Linear Logical Voting Protocols

Henry DeYoung[1] and Carsten Schürmann[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
hdeyoung@cs.cmu.edu
[2] IT University of Copenhagen, Copenhagen, Denmark
carsten@itu.dk

Abstract. Current approaches to electronic implementations of voting protocols involve translating legal text to source code of an imperative programming language. Because the gap between legal text and source code is very large, it is difficult to trust that the program meets its legal specification. In response, we promote linear logic as a high-level language for both specifying and implementing voting protocols. Our linear logical specifications of the single-winner first-past-the-post (SW-FPTP) and single transferable vote (STV) protocols demonstrate that this approach leads to concise implementations that closely correspond to their legal specification, thereby increasing trust.

1 Introduction

Determining the outcome of an election is rarely as straightforward as simply counting the votes and declaring the candidate with the most votes to be the winner. Even for relatively simple voting protocols, such as first-past-the-post, election laws prescribe the detailed provisions for tallying votes and computing the final result. Legal language is precise enough to be used in courts of law to settle debates about the lawfulness of a traditional election implementation (e.g., one that uses paper ballots), but computer-based implementations pose unique challenges.

Because election laws are not written in a formal language, they cannot be directly executed by a computer. Instead, humans translate the legal text to source code of a programming language; typically a general-purpose imperative language, such as Java or C, is the target.

However, this approach to computer-based implementations is problematic: it is unreasonable to expect that the translation from the informal specification in legal text to its implementation as source code will be trusted outright. In large part, this is because programs written in general-purpose imperative languages are comparatively low-level and complex. To verify that such programs correctly implement their legal specifications, one must reason about concrete data structures, exotic language features (e.g., inheritance and method overloading), and vast third-party libraries. Certifying all of these components in full detail is extremely challenging and costly, if not impossible—the gap between legal text and source code is simply much too large.

In response to these problems, this paper proposes to use formal logic—more specifically, *linear logic*—as a foundation for electronic elections. First, logic will serve as an intermediate, formal specification language: rather than translating the legal text to low-level source code, it will be translated to a set of logical formulas. Because the logic allows a high level of abstraction, these formulas will be in close correspondence with the legal text from which they were derived. This minimizes the conceptual distance between the two, thereby increasing trustworthiness.

Second, by way of logic programming languages like the well-known Prolog [6], formal logic also provides a means of programming voting protocols declaratively. The declarative programming paradigm has the advantage of narrowing the gap between the formal specification and its implementation. In logic programming, for instance, the specification's logical formulas and the program's source code are one and the same; the operational behavior of the program is derived from a fixed proof-search strategy for the logical connectives that comprise the specification.

Thus, the trusted components of this approach are: *1*) that the formal, logical specification adequately reflects the informal, legal specification; and *2*) that the logic programming engine is correctly implemented. Requiring some degree of trust in the adequacy of the logical specification is unavoidable; however, as described above, because the logical specification is in close correspondence with the legal text, the conceptual gap to be bridged by adequacy is minimized and trustworthiness increases. Moreover, instead of trusting the logic programming engine, one can choose to trust a much simpler proof checker that validates the explicit proof objects produced by the engine; these proof objects are essentially human-readable abstract execution traces, and therefore can also be audited.

To meet the above goals, first-order logic (and its corresponding logic programming language, Prolog) would indeed be a technically adequate choice of logic. But, we contend that it is not an ideal choice. This paper instead advocates the use of linear logic [12,4], a logic in which assumptions are treated as resources that must be used exactly once. (Sect. 2 provides a brief introduction to linear logic and contrasts it with first-order logic.)

To illustrate the benefits of linear logic as a foundation for electronic elections, this paper presents full linear logical specifications of two voting protocols: single-winner first-past-the-post (SW-FPTP, Sect. 3) and proportional representation through the single transferable vote (STV, Sect. 5). Both protocols are widely used in practice: for example, first-past-the-post (also known as winner-take-all) for national elections in the United States and single transferable vote for parliamentary elections in Ireland, Malta, and Australia. More importantly, these protocols are valuable benchmarks because they represent two extremes of protocol complexity. That elegant characterizations of two such diverse protocols are possible speaks to linear logic's robustness as a specification language.

Linear logical specifications of these protocols can be transliterated, in a fully syntactic way, to source code of linear logic programming languages, such as LolliMon [13] and Celf [14]. To provide intuition about the operational behavior of specifications, Sect. 4 of this paper sketches a typical Celf execution of the single-winner first-past-the-post protocol. The transliterations of both protocols to Celf syntax are available at `http://www.itu.dk/~carsten/files/voteid2011.tgz`, if the reader wishes to experiment further.

Another benefit of our linear logical approach is the ability to formally prove the operational correctness of specifications using existing techniques [10,9]. In Sect. 6, we sketch proofs of correctness for the SW-FPTP and STV protocols.

Finally, it is also important to clarify what this paper does not set out to accomplish. First, this paper concentrates on *verified* elections, wherein an a priori static analysis verifies that the election software meets its specification. This is in contrast with *voter-verifiable* elections, which use end-to-end techniques, such as Prêt à Voter [5], whereby voters can convince themselves that the final tally is correct. In general, we believe that the two approaches are complementary: even with end-to-end techniques for detecting anomalies, one should still strive to minimize the occurrence of such costly errors beforehand by running verified software.

Second, beyond operational correctness, voters expect voting protocols to possess security properties such as privacy and coercion-resistance. Although we contend that linear logical protocol specifications will be readily amenable to reasoning about such meta-theoretic properties, we leave this to future work.

Related Work. In the past, there have been attempts to formally prove the implementation of an electronic voting protocol correct. Using the applied π-calculus, Delaune, Kremer, and Ryan [8] modeled a simple protocol and proved its fairness, eligibility, privacy, receipt-freeness, and coercion-resistance with Blanchet's ProVerif tool. Their work differs from ours in that they concentrate on the security of the protocol whereas we are interested in the auditable correctness of implementations. Perhaps most closely related is the work by Cochran and Kiniry on specifying STV in JML and ESC/Java [7]. Unfortunately, JML and ESC/Java are logically unsound. Program verification must be supplemented by testing to guarantee a reasonable level of program correctness. In contrast, our logical approach to implementing STV guarantees correctness automatically, thereby rendering testing superfluous. Aside from implementation concerns, the literature also contains proposals to improve the security of the STV protocol using cryptography [2]. Those ideas are largely orthogonal to the ambitions of this paper.

