Machine-checked Reasoning About Complex Voting Schemes Using Higher-order Logic

Jeremy E Dawson, Rajeev Goré, Thomas Meumann

Research School of Computer Science, Australian National University

Abstract. We describe how we first formally encoded the English-language Parliamentary Act for the Hare-Clark Single Transferable Vote-counting scheme used in the Australian state of Tasmania into higher-order logic, producing `SPECHOL`. Based on this logical specification, we then encoded an SML program to count ballots according to this specification inside the interactive theorem prover HOL4, giving us `IMPHOL`. We then manually transliterated the program as a real SML program `IMP`. We are currently verifying that the formalisation of the implementation implies the formalisation of the specification: that is, we are using the HOL4 interactive theorem prover to prove the implication `IMPHOL` $\rightarrow$ `SPECHOL`.

1 Introduction

Two fundamental principles in tallying an election are the transparency and trustworthiness of the process. Strict protocols are enforced when dealing with the ballot boxes and interested parties are provided with the opportunity to scrutinise the tally while it is being undertaken. Thus traditional manual vote-counting methods are designed to ensure trustworthiness via scrutiny.

Despite these measures, manually counting ballots is still error-prone. During the 2013 Senate election, the Australian Electoral Commission (AEC) was required to recount the Western Australian (WA) ballots after a close result. During the recount, approximately 1370 ballots were found to be missing. It is unclear whether these ballots were present in the first count and then mislaid, or whether the original tallies were wrong. The error had the capacity to influence the outcome of the election, so the AEC was forced to re-run the election for the WA seats in its entirety at a cost of approximately AUD 20 million [1]. The electoral commissioner of the AEC itself subsequently resigned.

Paper ballots and manual counting methods in modern elections are therefore increasingly seen as archaic, especially as cash-strapped electoral bodies seek cheaper alternatives. Indeed, numerous electoral bodies are ploughing ahead with electronic vote-casting and vote-counting of preferential votes using computers and the most recent state election in New South Wales in Australia even used an internet voting system which was shown to be vulnerable to vote-tampering [22]. Alarmingly, many of these software systems are not open to scrutiny and some are even officially deemed to be "commercial in confidence" and are deliberately kept from researchers like us who wish to scrutinise the code for correctness.

Given the importance of the task of electing a government, this state of affairs is totally unacceptable.

The ideal of course is to use some form of end-to-end verifiable system which provides strong evidence that electronic ballots are cast-as-intended, transported-without-tampering and that all electronic ballots are included in the final tally *without having to blindly trust the underlying computer code.* Unfortunately, such systems do not guarantee that the electronic ballots are counted correctly according to complex vote-counting schemes such as single-transferable voting (STV) and methods for extending them to STV are in their infancy [7].

A "voting scheme" is a method that spells out the structure of a ballot, how to cast a vote using such a ballot, and how to count such votes regardless of whether these activities are carried out using pen and paper, hand-counting or electronically. We describe a methodology for formally reasoning about complex vote-counting schemes. Specifically, we describe how we first formally encoded the English-language description of the Hare-Clark Single Transferable Vote-counting scheme used in the Australian state of Tasmania into higher-order logic, giving a formula called `SPECHOL`. We then created a more algorithmic version using the syntax of the functional programming language Standard ML (SML) to give a formula called `IMPHOL`. We manually transliterated this formula into an actual SML program `IMP` to count ballots. We are currently verifying that the formalisation of the implementation logically implies the formalisation of the specification: that is, we are using the HOL4 interactive theorem prover to machine-check the implication `IMPHOL` $\rightarrow$ `SPECHOL`.

Acknowledgements: We are extremely grateful to the many suggestions for improvement from the reviewers of VoteID 2015. We have tried to take every comment into account, and have even used some of the suggested prose verbatim.

2 Hare-Clark Single Transferable Voting

Farrell and McAllister [12] provide a definitive study of preferential electoral systems in Australia. Wen [23] provides an engineering perspective of preferential systems, legislation and verifiable cryptographic schemes for preferential voting and counting. Here, we briefly describe STV and Hare-Clark STV.

STV for Electing Multiple Candidates. We assume that there are more candidates than seats, as otherwise, there is no need for an election. Voters order the candidates on the ballot paper in order of preference, usually by placing a number next to each candidate's name. To become elected, a candidate must reach a quota of votes, as opposed to an absolute majority. This quota is set according to the number of seats available. There are several ways of calculating a quota.

