

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—
Index Terms—

I. INTRODUCTION

II. EVIDENCES

We are motivated to engineer a trustworthy practical framework to answer the following question. How can one verify that any execution of any implementation \mathcal{P} whose source code is secret of an arbitrary known STV algorithm \mathcal{A} correctly computes the end result according to instructions in \mathcal{A} ?

To begin with addressing the problem, assume \mathcal{P} is an implementation of an algorithm \mathcal{A} . We reasonably demand each execution of \mathcal{P} on a given input x consisting of ballots recorded in the election to output the winners y and evidence ω standing as a 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 to how one can verify the evidence of any *instance of computation* with any algorithm \mathcal{A} .

An evidence as such must be enough informative to offer 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 instructions of the counting scheme dictate so that winners truly reflect will of the voters. Therefore one wonders what information is enough to serve our end and what data structure for recording the information in evidence we should choose.

A. Characterisation of Evidence

A thorough analysis of different STV algorithms reveals existence of an underlying abstract data type common between STV algorithms. Each STV scheme operates on some ballots, each of which is a pair consisting of a preference list and a fractional value, recorded to be counted and has a quota which is the least amount of votes needed to attract in order to be elected, some candidates competing in the election, some vacancies for which candidates compete, and more importantly *discrete states of computation* which encapsulate necessary pieces of information needed to transparently know how the tallying has progressed from the beginning to the 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 terminal states where tallying reaches to a halting state by finding the winners of the election. The other set of states consist of non-final ones where bits and pieces of data are manipulated according to counting mechanism of the STV scheme used, to eventually obtain a terminal result. Each non-final state encloses 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 votes that each candidate have attracted,
- **backlog of elected** candidates who have exceeded the quota and their surplus votes awaits being transferred,
- **backlog of eliminated** ones whose votes awaits transfer,
- **elected candidates** who are the already elected ones whose surplus votes must be transferred,
- **continuing candidates** hopefully running in the election.

The non-final and final states specified above provide us with all 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 a piece of 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 FINITE STATE MACHINE

We have already found out 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 us with designing and developing the framework modularly while offering automation of synthesising process of verifiers.

In a secondary examination of the STV family, we also identify a universal algorithmic mechanism for performing

- **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.

Fig. 1. Generic STV Counting Steps

computation on evidence. This generic method comprises seven general actions (Figure 1) corresponding to actions that tally officers take when counting votes. Each of the actions is constituted of universal pre-conditions on when and how the action comes into effect, and ubiquitous post-conditions dictating what effect applying the action brings about by describing the new state to which computation proceeds. In fact, these conditions are the formal counterparts of the legal clauses in the textual specification of the counting scheme which tell tally officers how to proceed with tallying.

For example a constant pre-condition for applying elimination asserts that there must be no uncounted ballot left and another one declares that no candidate should be electable at the current state of counting. A post-condition 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 comprise what an action is and what an application of that action means.

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

Definition 2 (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 *generic STV* is the triple $M = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$.

A. Instantiations of the Machine with Specific STV schemes

There are nonetheless variations in the precise counting method of different STV algorithms as well. For example in the STV used for electing the upper house representatives of the Victoria state of Australia, 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 pre-condition 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 Tasmania state of Australia does not have such pre-condition simply because votes of an eliminated candidate are transferred in a single application of transfer-removed. There are also variations in post-conditions. For example transfer-elect of the STV of ACT requires distributing only the last parcel of surplus votes received by an elected candidate. But the scheme of the Victoria state specifies distributing all surplus votes of an elected candidate.

We formally realise variations of STV algorithms by instantiation (Definition 3) of the generic machine. Recall that the semantics of the machine is formed by conjunctions of formally specified universally appearing pre and post-conditions 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 3 (Instantiation of the Machine): Assume the generic machine $M = \langle \mathcal{S}, \mathcal{T}, (S_t)_{t \in \mathcal{T}} \rangle$ as in Definition 2. An instantiation $\hat{\mathcal{A}}$ for an STV algorithm \mathcal{A} into 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 post-condition specific to \mathcal{A} .

