costlier than detecting the invalidity of evidence of the same size, because the former takes parsing *all* lines and verifying *every* transition while the latter does not. Contrary

Electorate	Evidence size (mb)	Validation time (sec)	year
Brindabella	57.5	1627	2012
Ginnindra	72.7	1789	2012
Molonglo	195.6	12063	2008

Fig. 4. Evidence Validation for the ACT Lower house elections 2008/2012

to the folklore that theorem provers are basically meant for verification, rather than efficient computation, the generated

²Source code at https://github.com/MiladKetabGhale/Modular_Checker

³Using one processor of an Intel Core i7-7500U CPU 2.70 GHz \times 4

verifier for ACT STV shows good performance. For example, Molonglo electorate is the biggest electoral district among the lower elections in terms of the constituency magnitude and vacancies. Despite costly computation for meticulously examining correctness of every small detail in the evidence, the verifier validates it in about three hours. Considering the typical time lapse in publicly announcing real election results, this performance is acceptable.

IX. RELATED WORK

Use of theorem proving to verify vote counting is an emerging field. The only use of theorem proving and certifying algorithms [16] that we are aware of is our own previous work [12] which just verifies a single instance of STV. While the overall approach is similar, the underlying data structure and the algorithmic content which is being formalised does not allow modular specification, verification and synthesis of verifiers for different types of STV: only a single STV algorithm is being verified. Adapting to other schemes requires a considerable effort of re-encoding (sometimes the whole data structure) and non-trivial expertise with HOL4 and CakeML.

Other approaches focus on the formalisation and verification of algorithms for correctly computing election results, rather than checking correctness of computation performed. This is usually based on so-called heavy-weight formal methods (interactive theorem proving) and uses a variety of tools.

There is work which formally verifies some STV algorithms in the theorem prover Coq. Ghale et al [10] formalise the student union STV algorithm used in some union elections Australia. Pattinson and Schürmann [17] verify the “vanilla” STV algorithm. Pattinson and Tiwari [18] prove the Schultz voting scheme correct, and Verity and Pattinson [20] verify an invariant of STV. Finally, Ghale et al [11] provide a modular treatment of formalisation and verification of STV algorithms. They all then employ the automatic program extraction tool of Coq to obtain Haskell or OCaml programs to provably correctly compute election results based on the algorithm for any real input data. These programs output verifiable evidence upon each execution. Some of them [17], [18] implement a verifier for checking the correctness of execution instances. However, the verifier is not verified. They implement their verifier as Haskell programs without proving that the verifier itself computes correctly. Moreover, even trusting that the verifier is reliable, there remain other unverified layers to trust in. For example, they rely on the GHC compiler to execute the program. As GHC is not a verified compiler one has to trust that the machine code executed also behaves correctly.

Others employ other theorem proving means such Delphin [6] or HOL4. Schürmann and DeYoung [7] have pioneered the use of theorem proving in vote counting. They use Linear Logic [13] to specify the semantics of a specific STV which is similar to Victoria STV except that it transfers votes of an eliminated candidate at once. Although the program obtained from the formalisation performs decently, reading their encoding is challenging due to the complexity of Linear Logic syntax. As the expressive power of Linear Logic is

limited compared to higher-order logic, their definition of the count action, for example, uses eight auxiliary assertions which further complicates the encoding and therefore the adaptability and usability of this approach. Dawson et al [5] formalise Hare-Clark STV of Tasmania state in the theorem prover HOL4 and verify properties about the algorithm. They however manually transliterate their HOL4 implementation into an ML program. There are several issues with this approach. For example, manual code translation is not a verified process. Also the semantics of the source (HOL4) and the target (ML) differ. Hence verification established for the HOL4 program does not provably extend to the transliterated one.

X. CONCLUSION

We introduced a framework to synthesise provably trustworthy tools for transparent independent verification of computation carried out for computing winners of election which rely STV vote counting algorithms. Our work witnesses usability of formal methods, in particular theorem proving, as means for building software that is both correct and efficient.

REFERENCES

- [1] ACT Electoral Commission, www.elections.act.gov.au/elections_and_voting
- [2] M.L. Blom, P. J. Stuckey, V. J. Teague, Computing the Margin of Victory in Preferential Parliamentary Elections, E-Vote-ID, 2018, pp.1–16
- [3] CADE: The Single Transferable Vote, <http://www.cadeinc.org/bylaws>
- [4] CakeML: A Verified Implementation of ML, <https://cakeml.org/>
- [5] J.E. Dawson, R. Goré, T. Meumann, Machine-checked Reasoning about Complex Voting Schemes Using Higher-order Logic, EVote-ID 2015, pp.142–158
- [6] The Delphine Project, <http://delphin.poswolsky.com/home>
- [7] H. DeYoung, C. Schürmann, Linear Logical Voting Protocols, VoteID 2011, LNCS Springer, pp. 53–70
- [8] H.R. Droop, On Methods of Electing Representatives. J. of the Statistical Society of London 44(2), 1881, pp.141–202
- [9] D.M. Farrell, Electoral Systems: A Comparative Introduction, 2nd ed., New York: Palgrave Macmillan, 2011
- [10] M.K. Ghale, R. Goré, D. Pattinson, A Formally Verified Single Transferable Voting Scheme with Fractional Values, E-Vote-ID 2017, pp.163–182
- [11] M.K. Ghale, R. Goré, D. Pattinson, M. Tiwari, Modular Formalisation and Verification of STV Algorithms, E-Vote-ID 2018, pp.51–66
- [12] M.K. Ghale, D. Pattinson, R. Kumar, M. Norrish, Verified Certificate Checking for Counting Votes, VSTTE 2018, pp.69–87
- [13] J. Girard, On the Unity of Logic. Ann. Pure Appl. Logic 59(3), 1993, pp.201–217
- [14] S. Ho, O. Abrahamsson, R. Kumar, M. O. Myreen, Y. K. Tan, Michael Norrish, Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions, Proc. 9th Conf. IJCAR 2018, pp. 646–662
- [15] HOL Interactive Theorem Prover, <https://hol-theorem-prover.org/>
- [16] R.M. McConnell, K. Mehlhorn, S. Näher, P. Schweitzer, Survey: Certifying Algorithms, J. Comput. Sci. Rev., vol5(2), 2011, pp.119–161
- [17] D. Pattinson, C. Schürmann, Vote Counting as Mathematical Proof, 28th Australasian Joint Conference 2015, pp.464–475
- [18] D. Pattinson, M. Tiwari, Schulze Voting as Evidence Carrying Computation, ITP 2017, pp.410–426
- [19] The Parliament of Victoria: Electoral Act 2002
- [20] F. Verity, D. Pattinson, Formally Verified Invariants of Vote Counting Schemes, ACSW 2017, pp.1–10
- [21] Tan K. Yong, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, M. Norrish, A New Verified Compiler Backend for CakeML, Proc. 21st ACM SIGPLAN, ICFP 2016, pp.60–73