2 A Brief Introduction to Linear Logic[1]

Traditional first-order logic is concerned solely with truth. Being an abstract idea, truth is inherently free. Consequently, in traditional logic, each logical assumption may be used as many or as few times (including none) as desired—it has no cost.

[1] We encourage the reader who is interested in a more complete introduction to linear logic to refer to Philip Wadler's excellent tutorial [15].

On the other hand, linear logic admits that, unlike truth, not everything is free. It instead concerns itself with consumable, valuable resources. Because resources are consumable, they may not be freely duplicated and may be used at most once; because resources are also valuable, they may not be freely disposed and must be used at least once. To reflect this, linear logic represents resources as logical assumptions that must be used *exactly* once. With this *resource discipline*, linear logic is able to express—more elegantly and concisely than can traditional first-order logic—operations that must occur only once. Because voting protocols in particular rely significantly on the one-occurrence idiom (e.g., registering each voter only once or counting each ballot only once), this elegance is crucial to minimizing the conceptual gap between the informal, legal specification and the formal, logical specification.

2.1 Connectives of Linear Logic

Full linear logic contains a rich set of connectives for building formulas. To assign a logic programming interpretation, restrictions must be placed on the ways in which these connectives fit together so that proof search becomes more deterministic. Thus, in Celf [14] and its predecessor, LolliMon [13], the formulas of linear logic are polarized into positive and negative classes [1] and a monad is used to prevent interference between the two [16]. For the examples in this paper, only the following fragment of polarized monadic linear logic is needed:

$$\begin{array}{ll} \text{Negative Formulas} & A^-, B^- ::= P^- \mid \forall x{:}\tau.\, A^- \mid A^+ \multimap \{B^+\} \\ \text{Positive Formulas} & A^+, B^+ ::= A^+ \otimes B^+ \mid \mathbf{1} \mid {!A^-} \mid A^- \end{array}$$

The fragment includes atomic formulas P^-, universal quantification $\forall x{:}\tau.\, A^-$, linear implication $A^+ \multimap \{B^+\}$, simultaneous conjunction $A^+ \otimes B^+$ and its unit $\mathbf{1}$, the unrestricted modality $!A^-$, and an inclusion, A^-, of negative formulas as positive formulas.

To present the meanings of these connectives, we will now develop a specification of voter check-in at a polling place. Prior to election day, each voter receives a voting authorization card in the mail. To check in at her designated polling place on election day, the voter exchanges her voting authorization card for a blank ballot form. Because each voter receives only one authorization card, the card thus helps prevent ballot stuffing.

In traditional logic, one might try to specify this check-in process by taking as an axiom the formula

$$\textit{voting-auth-card} \rightarrow \textit{blank-ballot} \quad :$$

if a voter has a voting authorization card, then she may have a blank ballot form. However, this specification would allow proofs of such nonsense as $\textit{voting-auth-card} \rightarrow \textit{blank-ballot} \wedge \textit{voting-auth-card}$: if a voter has a voting authorization card, she can receive a blank ballot and keep her authorization card. By iterating this proof, one can show that, under this specification, ballot stuffing is possible: $\textit{voting-auth-card} \rightarrow \textit{blank-ballot} \wedge \cdots \wedge \textit{blank-ballot} \wedge \textit{voting-auth-card}$. Therefore, this specification of the check-in procedure is clearly unsound.

Linear Implication, $\multimap$. The problem is one of expressivity—traditional implication does not express that the check-in process *consumes* the voter's authorization card. But, as a logic of resources, linear logic provides just the right expressive power. It includes the linear implication formula $A^+ \multimap \{B^+\}$, which, like the traditional implication, is a procedure for producing resource B^+ if given A^+; unlike the traditional implication, however, this procedure consumes resource A^+ as part of the production.[2]

Thus, a sound specification of voter check-in is given by taking as an axiom the linear logical formula

$$\textit{voting-auth-card} \multimap \{\textit{blank-ballot}\} \quad :$$

the check-in process consumes the voter's authorization card and gives her a blank ballot in exchange.

Simultaneous Conjunction, $\otimes$, and Its Unit, 1. Now suppose that voters are also required to present a photo ID during check-in. The specification will have the same basic structure: 'a voting authorization card and a photo ID' $\multimap$ $\{\textit{blank-ballot}\}$. But how can we express the 'a voting authorization card *and* a photo ID' resource as a formula of linear logic?

Fortunately, linear logic provides a simultaneous conjunction, $A^+ \otimes B^+$ (read 'both resources A^+ and B^+'). Thus, a specification of the revised check-in process can be given by the formula

$$\textit{voting-auth-card} \otimes \textit{photo-ID} \multimap \{\textit{blank-ballot}\} \quad :$$

when a voter gives a voting authorization card and a photo ID, she receives a blank ballot form in exchange. (Note that $\otimes$ binds more tightly than $\multimap$.)

Linear logic also includes a unit for simultaneous conjunction, **1** (read 'nothing'), which represents the empty collection of resources. The proposition **1** is primarily used in the idiom $A^+ \multimap \{\mathbf{1}\}$, which consumes resource A^+ and produces nothing in return.

Unrestricted Modality, !. The prior specification of the check-in process, $\textit{voting-auth-card} \otimes \textit{photo-ID} \multimap \{\textit{blank-ballot}\}$, is not fully satisfactory, however. Because *photo-ID* is treated as a resource and linear implication (which consumes the resources it is given) is used, this axiom specifies a check-in process in which voters must relinquish their photo IDs to vote! This is not the intent; voters should always retain their photo IDs. And so, at first glance, *photo-ID* does not appear to fit into the resource discipline of linear logic.