The operational semantics of each instantiation allows us performing operations such validation on evidences. Definition 4 lays down what verifying an evidence amount to. Informally speaking, to check if an 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 4 (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 with \mathcal{A} is a function \hat{V} mapping from $\mathbb{Q} \times \mathbb{N} \times 2^C \times \text{List}(\mathcal{S})$ to $\{0, 1\}$ such that for any evidence $\hat{\omega} = (Q, s, C, \Omega)$, $\hat{V}(\hat{\omega}) = 1$ if and only if the following holds: $\forall i \in \{1, \dots, |\Omega| - 1\}$.

$$\forall \Omega_{i-1} \Omega_i \in \mathcal{S}. \exists \Omega_1 \Omega_2 \in \text{List}(\mathcal{S}). \Omega = \Omega_1 + [\Omega_{i-1}; \Omega_i] + \Omega_2 \\ \Rightarrow \exists t \in \mathcal{T}. (\Omega_{i-1}, \Omega_i) \in t \wedge \vdash S'_t[\Omega_{i-1}, \Omega_i]$$

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. 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 of an evidence to concretely illustrate them and exemplify how counting with an STV scheme works.

B. An Example of Evidence

Figure 2 depicts 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, and ballots counted in favour of particular candidates are placed onto that candidate's pile. Here, a is the first preference on b_1 , b_2 , and b_3 , and 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, 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. b and c each attracts one vote. However neither of them reaches to the quota. Therefore the weakest one b is eliminated. As the number of elected and continuing candidates does not exceed the initial vacancies, the election terminates by declaring a and c 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 assertions 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 3 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 formal specification of the machine and its components, implementation of functions meant to be the com-

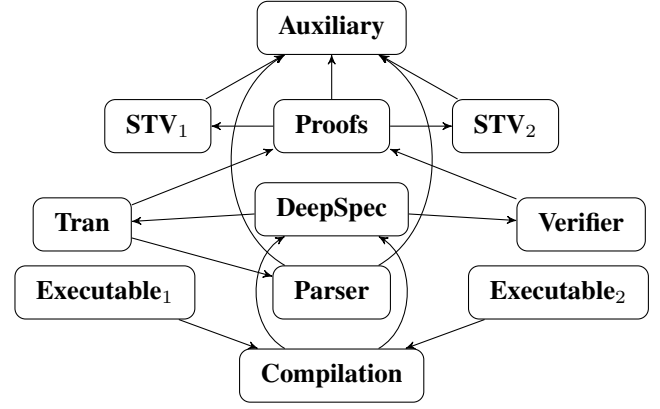


Fig. 3. Architecture of the Framework

putational 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 preceding 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 4. However, it is developed in a way that automatically operates regardless of which STV algorithm is 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 x86_64 executable verifier \mathcal{V}_i .

V. THE GENERIC STV FORMALISED

We proceed through three steps, namely specification, implementation and verification of implementations in HOL4 to

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

Fig. 2. An example of Evidence

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 2 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 evidences. To this end, we need decision procedures used for deciding whether or not a given evidence is valid. Therefore we define in the implementation sub-module boolean-valued functions meant 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.

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 enough skills in grasping functional programming style 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 carried out in a functional language.

A. Data Structure of Machine States

We choose the data structure given in Definition 5 for implementing the abstract data type underlying the generic STV. We have four reasons for this choice of data structure.

- HOL4 has well developed libraries comprised of verified assertions of operations on list structure. By structuring the data on lists, we facilitate us and users with exploitation of already verified tools to formalise the framework and avoid inventing the unnecessary from the scratch.
- HOL4 also has mighty tactics for discharging proof obligations on assertions involving list structure and operations on it. We therefore come to provide us with means for ease of verification of the formalised assertions.
- Understanding the data structure used in the implementation of the framework is critical for ease of its usability and extensibility by third parties. The type *judgement* closely models concrete evidences such as the one in

figure 2. Therefore one can sensibly perceive the abstraction step taken for modelling concrete data which in turn enhances understandability of the framework and its mechanism.

- As the abstract syntactic representation of evidences closely represents concrete evidences, designing the parser and inspecting its correctness is less challenging.