The votes are all initially allocated according to their first preference. A candidate who reaches the quota is elected, or else, if no candidate reaches the quota, then one "weakest" candidate is eliminated. There are several ways to choose the "weakest" candidate. If a candidate is elected by reaching the quota,

each surplus ballot for that candidate is transferred to the next continuing (un-elected and un-eliminated) candidate on that ballot. There are many different ways to choose a surplus ballot, and many ways to choose its new, possibly fractional, value. If a candidate is eliminated, all of the ballots currently counted as being for that candidate are transferred to their next (continuing) preference, again possibly with a fractional transfer value. The election is complete either when all seats are filled, or the number of vacant seats equals the number of continuing candidates, in which case all these candidates are elected.

The transfer of votes is key to ensuring that candidates with particular political views are elected in proportion to their support within the community, so the complexity resulting from surplus calculations and transfers cannot be removed without seriously crippling the system. As we shall see, there are many subtleties in the naive description above.

The Hare-Clark Scheme. Hare-Clark is an instance of the proportional representation scheme that uses single transferable vote as described above and has been used to elect members of Tasmania's House of Assembly since 1907 [17, 8]. A slightly different version has also been used to elect members of the Legislative Assembly in the Australian Capital Territory (ACT) since 1995 [3]. Hand-counting according to Hare-Clark is notoriously difficult and error-prone with some ballots examined in excess of 50 times before a result is declared. Thus a formally verified program for either version is likely to have practical benefits almost immediately. We already have a formal specification of Hare-Clark ACT [2], so we decided to concentrate on Hare-Clark Tasmania as this will allow us to compare and contrast the properties of these two variants of Hare-Clark.

3 Related Work

Various authors have attempted to apply formal methods to algorithms for STV counting, starting from early work using only pen-and-paper proofs, and ending with more recent work using light-weight computer-based tools. We present them in order of the amount and rigour of machine-checking involved in each. As far as we know, the only other work on using heavy-weight verification is our own previous work on reasoning about the first-past-the-post voting scheme [14].

Hill *et al* [15] give a pen-and-paper proof of various properties of an algorithm to count votes using the Meek's method. It's correctness relies totally on these pen-and-paper proofs, which presumably were checked by the referees.

Poppleton [19] takes a step towards machine-checked proofs by writing a specification for STV vote-counting in the logic-based specification language Z, but does not verify an implementation using computer tools based on Z [20].

Kiniry *et al* [10] formalised the STV scheme used for proportional representation elections in Ireland using the Alloy tool. They automatically generated test cases that covered every possible scenario using breadth-first search. Finally, they tested an implementation of the Irish vote-counting scheme, which had been developed using light-weight formal methods, and found two errors.

They conclude that *"this level of coverage (100% statement and condition coverage) does not prove that the system is error-free. ... But what it does do is (a) provide strong evidence, especially when combined with a rigorous development method and formal verification, that the system is correct, and (b) raise the state-of-the-art for election tally system testing enormously"* [10].

Cochran conducted a comprehensive study of verifying STV counting using light-weight (automatic) formal methods [9] by attempting to formally verify a Java program for the Irish proportional representation single-transferable voting scheme against its English natural language description using the ESC/Java tool. Most proofs were completed automatically, but in some, "*ESC/Java2 could neither verify the loop invariants nor the post-condition*" [9, p 46]. Moreover, lightweight formal methods, such as ESC/Java, are not guaranteed to be sound or complete since their code base is huge. Cochran concludes with "*Despite the use of a verification-centric process, and 100% statement coverage of the code, the following issues are outstanding, representing a potential inconsistency in the JML specifications.*" [9, page 63].

Recent attempts by Beckert et al [5] show that even state-of-the-art light-weight verification techniques such as bounded model-checking do not scale to realistic elections for even simple voting schemes such as first-past-the-post.

The move to using interactive theorem proving technology based upon (higher-order) logic is apparent in the work of De Young and Schurmann [11]. Rather than translating English prose into higher-order logic, they express the vote-counting scheme itself as a linear logic program. Read purely declaratively, this logic program specifies what the algorithm should do. It can also be executed to count actual ballots, although tests showed that it did not scale to real-world elections. The logical framework they utilise is not able to capture formal reasoning about the logic program itself: thus there is no correctness proof.