However, the unrestricted modality, $!A^-$, of linear logic provides a way out. The proposition $!A^-$ is a version of A^- that is not subject to the resource

[2] The braces around B^+ denote a monad that is not found in conventional presentations of linear logic. It is used to give a committed-choice operational semantics for the logic programming interpretation [13] that is important to our work.

discipline—an assumption $!A^-$ can be used an unlimited number of times (including none). Alternatively, one may think of $!A^-$ as stating that A^- is a fact that will remain true regardless of how the system evolves.

Using the ! modality, the revised specification can therefore be given by

$$\textit{voting-auth-card} \otimes !\textit{photo-ID} \multimap \{\textit{blank-ballot}\} \quad :$$

when a voter gives an authorization card and shows a photo ID, she receives a blank ballot form. (Note that ! binds more tightly than $\otimes$ and $\multimap$.)

Universal Quantification, $\forall x{:}\tau$. Strictly speaking, the current specification of voter check-in does not capture the requirement that the name on the authorization card must match the name on the photo ID.

This problem can be resolved using universal quantification. In linear logic, multi-sorted universal quantification, $\forall x{:}\tau.\, A^-$, behaves just as in traditional logic. In particular, the members of the domain of quantification are not subject to a resource discipline. Thus, the specification may be revised to

$$\forall v{:}\mathsf{voter}.\, \big(\textit{voting-auth-card}(v) \otimes !\textit{photo-ID}(v) \multimap \{\textit{blank-ballot}\}\big) \quad :$$

when a voter v gives *her* authorization card and shows *her* photo ID, she receives a blank ballot form.

3 A Linear Logical Specification of First-Past-the-Post

To demonstrate how linear logic can be used to specify voting systems, we now present a concise, elegant specification of single-winner first-past-the-post voting (SW-FPTP). In SW-FPTP voting, each voter casts a ballot on which she has selected a single candidate. After all ballots have been counted, the candidate with greatest vote total is determined; this candidate is declared the winner. Because SW-FPTP voting is relatively simple, it makes an ideal first example.

For our specification of SW-FPTP, we must introduce several predicates, which are summarized in Table 1. The *uncounted-ballot*, *hopeful*, *defeated*, and *elected* predicates are used to characterize the ballot box and the candidates' electoral statuses, and the *count-ballots* and *determine-max* predicates indicate progress through the algorithm's two phases. We also assume the existence of the usual ordering predicates on natural numbers, such as $!(N \geq N')$.

SW-FPTP voting is specified by the collection of linear logical axioms shown in Fig. 1. (For conciseness, we follow the standard convention that universal quantification is implicit for all variables written in upper case.) The count/run and count/done axioms specify how the ballot counting phase of SW-FPTP works, whereas max/run and max/done characterize a random tournament for finding the candidate who has the greatest vote total. Although it would be straightforward to use other tie-breaking criteria, for simplicity of presentation we will assume that ties are broken arbitrarily.

Table 1. Descriptions of predicates used in the SW-FPTP specification

Predicate	Meaning
uncounted-ballot(C)	An uncounted ballot for candidate C.
hopeful(C, N)	Candidate C is not yet defeated nor elected, and N ballots have been counted for C thus far.
$!$*defeated*(C)	Candidate C has been (and will remain) defeated.
$!$*elected*(C)	Candidate C has been (and will remain) elected.
count-ballots(U, H)	Token to indicate that the algorithm is in the process of counting ballots; there are U uncounted ballots remaining, and H candidates are hopefuls.
determine-max(H)	Token to indicate that the algorithm is in the process of determining which candidate has the greatest vote total; there are H hopeful candidates remaining.

$$\begin{aligned}
\mathsf{count/run} : {} & \mathit{count\text{-}ballots}(U, H) \otimes \\
& \mathit{uncounted\text{-}ballot}(C) \otimes \mathit{hopeful}(C, N) \\
& \quad \multimap \{\mathit{hopeful}(C, N{+}1) \otimes \mathit{count\text{-}ballots}(U{-}1, H)\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{count/done} : {} & \mathit{count\text{-}ballots}(0, H) \\
& \quad \multimap \{\mathit{determine\text{-}max}(H)\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{max/run} : {} & \mathit{determine\text{-}max}(H) \otimes \\
& \mathit{hopeful}(C, N) \otimes \mathit{hopeful}(C', N') \otimes !(N \geq N') \\
& \quad \multimap \{\mathit{hopeful}(C, N) \otimes !\mathit{defeated}(C') \otimes \mathit{determine\text{-}max}(H{-}1)\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{max/done} : {} & \mathit{determine\text{-}max}(1) \otimes \mathit{hopeful}(C, N) \\
& \quad \multimap \{!\mathit{elected}(C)\}
\end{aligned}$$

Fig. 1. A linear logical specification of single-winner first-past-the-post voting

3.1 Counting Ballots with Count/Run and Count/Done

An intuitive reading of the axioms used to specify SW-FPTP ballot counting is:

count/run: 'If we are in the process of counting ballots (*count-ballots*(U, H)) and there is an uncounted ballot for some candidate C (*uncounted-ballot*(C)) and C's vote count is currently N (*hopeful*(C, N)), then C's vote count is updated to $N{+}1$ (*hopeful*$(C, N{+}1)$) and we continue counting ballots (*count-ballots*$(U{-}1, H)$).'

count/done: 'If we are in the process of counting ballots and no uncounted ballots remain (*count-ballots*$(0, H)$), then we have finished counting ballots and should now begin determining which candidate has the greatest vote total (*determine-max*(H)).'

There are several key observations to be made.

Use of Linearity. The linear resource discipline is crucial for the count/run and count/done axioms to adequately specify SW-FPTP ballot counting.

First, the *hopeful* propositions that record candidates' vote counts are treated as linear resources. That is, they are not prefixed with the unrestricted modality, !, which would escape the resource discipline. Being linear resources allows the vote counts to be mutable. This is fundamental to the correctness of the count/run axiom: whenever a ballot for candidate C is counted, the record of C's vote count ($\mathit{hopeful}(C, N)$) is consumed and is replaced with a new, updated record ($\mathit{hopeful}(C, N{+}1)$). If these *hopeful* records were not linear resources, then they would not be consumed and the old vote counts would persist alongside the new count, causing untold confusion.