Definition 5: We formalise machine states as an inductive type *judgement* whose constructors are *NonFinal* and *Final*;

judgement =
 $\text{NonFinal } (\text{ballots} \times \text{tallies} \times \text{piles} \times \text{cand list} \times \text{cand list} \times \text{cand list} \times \text{cand list})$
 $\mid \text{Final } (\text{cand list})$

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. 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.

Remember that objects whose type is *piles* serve as containers for recording the ballots attracted by each candidate and storing them under the candidate's name. Our choice for 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 $(\text{cand} \times (\text{ballots list}))\text{ list}$;

(a): Some STV schemes such as the lower house ACT and Tasmania STV, employ a nation 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 made the tally of the candidate reach or exceed the quota and therefore be elected. Only the last parcel, instead of all, of an elected candidate is then distributed the ballots received by the candidate. 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* as the underlying arithmetic for the formalisation, instead of other candidates such as floating numbers. Elections with an STV counting scheme have a small margin of victory especially for the last vacancy left to fill. This margin may happen to be less than magnitude of the mistake in calculations due to aggregation of rounding errors as it is with floating points. Therefore one must sensibly avoid paying the high cost of electing a wrong candidate by falling into traps of imprecisions in calculations introduced by unintelligent choices. Using exact fractions allows us safe correct handling of calculations.

B. The Semantics and its Auxiliary Components

The semantics of each machine transition label is constituted of conjunctions of formally specified omnipresent pre- and post-conditions across STV schemes which legally inform the officers when and how to take an 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-elect transition.

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

- 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-elect. 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 appears 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 forms the semantics of the transition label transfer-elect. The predicate *TransferAuxSpec* defines the semantics of this transition.

Transfer-elect, as is the case for other formalised transitions, is parametrised by the quota *qu*, 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 *bl₂* 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.

$$\begin{aligned} \text{TransferAuxSpec } (\dots) \text{ } ba \text{ } t \text{ } t' \text{ } p \text{ } p' \text{ } bl \text{ } bl_2 \text{ } bl'_2 \text{ } e \text{ } e' \text{ } h \text{ } h' \iff \\ bl_2 = [] \wedge bl \neq [] \wedge ba = [] \wedge bl'_2 = [] \wedge t' = t \wedge e' = e \wedge \\ h' = h \wedge \text{LENGTH } e < st \wedge (\forall d. \text{mem } d (h \# e) \Rightarrow \text{mem } d l) \wedge \\ (\forall d. \dots bl \Rightarrow \dots l) \wedge \text{ALL_DISTINCT } (h \# e) \wedge \\ \dots l \wedge \dots \wedge \dots \end{aligned}$$

$$\text{Valid_Init_CandList } l \iff l \neq [] \wedge \text{ALL_DISTINCT } l$$

The predicate *Valid_Init_CandList* 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 computational 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*.

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

$$\begin{aligned} \text{Valid_PileTally_dec2 } t \text{ } [] &\iff \text{True} \\ \text{Valid_PileTally_dec2 } t \text{ } (l_0::ls) &\iff \\ \text{if mem } l_0 (\text{MAP FST } t) \text{ then Valid_PileTally_dec2 } t \text{ } ls \text{ else False} \end{aligned}$$

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-elect semantics *TransferAuxSpec*.

$$\begin{aligned} \text{TransferAuxDec } (\dots) \text{ } ba \text{ } t \text{ } t' \text{ } p \text{ } p' \text{ } bl \text{ } bl_2 \text{ } bl'_2 \text{ } e \text{ } e' \text{ } h \\ h' \iff \\ \text{NULL } bl_2 \wedge e' = e \wedge h' = h \wedge t' = t \wedge \text{LENGTH } e < st \wedge \\ \text{list_MEM_dec } (h \# e) \text{ } l \wedge \text{list_MEM_dec } bl \text{ } l \wedge \\ \text{ALL_DISTINCT } (h \# e) \wedge \text{Valid_PileTally_dec1 } t \text{ } l \wedge \\ \text{Valid_PileTally_dec2 } t \text{ } l \wedge \text{Valid_PileTally_dec1 } p \text{ } l \wedge \\ \dots l \wedge \dots \wedge \dots \end{aligned}$$

In the same manner as elaborated for transfer-elect, we obtain specification, 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 that for each transition, the implementation corresponds with its specification.

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

One can already proceed to synthesise an executable verifier from the machine. However, such a verifier generically operates for any algorithm that belongs to the STV family without being able to distinguish between evidence produced by computation with different STV members. To produce verifiers that can distinctively check information in evidence of specific STV algorithms, we need to enrich the machine semantics. As mentioned earlier, enrichment of the semantics which we refer to as instantiation happens by augmenting pre- and post-conditions particular to individual algorithms.

VI. INSTANTIATIONS OF THE MACHINE

We exemplify how instantiation of the transfer-elected succeeds for transferring surplus of elected candidates based on the ACT STV ¹. Instantiation of other machine transitions proceed in a similar manner. Under 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 more candidate exceeds the quota
- ₂ the parcel of an elected with surplus is not empty.
- ₃ distribution of the surplus of an elected candidate proceeds in one single step.
- ₄ surplus of elected candidates is distributed one at a time beginning with those who are elected earlier.
- ₅ pile of the candidate whose surplus is transferred is emptied in the post-state of transfer-elect.
- ₆ always the last parcel of votes received (which resulted in a surplus) are 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 has happened as a result of earlier elect, transfer or eliminate actions)
- ₇ 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.

$\text{TransferActSpec } (qu, st, l) j_1 j_2 \iff$
 $\exists ba nba t nt p np bl nbl bl_2 nbl_2 e ne h nh.$
 $j_1 = \text{NonFinal } (ba, t, p, bl, bl_2, e, h) \wedge$
 $\dots nt p np bl bl_2 nbl_2 e ne h nh \wedge$
 $(\forall c'. \text{mem } c' h \Rightarrow \exists x. \text{mem } (c', x) t \wedge x < qu) \wedge$
 $\exists l' c.$
 $(bl = c::l' \wedge \text{mem } c l \wedge$
 $(\forall l''. \text{mem } (\dots, \dots) p \Rightarrow l'' \neq []) \wedge nbl = l' \wedge$
 $nba = \text{LAST } (\dots, \dots) \wedge \dots np \wedge \forall d'. \dots \Rightarrow \dots) \wedge$
 $j_2 = \text{NonFinal } (nba, nt, np, nbl, nbl_2, \dots, \dots)$

We augment the formal counterparts of the •_i conditions to the clauses given in section V-B to obtain the specification TransferActSpec for transfer-elect of ACT STV. The last conjunct of TransferActSpec asserts that the pile of every candidate other than c remains the same in both pre- and post-state of the transition. 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 formalisation of the elect transition instead of transfer-elect.

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-elect. For example, the function get_cand_tally 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).

$\text{ALL_DISTINCT } (\text{MAP FST } t) \wedge \text{mem } (c, x) t \Rightarrow$
 $\text{get_cand_tally } c \ t = x$

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

$\text{less_than_quota } qu \ l \ ls \iff$
 $\text{EVERY } (\lambda h. \text{get_cand_tally } h \ l < qu) \ ls$

We show less_than_quota is a computational realisation of its specification:

$(\forall c. \text{mem } c \ h \Rightarrow \exists x. \text{mem } (c, x) t \wedge x < qu) \wedge \text{ALL_DISTINCT } (\text{MAP FST } t) \wedge$
 $(\forall c''. \text{mem } c'' \ h \Rightarrow \text{mem } c'' (\text{MAP FST } t)) \Rightarrow$
 $\text{less_than_quota } qu \ t \ h$

Moreover less_than_quota enforces its specification.

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

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

$\text{TransferActDec } (qu, st, l) (\text{NonFinal } (ba, t, p, bl, bl_2, e, h))$
 $(\text{NonFinal } (ba', t', p', bl', bl'_2, e', h')) \iff$
 $\dots ba \ t \ t' \ p \ p' \ bl \ bl_2 \ bl'_2 \ e \ e' \ h \ h' \wedge \text{less_than_quota } qu \ t \ h \wedge$
 $\text{case } bl \text{ of}$
 $[] \Rightarrow \text{F}$
 $| hbl::tbl \Rightarrow$
 $(\text{let}$
 $\quad gcp = \text{get_cand_pile } hbl \ p$
 in
 $\neg \text{NULL } gcp \wedge \text{mem } hbl \ l \wedge bl' = tbl \wedge ba' = \text{LAST } gcp \wedge$
 $\text{mem } (hbl, []) \ p' \wedge \text{subpile1 } hbl \ p \ p' \wedge \text{subpile2 } hbl \ p' \ p)$
 $\text{TransferActDec } v_0 (\text{Final } v_1) \ v_2 \iff \text{F}$
 $\text{TransferActDec } v_3 (\text{NonFinal } v_9) (\text{Final } v_5) \iff \text{F}$