Second, and equally crucial, $\mathit{uncounted\text{-}ballot}(C)$ is treated as a linear resource. Consequently, ballots are consumed whenever they are counted by the count/run axiom. If ballots were not linear, they would not be consumed upon being counted; they would effectively remain in the ballot box, leaving open the possibility that a ballot could be counted more than once. (Note that we need not preserve the ballot in a counted form, e.g., as in $\mathit{counted\text{-}ballot}(C)$; the candidate's vote count is sufficient to reconstruct the ballots that were cast.)

Treating uncounted ballots as linear resources also provides a further benefit. Because linear logic demands that resources be used *at least* once, the specification framework itself ensures that all ballots are eventually counted by count/run. This is a strong guarantee that is provided to the specification for free!

Tracking the Number of Uncounted Ballots. The reader may wonder why we bother to maintain the invariant that there are U uncounted ballots and H hopeful candidates remaining whenever $\mathit{count\text{-}ballots}(U, H)$ holds. For example, couldn't one just use

$$\begin{aligned}&\mathit{count\text{-}ballots}' \otimes\\ &\mathit{uncounted\text{-}ballot}(C) \otimes \mathit{hopeful}(C, N)\\ &\quad \multimap \{\mathit{hopeful}(C, N{+}1) \otimes \mathit{count\text{-}ballots}'\}\end{aligned}$$

in place of the count/run axiom?

The answer is that this invariant makes it possible to decide when there are no more ballots to count: there are no more ballots exactly when $\mathit{count\text{-}ballots}(0, H)$ holds. This forms the basis for count/done. Without tracking this information, count/done would need to rely on extra-logical machinery, such as negation-as-failure, to check for the absence of $\mathit{uncounted\text{-}ballot}(C)$. And, extra-logical machinery would forfeit the benefits of a purely logical specification as discussed in Sect. 1.

The *count-ballots* and *determine-max* predicates track the number of remaining hopefuls for similar reasons. Doing so provides a purely logical way to determine when exactly one hopeful remains, forming the basis for max/done, as described below.

3.2 Determining the Winner with Max/Run and Max/Done

The max/run and max/done axioms characterize a random tournament for determining which candidate has the greatest vote total. An intuitive reading of the axioms is as follows:

max/run: 'If we are in the process of determining who has the greatest vote total ($\textit{determine-max}(H)$) and there are at least two candidates C and C' that remain hopefuls, with vote totals N and N' respectively, ($\textit{hopeful}(C, N) \otimes \textit{hopeful}(C', N')$) and C's vote total is larger ($!(N \geq N')$), then C remains a hopeful ($\textit{hopeful}(C, N)$) and C' is defeated ($!\textit{defeated}(C')$) and we continue to determine who has the greatest vote total ($\textit{determine-max}(H-1)$).'

max/done: 'If we are in the process of determining who has the greatest vote total and only one candidate remains a hopeful ($\textit{determine-max}(1)$) and that candidate is C ($\textit{hopeful}(C, N)$), then C is declared the winner ($!\textit{elected}(C)$).'

Two key points benefit from further explanation.

Use of the Unrestricted Modality, !. The unrestricted modality is used strategically at three points in the max/run and max/done axioms.

First, the max/run axiom includes $!(N \geq N')$ to ensure that C's vote total is no smaller than that of C'. As presented in Sect. 2.1, the unrestricted modality, !, denotes a fact that remains true regardless of how the system's state evolves. Because natural number inequalities are true independent of the voting system's state, the use of ! here is justified.

Second and third, the max/run and max/done axioms use $!\textit{defeated}(C')$ and $!\textit{elected}(C)$, respectively, to reflect changes in a candidate's status. Just as ! denotes a fact that remains true regardless of how the system's state evolves, these uses of ! express that once a candidate is either defeated or elected her status (in that election) never changes.

No Axiom Dual to Max/Run. At first glance, it may be surprising that there is no axiom, dual to max/run, for the case $!(N \leq N')$. In fact, max/run itself handles this case. For a fixed pair of *hopeful* assumptions, the premise $\textit{hopeful}(C, N) \otimes \textit{hopeful}(C', N')$ can be instantiated in two ways: one for each permutation of those two assumptions. At least one of these will satisfy the inequality $!(N \geq N')$, and so the single max/run axiom suffices.

4 Viewing Specifications as Linear Logic Programs

Thus far we have given a static specification of SW-FPTP as a collection of linear logical axioms. However, viewed through the lens of linear logic programming, such specifications can also be seen as rules defining a forward-chaining, committed-choice linear logic program [13,14]. Thus, specifications can be directly executed using a logic programming engine. (As a demonstration, a transliteration of the SW-FPTP specification into Celf source code is available at `http://www.itu.dk/~carsten/files/voteid2011.tgz`.)

Rules of such logic programs are essentially multiset rewriting rules [3]. For instance, max/run can be seen as the following multiset rewriting rule: choose any two *hopeful* terms and replace the one having a smaller vote count with a corresponding !*defeated* term. Given similar interpretations of the other axioms, the SW-FPTP algorithm can be run by issuing logic programming queries.

Example 1. Consider the following election scenario. Three candidates, a, b, and c, are running for office. We model this with three linear resources that initialize the candidates' vote counts to 0: $\mathit{hopeful}(\mathrm{a}, 0)$, $\mathit{hopeful}(\mathrm{b}, 0)$, and $\mathit{hopeful}(\mathrm{c}, 0)$. Each uncounted ballot is also modeled as a linear resource:

$$\begin{array}{l} \mathit{uncounted\text{-}ballot}(\mathrm{a}), \mathit{uncounted\text{-}ballot}(\mathrm{b}), \mathit{uncounted\text{-}ballot}(\mathrm{a}), \\ \mathit{uncounted\text{-}ballot}(\mathrm{c}), \mathit{uncounted\text{-}ballot}(\mathrm{b}), \mathit{uncounted\text{-}ballot}(\mathrm{a}). \end{array}$$

To initiate the execution, we add the resource $\mathit{count\text{-}ballots}(6, 3)$, which is indexed by the number of uncounted ballots and the number of hopefuls. The execution consists of two phases: in the first phase, the votes are tallied, and in the second phase, the candidate who has the most votes is determined. At each step of the execution, one of the SW-FPTP axioms is applied as a multiset rewriting rule, following the rules of linear logic. First, count/run fires exactly six times, once for each ballot. Next, the rule count/done fires, and the execution commences with the second phase. Determining the winner requires two comparisons, which means that max/run fires twice. And finally, the max/done rule fires, announcing a as the winner and concluding the execution.

The logic programming approach therefore provides two key benefits. First, and most importantly, there is no need to verify the executable code against a separate formal specification: the code and specification are one and the same! (Only a linear logic proof engine needs to be trusted as an interpreter.) This benefit cannot be overemphasized, and is a direct result of choosing linear logic as the high-level, yet fully rigorous, specification language.

Second, the traces of rewriting steps that are produced by the logic programming engine provide an immediate means for auditing the election. Because the traces are, in fact, proof objects in linear logic, auditing is easy: a lightweight linear logic proof checker can formally verify the validity of a trace. In particular, costly recounts become unnecessary because the verifiable traces record each step of vote counting.

5 Single Transferable Vote in Linear Logic

To show that linear logic is robust enough to be used for specifying complex voting systems, we now turn our attention to single transferable vote (STV).

In STV, each voter casts a ballot that lists candidates in order of the voter's preference. To be elected, a candidate must reach a threshold, or quota, of votes. For the purposes of this paper, the particular choice of quota is arbitrary. Because it is commonly used in practice, we choose the Droop quota,

$$\text{quota} = \frac{\#\text{ballots}}{\#\text{seats} + 1} + 1 \ ,$$

however any quota could easily be substituted. Once the quota is computed, the ballots are counted and the following rules are repeated until all open seats are filled.

1. If a candidate has enough votes to meet the quota, she is declared elected. Any surplus votes for this candidate are transferred.
2. If all ballots have been assigned to candidates and no candidate meets the quota, then the candidate with the fewest votes is eliminated and her votes are transferred. If several candidates tie for the fewest votes, one is eliminated at random.
3. When a vote is transferred, it is assigned to the hopeful candidate with the next highest preference listed on that ballot. That is, candidates that are already elected or defeated do not receive transferred votes.
4. If, at any point, there are at least as many open seats as hopeful candidates remaining, then all remaining hopefuls become elected.

5.1 A Linear Logical Specification of Single Transferable Vote

For our specification of STV, we must introduce several predicates, which are summarized in Table 2. The *uncounted-ballot*, *counted-ballot*, *hopeful*, *defeated*, *elected*, *quota*, and *winners* predicates characterize the ballot box, candidates' statuses, and the election's state. The *elect-all*, *defeat-min*, *defeat-min′*, *transfer*, and *begin* predicates are used to indicate progress through the STV algorithm's phases. Finally, *minimum* is an auxiliary predicate used in determining a candidate with the fewest votes. (We again assume the usual ordering predicates on natural numbers, such as $!(N \geq N')$.)

The linear logical axioms that specify STV are given in Fig. 2. Several of these axioms pattern-match on the shape of a list of candidates. Following standard convention, we use $[\,]$ to stand for the empty list and $[C \mid L]$ to stand for the non-empty list with head C and tail L. (We again follow the convention that universal quantification is implicit for variables written in upper case.)

These axioms faithfully encode STV in a concise and elegant fashion—rather than requiring hundreds or thousands of lines of imperative source code, our full STV specification fits on a single page! To make plain the close correspondence of the axioms with the natural language description of STV used in current practice, we will now walk through their meanings.

Beginning the STV Algorithm. The `begin/1` axiom describes the initial step of the STV algorithm: the Droop quota is computed and recorded. Ballot counting is initiated, with no candidates having been declared winners.

Table 2. Descriptions of predicates used in the STV specification

Predicate	Meaning
$\mathit{uncounted\text{-}ballot}(C, L)$	An uncounted ballot with highest preference for candidate C and list L of lower preferences.
$\mathit{counted\text{-}ballot}(C, L)$	A ballot counted for candidate C, with list L of lower preferences.
$\mathit{hopeful}(C, N)$	Candidate C is not yet defeated nor elected, and N ballots have been counted for C thus far.
$!\mathit{defeated}(C)$	Candidate C has been (and will remain) defeated.
$!\mathit{elected}(C)$	Candidate C has been (and will remain) elected.
$!\mathit{quota}(Q)$	Q votes are needed to be elected.
$\mathit{winners}(W)$	The candidates in list W have been elected thus far.
$\mathit{begin}(S, H, U)$	Token to signal that the STV algorithm should begin running. There are S seats up for election, H hopeful candidates, and U ballots cast.
$\mathit{count\text{-}ballots}(S, H, U)$	Token to indicate that the algorithm is counting ballots, and that there are S open seats, H hopeful candidates, and U uncounted ballots remaining.
$!\mathit{elect\text{-}all}$	Token to indicate that there are more open seats than hopefuls remaining; all remaining hopefuls should become elected.
$\mathit{defeat\text{-}min}(S, H, M)$	Token to indicate that the algorithm is in the first step of determining a candidate who has the fewest votes. There are S open seats, H hopeful candidates, and M potential minimums remaining.
$\mathit{defeat\text{-}min}'(S, H, M)$	Token to indicate that the algorithm is in the second step of determining a candidate who has the fewest votes. There are S open seats, H hopeful candidates, and M potential minimums remaining.
$\mathit{minimum}(C, N)$	Candidate C's vote count of N is a potential minimum.
$\mathit{transfer}(C, N, S, H, U)$	Token to indicate that newly defeated candidate C's remaining N votes are being transferred. There are S open seats, H hopeful candidates, and U uncounted ballots.