We next demonstrate how the framework modularly extends to instantiations with various STV algorithms.

A. Variations in Instantiation

We discuss instantiation of the transfer-elect based on the Victoria and CADE STV. As we have already illustrated how a specification and the corresponding implementation of an algorithm advance, we therefore only elaborate on how they vary from ACT STV in textual descriptions of their semantics and subsequently their implementations.

1) *Victoria STV*: Legislature of Victoria's counting scheme does not strictly speak about the notion of parcel. However, as explained under the subsection V-A it transfers votes of

¹The Tasmania STV also uses similar surplus transfer mechanism.

an eliminated candidate stepwise which therefore requires separating votes into different chunks (or parcels). Having eliminated possible misunderstanding, note that Victoria STV accords with ACT STV on every \bullet_i item except for $i \in \{6, 8\}$.

- _{6'} transfer all of the surplus votes of an elected candidate at a reduced fraction.
- _{8'} the fractional transfer value is computed based on all of the surplus (not necessarily depending on the last parcel).

Note that we deal with updating fractional transfer value in the semantics of elect transitions. Also in instantiations of the elect semantics, we guarantee elements of the backlog of elected candidates (bl_1) are all above the quota. Therefore we formalise Victoria's transfer-elect semantics as follows.

```

TransferVicDec (qu,st,l) (NonFinal (ba,t,p,bl,bl2,e,h))
  (NonFinal (ba',t',p',bl',bl'2,e',h'))  $\iff$ 
... .. ba t t' p p' bl bl2 bl'2 e e' h h'  $\wedge$  less_than_quota qu t h  $\wedge$ 
case bl of
|  $\square \Rightarrow$  F
| hbl::tbl  $\Rightarrow$ 
  (let
    gcp = get_cand_pile hbl p
  in
    bl' = tbl  $\wedge$   $\neg$ NULL gcp  $\wedge$  ba' = FLAT gcp  $\wedge$  mem (hbl, $\square$ ) p'  $\wedge$ 
    subpile1 hbl p p'  $\wedge$  subpile2 hbl p' p)
TransferVicDec v0 (Final v1) v2  $\iff$  F
TransferVicDec v3 (NonFinal v9) (Final v5)  $\iff$  F

```

2) *CADE STV*: This scheme is radically different than “standard” STV algorithms. In particular, to the best of our knowledge, every “normal” STV algorithm at least respects

- ₁. However CADE violates not only this condition but also all of the ACT's transfer-elect semantics components except
- ₂. This outrageous 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-elect 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-elect.
- ★₄ the election restarts after each round of transfer-elect.

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.

```

TransferCadeDec (qu,st,l) (NonFinal (ba,t,p,bl,bl2,e,h))
  (NonFinal (ba',t',p',bl',bl'2,e',h'))  $\iff$ 
... .. ba t t' p p' bl bl2 bl'2 e e' h h'  $\wedge$ 
case bl of
|  $\square \Rightarrow$  F
| hbl::tbl  $\Rightarrow$ 
  tbl =  $\square$   $\wedge$  bl' =  $\square$   $\wedge$  ba' = APPEND_ALL p  $\wedge$ 
   $\neg$ NULL (get_cand_pile hbl p)  $\wedge$  ALL_EMPTY l p'  $\wedge$  h' = l
TransferCadeDec v0 (Final v1) v2  $\iff$  F
TransferCadeDec v3 (NonFinal v9) (Final v5)  $\iff$  F

```

Remark 1: In this section, when we instantiated transitions of the machine, particularly transfer-elect, we named instantiations differently. For example, we represented instantiation of transfer-elect with the Victoria STV by TransferVicDec. This choice was out of pedagogical purposes to assist the reader with understanding the work. In actual engineering however we practice a sensible alternative. 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 Victoria STV, we have a module accordingly named and inside the module we refer, for instance, to instantiation of transfer-elect as TransferDec or TransferSpec without the infix Vic.