begin/1 :
$$\begin{array}{l} \mathit{begin}(S,H,U) \otimes \\ !(Q = U/(S+1)+1) \\ \quad \multimap \{!\mathit{quota}(Q) \otimes \mathit{winners}([\,]) \otimes \\ \qquad \mathit{count\text{-}ballots}(S,H,U)\} \end{array}$$

count/1 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,L) \otimes \\ \mathit{hopeful}(C,N) \otimes \\ !\mathit{quota}(Q) \otimes !(N{+}1 < Q) \\ \quad \multimap \{\mathit{counted\text{-}ballot}(C,L) \otimes \\ \qquad \mathit{hopeful}(C,N{+}1) \otimes \\ \qquad \mathit{count\text{-}ballots}(S,H,U{-}1)\} \end{array}$$

count/2 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,L) \otimes \\ \mathit{hopeful}(C,N) \otimes \\ !\mathit{quota}(Q) \otimes !(N{+}1 \geq Q) \otimes \\ !(S \geq 1) \otimes \mathit{winners}(W) \\ \quad \multimap \{\mathit{counted\text{-}ballot}(C,L) \otimes \\ \qquad !\mathit{elected}(C) \otimes \mathit{winners}([C \mid W]) \otimes \\ \qquad \mathit{count\text{-}ballots}(S{-}1,H{-}1,U{-}1)\} \end{array}$$

count/3.1 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,[C' \mid L]) \otimes \\ !\mathit{elected}(C) \\ \quad \multimap \{\mathit{uncounted\text{-}ballot}(C',L) \otimes \\ \qquad \mathit{count\text{-}ballots}(S,H,U)\} \end{array}$$

count/3.2 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,[C' \mid L]) \otimes \\ !\mathit{defeated}(C) \\ \quad \multimap \{\mathit{uncounted\text{-}ballot}(C',L) \otimes \\ \qquad \mathit{count\text{-}ballots}(S,H,U)\} \end{array}$$

count/4.1 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,[\,]) \otimes \\ !\mathit{elected}(C) \\ \quad \multimap \{\mathit{count\text{-}ballots}(S,H,U{-}1)\} \end{array}$$

count/4.2 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,U) \otimes \\ \mathit{uncounted\text{-}ballot}(C,[\,]) \otimes \\ !\mathit{defeated}(C) \\ \quad \multimap \{\mathit{count\text{-}ballots}(S,H,U{-}1)\} \end{array}$$

count/5 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,0) \otimes !(S < H) \\ \quad \multimap \{\mathit{defeat\text{-}min}(S,H,0)\} \end{array}$$

count/6 :
$$\begin{array}{l} \mathit{count\text{-}ballots}(S,H,0) \otimes !(S \geq H) \\ \quad \multimap \{!\mathit{elect\text{-}all}\} \end{array}$$

defeat-min/1 :
$$\begin{array}{l} \mathit{defeat\text{-}min}(S,H,M) \otimes \\ \mathit{hopeful}(C,N) \\ \quad \multimap \{\mathit{minimum}(C,N) \otimes \\ \qquad \mathit{defeat\text{-}min}(S,H{-}1,M{+}1)\} \end{array}$$

defeat-min/2 :
$$\begin{array}{l} \mathit{defeat\text{-}min}(S,0,M) \\ \quad \multimap \{\mathit{defeat\text{-}min}'(S,0,M)\} \end{array}$$

defeat-min′/1 :
$$\begin{array}{l} \mathit{defeat\text{-}min}'(S,H,M) \otimes \\ \mathit{minimum}(C,N) \otimes \\ \mathit{minimum}(C',N') \otimes \\ !(N \leq N') \\ \quad \multimap \{\mathit{minimum}(C,N) \otimes \\ \qquad \mathit{hopeful}(C',N') \otimes \\ \qquad \mathit{defeat\text{-}min}'(S,H{+}1,M{-}1)\} \end{array}$$

defeat-min′/2 :
$$\begin{array}{l} \mathit{defeat\text{-}min}'(S,H,1) \otimes \\ \mathit{minimum}(C,N) \\ \quad \multimap \{!\mathit{defeated}(C) \otimes \\ \qquad \mathit{transfer}(C,N,S,H,0)\} \end{array}$$

transfer/1 :
$$\begin{array}{l} \mathit{transfer}(C,N,S,H,U) \otimes \\ \mathit{counted\text{-}ballot}(C,L) \\ \quad \multimap \{\mathit{uncounted\text{-}ballot}(C,L) \otimes \\ \qquad \mathit{transfer}(C,N{-}1,S,H,U{+}1)\} \end{array}$$

transfer/2 :
$$\begin{array}{l} \mathit{transfer}(C,0,S,H,U) \\ \quad \multimap \{\mathit{count\text{-}ballots}(S,H,U)\} \end{array}$$

elect-all/1 :
$$\begin{array}{l} !\mathit{elect\text{-}all} \otimes \\ \mathit{hopeful}(C,N) \otimes \mathit{winners}(W) \\ \quad \multimap \{!\mathit{elected}(C) \otimes \mathit{winners}([C \mid W])\} \end{array}$$

cleanup/1 :
$$\begin{array}{l} !\mathit{elect\text{-}all} \otimes \\ \mathit{counted\text{-}ballot}(C,L) \\ \quad \multimap \{\mathbf{1}\} \end{array}$$

Fig. 2. A linear logical specification of single transferable vote

Counting the Ballots

- count/1 describes counting a ballot that does not cause its candidate, C, to reach the quota: C's vote total increases and ballot counting continues.
- count/2 describes counting a ballot that causes its candidate, C, to finally reach the quota. C becomes elected, being a hopeful no longer, and is added to the list of winners. Any ballots remaining uncounted for C constitute C's vote surplus; the surplus is randomly selected because ballots are counted in a random order. After C is elected, ballots continue to be counted.
- count/3.1, count/3.2, count/4.1, and count/4.2 express that no more ballots are counted for candidates that are already either elected or defeated. The ballots transfer to the next highest preference; if none exists, the ballot is consumed—that is, the vote is wasted.
- Finally, count/5 and count/6 describe what happens when there are no more ballots to count. If there are fewer open seats than hopefuls remaining (count/5), then a candidate with the fewest votes is defeated; the generated *defeat-min* token begins this process. Otherwise, if there are at least as many open seats as hopefuls (count/6), then all remaining hopefuls are elected.

Defeating a Candidate with the Fewest Votes

- defeat-min/1 labels all hopeful candidates as potential minimums. When there are no more hopefuls to label (i.e., when the H counter reaches 0), defeat-min/2 transitions to the second phase of defeating a candidate.
- defeat-min′/1 and defeat-min′/2 describe a random tournament for finding, among the potential minimums, a candidate with the fewest votes. Candidates not selected as the minimum are restored to their hopeful status (defeat-min′/1). When only one candidate is a potential minimum (i.e., when the M counter reaches 1), that candidate must have the fewest votes; she is defeated and the process of transferring her votes begins (defeat-min′/2).

Transferring a Defeated Candidate's Votes

- transfer/1 expresses that ballots counted for a newly defeated candidate, C, are returned to the ballot box as uncounted ballots. As transfer/2 shows, when the N counter reaches 0, these ballots will be re-counted. Because C is now defeated, re-counting these ballots will effectively transfer them to the next highest preference, if one exists (count/3.2 and count/4.2).

Finishing the STV Election

- elect-all/1 expresses that the STV algorithm finishes by electing all remaining hopefuls. (Note that there may possibly be no remaining hopefuls at this point.) Because this is the last step of the STV algorithm, we may think of this step as continuing forever, idling once all remaining hopefuls have been elected. This justifies the use of the ! modality here and also in count/6.

– When the STV algorithm finishes, the counted ballots will remain as linear resources. The resource discipline of linear logic demands that these be used once. Therefore, the `cleanup/1` axiom consumes any remaining ballots. This is safe because the STV algorithm has already filled all seats.

5.2 Viewing the STV Specification as a Linear Logic Program

As we did for SW-FPTP (Sect. 4), we can view the STV specification as a linear logic program. This provides executable code for STV with the same benefits as for SW-FPTP before: a close correspondence between code and specification, with no separate verification needed, and verifiable traces to audit the election. As a demonstration, a transliteration of the STV specification into Celf source code is available at `http://www.itu.dk/~carsten/files/voteid2011.tgz`.

For STV, because of the coercion problem [2], only trusted individuals should be given access to these traces. Determining the right way to adapt the cryptographic solutions of the coercion problem to the logic programming approach is an intriguing area for future work.

6 Proving the Specifications Correct

From the preceding discussions of their axioms, it should be clear that the specifications correspond to the SW-FPTP and STV protocols, respectively. However, we would like to rigorously prove these claims of correctness. Following ideas for other logical specifications [10,9], we can prove such properties by straightforward induction on the specification's operational semantics. To demonstrate this technique, we will sketch correctness proofs for our protocol specifications.

6.1 Correctness of the SW-FPTP Specification

To prove that the SW-FPTP specification is correct, we must show that all executions elect a candidate who has at least as many votes as all other candidates. First, we establish invariants that characterize valid states in executions of the SW-FPTP protocol.

Lemma 1. *All SW-FPTP axioms preserve the following state invariants.*

– *Exactly one of the following holds:*
  - *there is exactly one assumption count-ballots*(U, H),
  - *there is exactly one assumption determine-max*(H), *or*
  - *there are no assumptions count-ballots*(U, H) *and determine-max*(H).
– *For each candidate* C, *exactly one of the following holds:*
  - $!$*elected*(C) *and there are no uncounted-ballot*(C) *assumptions,*
  - $!$*defeated*(C) *and there are no uncounted-ballot*(C) *assumptions, or*
  - *there is exactly one assumption hopeful*(C, N).

- *If count-ballots*(U, H)*, then*
  - *there are no* $!$*elected*(C) *or* $!$*defeated*(C) *assumptions,*
  - *there are* H *assumptions of the form hopeful*(C, N)*, and*
  - *there are* U *assumptions of the form uncounted-ballot*(C)*.*
- *If determine-max*(H)*, then*
  - *there are no* $!$*elected*(C) *assumptions,*
  - *there are* H *assumptions of the form hopeful*(C, N)*, and*
  - *there are no assumptions of the form uncounted-ballot*(C)*.*
- *If neither count-ballots*(U, H) *nor determine-max*(H)*, then*
  - *either there is exactly one candidate* C *for which* $!$*elected*(C)*, or there are no* $!$*elected*(C) *or* $!$*defeated*(C) *assumptions; and*
  - *there are no hopeful*(C, N) *assumptions.*

Proof. Directly by case analysis on the SW-FPTP axioms.

Next, let a hopeful candidate C's vote total be the sum of the number of *uncounted-ballot*(C) assumptions and the unique value N for which *hopeful*(C, N) holds. Correctness of our SW-FPTP specification then follows:

Theorem 1. *For all partial executions, the following hold for each candidate* C*:*

- *If hopeful*(C, N_1) *holds at the end of the partial execution, then, at the beginning of the partial execution, hopeful*(C, N_0) *holds and* C*'s vote total is the same as it will be at the end of the partial execution.*
- *If* $!$*defeated*(C) *holds at the end of the partial execution, then, at the beginning of the partial execution, either:*
  1. $!$*defeated*(C) *holds, or*
  2. *hopeful*(C, N_0) *holds and* C*'s vote total is no larger than the vote total of some candidate* C' *for which hopeful*(C', N_0') *holds.*
- *If* $!$*elected*(C) *holds at the end of the partial execution, then, at the beginning of the partial execution, either:*
  1. $!$*elected*(C) *holds, or*
  2. *hopeful*(C, N_0) *holds and* C*'s vote total is at least as large as the vote totals of all candidates* C' *for which hopeful*(C', N_0') *holds.*

Proof. By induction on the specification's operational semantics (that is, the length of the execution's trace), using the invariants from Lemma 1.