Remark 2: 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* instantiations of transitions regardless of the algorithm used. Therefore we can formulate evidence verifier based on the names of the transitions but call in the desired instantiation module to embody the names with the semantics of the algorithm intended to obtain a verifier for. Consequently the verifier, translation, deepSpec and compilation modules 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 next proceed to verify logical equivalences of the specification of each transition as a unity with its implementation. Based on our experience, proving such equivalence roughly requires 200 lines of HOL4 encoding. We desire to automate them in a way that they practically approximate the following Conjecture.

Conjecture 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. In light of the remark 1 in the proofs module, we uniformly declare the desired equivalence between the specification and implementation of an instantiation transition label and discharge the proof. Engineered this way, under two scenarios the conjecture holds for the proofs module. However for a third scenario, proofs

in the module may break so that discharging the equivalences becomes a semi-automatic interactive procedure. The proofs nonetheless remain reusable after proper refinements so that one needs not encoding 200 lines again.

Scenario one: If the algorithm \mathcal{A} already exists in our framework then the conjecture is satisfied. Note that we have already formalised and verified instantiation of the machine with five different STV algorithms.

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 conjecture.

Scenario three: Suppose there is a clause in 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 there 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 4 the notion of verifier depends on instantiation of the machine with an algorithm \mathcal{A} . In light of Remark 1, we only need to develop the verifier module once and simply vary the parent instantiation module to adapt the definition of the verifier for a different instantiation. So let's assume $\hat{\mathcal{A}}_{spec} = \langle \mathcal{S}, \mathcal{T}, (\mathcal{S}_t^{spec})_{t \in \mathcal{T}} \rangle$ and $\hat{\mathcal{A}}_{dec} = \langle \mathcal{S}, \mathcal{T}, (\mathcal{S}_t^{dec})_{t \in \mathcal{T}} \rangle$ are the specification and implementation of an STV algorithm \mathcal{A} where for any $t \in \mathcal{T}$, $\mathcal{S}_t^{dec} \Leftrightarrow \mathcal{S}_t^{spec}$. Then we define \mathcal{V}_{spec} as the specification of the verifier and implement \mathcal{V}_{dec} as its computational counterpart. Drawing on proofs established between \mathcal{S}_t^{dec} and \mathcal{S}_t^{spec} for the transitions, we formally prove the following result.

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 on 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  $\Leftrightarrow$ 
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 []  $\Leftrightarrow$  F
valid_judgements_dec v1 [Final v2]  $\Leftrightarrow$  T
valid_judgements_dec v3 [NonFinal v10]  $\Leftrightarrow$  F
valid_judgements_dec params (j0::j1::js)  $\Leftrightarrow$ 
Valid_Step params j0 j1  $\wedge$ 
valid_judgements_dec params (j1::js)
```

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

```
Check_Parsed_Certificate params []  $\Leftrightarrow$  F
Check_Parsed_Certificate params
(first_judgement::rest_judgements)  $\Leftrightarrow$ 
Initial_Judgement_dec (SND (SND params)) first_judgement  $\wedge$ 
valid_judgements_dec params (first_judgement::rest_judgements)
```

One can use Check_Parsed_Certificate to run small sample elections manually encoded in HOL4's environment. However real evidence such as the one in figure are stored in a file in an operating system which need 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 Check_Parsed_Certificate 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, its ecosystem and the compiler.

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 HOL4 provably extends to that of \mathcal{V}_τ . The translated verifier \mathcal{V}_τ is a pure function operating in CakeML's environment. In order 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 a Basis library 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 upon compilation check_count behaves as follows.

check_count upon synthesis accepts a file as input on the standard 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 of the evidence 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 them 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 messages informing us where it takes place.

VIII. EXPERIMENTAL RESULTS

IX. RELATED WORK

X. CONCLUSION