6.2 Correctness of the STV Specification

Unfortunately, unlike for SW-FPTP, there is no independent, formal model of STV against which we might prove the full correctness of our specification. This, in a sense, is a primary benefit of our linear logical approach: the specification's operational semantics serves as a formal model of STV.

Despite the absence of an independent model, we *can* prove properties that we expect the putative STV specification to possess. For instance, when the STV protocol finishes, it should be that the number of elected candidates exactly equals the number of seats, `SEATS`, that were to be filled by the election (assuming that, initially, there are at least as many candidates as seats). To prove this, we follow a similar strategy as for SW-FPTP.

Let $\#\mathit{elected}$ and $\#\mathit{hopeful}$ be the number of distinct candidates C for which $!\mathit{elected}(C)$ and $\mathit{hopeful}(C, N)$, for some N, hold, respectively. We first establish invariants that characterize valid states in executions of the STV protocol.

Lemma 2. *All STV axioms preserve the following state invariants provided that $\mathit{begin}(S, H, U)$ implies $S = \texttt{SEATS}$, $\#\mathit{elected} = 0$, $H = \#\mathit{hopeful}$, and $S \leq H$.*

- *If $\mathit{count\text{-}ballots}(S, H, U)$, then $S = \texttt{SEATS} - \#\mathit{elected}$, $H = \#\mathit{hopeful}$, and $S \leq H$.*
- *If $\mathit{defeat\text{-}min}(S, H, M)$ or $\mathit{defeat\text{-}min}'(S, H, M)$, then $S = \texttt{SEATS} - \#\mathit{elected}$, $H = \#\mathit{hopeful}$, and $S < H + M$.*
- *If $\mathit{transfer}(C, N, S, H, U)$, then $S = \texttt{SEATS} - \#\mathit{elected}$, $H = \#\mathit{hopeful}$, and $S \leq H$.*
- *If $!\mathit{elect\text{-}all}$, then $\#\mathit{elected} + \#\mathit{hopeful} = \texttt{SEATS}$.*

Proof. Directly by case analysis on the STV axioms.

Then, because $!\mathit{elect\text{-}all}$ holds and no hopefuls remain when the execution concludes, the property is immediate:

Corollary 1. *For all executions of the STV specification, the final state satisfies $\#\mathit{elected} = \texttt{SEATS}$.*

Because STV is the more complicated protocol, it may seem counterintuitive that the STV invariants stated in Lemma 2 are simpler than the SW-FPTP invariants of Lemma 1. In fact, the invariants of Lemma 2 are not the strongest invariants that we could prove, but suffice to prove the property of interest.

7 Conclusion

In this paper, we have promoted linear logic as a practical, mathematical language for the rigorous specification and implementation of voting protocols. We have demonstrated this methodology on the SW-FPTP and STV protocols. Linear logic yields concise specifications, implementations for free, auditable executions, and, perhaps most importantly, formal proofs of operational correctness.

In future work, we plan to extend our linear logic and its logic programming engine with modalities for knowledge, possession, and secrecy, which will greatly increase the expressive strength of the logic. For example, an epistemic modality would give logical meaning to the secrecy properties of encryption. Modal logics of Garg et al. [10,11] will serve as a foundation on which to build. We also intend to develop techniques for reasoning about the security of voting protocols specified in linear logic, such as for proving privacy and coercion-resistance.

Acknowledgements. This material is based upon work supported by a National Science Foundation Graduate Research Fellowship for the first author. The second author was supported in part by grant 10-092309 from the Danish Council for Strategic Research, Programme Commission on Strategic Growth Technologies, and by grant 10-093084 from the Danish Council for Independent Research, Natural Sciences. The authors also wish to thank the anonymous reviewers for their invaluable comments.

References

1. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation 2(3), 297–347 (1992)
2. Benaloh, J., Moran, T., Naish, L., Ramchen, K., Teague, V.: Shuffle-sum: Coercion-resistant verifiable tallying for STV voting. IEEE Transactions on Information Forensics and Security 4(4), 685–698 (2009)
3. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information & Computation 207(10), 1044–1077 (2009)
4. Chang, B.Y.E., Chaudhuri, K., Pfenning, F.: A judgmental analysis of linear logic. Tech. Rep. CMU-CS-03-131R, Carnegie Mellon University (December 2003)
5. Chaum, D., Ryan, P.Y.A., Schneider, S.: A Practical Voter-Verifiable Election Scheme. In: De Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 118–139. Springer, Heidelberg (2005)
6. Clocksin, W.F., Mellish, C.S.: Programming in Prolog, 5th edn. Springer (2003)
7. Cochran, D., Kiniry, J.: Vótáil: A formally specified and verified ballot counting system for Irish PR-STV elections. In: Beckert, B., Marché, C. (eds.) Pre-proceedings of the International Conference on Formal Verification of Object-Oriented Software, Paris, France (June 2010)
8. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security 17(4), 435–487 (2009)
9. DeYoung, H., Pfenning, F.: Reasoning about the consequences of authorization policies in a linear epistemic logic. In: Cortier, V., Shmatikov, V. (eds.) Proceedings of the Workshop on Foundations of Computer Security, Los Angeles, California (August 2009)
10. Garg, D., Bauer, L., Bowers, K.D., Pfenning, F., Reiter, M.K.: A Linear Logic of Authorization and Knowledge. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 297–312. Springer, Heidelberg (2006)
11. Garg, D., Pfenning, F.: A proof-carrying file system. In: 31st IEEE Symposium on Security and Privacy, pp. 349–364. IEEE Computer Society Press, Oakland (2010)
12. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987)
13. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: Barahona, P., Felty, A.P. (eds.) Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 35–46. ACM Press, Lisbon (2005)
14. Schack-Nielsen, A., Schürmann, C.: Celf – A Logical Framework for Deductive and Concurrent Systems (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–326. Springer, Heidelberg (2008)
15. Wadler, P.: A Taste of Linear Logic. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 185–210. Springer, Heidelberg (1993)
16. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework I: Judgments and properties. Tech. Rep. CMU-CS-02-101, Carnegie Mellon University (2002) (revised May 2003)