The related work described above is mostly about verifying algorithms against specifications. Thus there is no ability to formally compare and contrast two variants of the same voting scheme. Light-weight methods allow us to specify two variants of an STV voting scheme (say) and compare them by specifying different post-conditions. But recall that such tools are not guaranteed to be sound or complete. Recent work of Beckert *et al* [6] shows other pen-and-paper methods for reasoning about voting schemes using first-order and linear logic.

Our methodology goes beyond all of these efforts in the following senses:

Formal specifications: the specification is encoded as a formula of higher-order logic inside the HOL4 theorem prover. Thus it is type-checked and we can be sure that it actually is a well-formed formula of higher-order logic;

Formal termination: the SML program is encoded into HOL4 as `IMPHOL` and HOL4 will only accept the program if we can create a proof inside HOL4 that the program will terminate for all inputs;

Proof Objects: both the implementation and the specification are encoded as formulae of higher-order logic inside the theorem prover HOL4. Thus we can construct a proof that `IMPHOL` $\rightarrow$ `SPECHOL` which can be exported and checked by others using their own favourite theorem prover;

Correctness: the HOL4 theorem prover checks all steps in this proof are correct so we can be certain that the proof is mathematically correct.

Our methodology has three inherent weaknesses. As with all formal methods, there is no guarantee that `SPECHOL` correctly captures the English prose that makes up the Hare-Clark method of STV counting since it is merely one person's interpretation of the English-language prose of the relevant Parliamentary Act. We mitigated the risks of errors in interpretation by using two people to complete the formalisation: Meumann wrote the initial `SPECHOL` and `IMPHOL` but Dawson carried out all the proofs. Thus, Dawson first had to check whether these formulae accurately captured the Hare-Clark act and Meumann's implementation. In so doing, Dawson found some errors, as discussed in Section 8.3. Second, we have no formal model of the programming language SML, so we cannot prove that the final SML code meets its formal programming language semantics. As we point out previously [14], the CakeML [16] project will allows us to provide such proofs in the future. Finally, our approach is very labour-intensive: it took Dawson at least six months of full-time work to complete these proofs and he has over 20 years of experience in using higher-order logic theorem provers!

4 Higher-order Logic and the HOL4 Theorem Prover

The rigorousness of our approach stems from the use of HOL4 to construct the proofs. HOL4 is an interactive theorem proving assistant based upon Dana Scott's "Logic for Computable Functions" (LCF), a mathematically rigorous logic engine consisting of 8 primitive inference rules which have been proven to be mathematically correct [13]. HOL4 implements this logic engine using approximately 3000 lines of ML code and this code has been scrutinised by experts in LCF to ensure that it correctly implements the 8 inference rules. Any complex inference rules must be constructed as a programmatic combination of the core primitive rules only. Thus its code base is small and trusted.

Our verification process falls under the rubric of "heavy-weight verification" since it requires a person to direct the process in an interactive fashion. As such, it is very labour intensive. It involves producing a logical formalisation of both the program's requirements and the program itself in the HOL4 theorem proving assistant (`http://hol.sourceforge.net/`), then constructing a formal proof showing that the program matches the requirements. Producing the program using a strictly functional programming style ensures the program can be readily represented in higher order logic with minimal alterations. We used Standard ML (SML), the same language in which HOL4 is itself implemented.

When applied to electoral systems, the requirements are usually informed by the relevant legislation. As we shall show, translating complex legislation into rigorous formal logic can be a non-trivial task. Our methodology involves producing the following:

`SPECHOL`: a hand-encoding of the English-language description of the vote-counting process into higher-order logic;

`IMPHOL`: a hand-translation of `SPECHOL` into the HOL4 rendering of SML;
`IMP`: a hand-transliteration of `IMPHOL` into SML;
Formal Proof: a proof acceptable to the HOL4 theorem prover that `IMPHOL` logically implies `SPECHOL` which guarantees that the translation of the implementation meets the translation of the Parliamentary Act.

When applying this methodology to vote counting schemes, the counting program is represented in higher-order logic (as `IMPHOL`). It thus becomes possible to prove various results about the program. We can also verify various desiderata of the voting scheme (`SPECHOL`) itself. Our methodology is particularly suited to the verification of new voting schemes against the presence of desired properties or the avoidance of objectionable ones. For example it would be possible to prove that the voting scheme in question adheres to the independence of irrelevant alternatives (see [4]). It is also possible to prove comparative results between different voting schemes: for instance that voting scheme A differs from voting scheme B in only x specific situations.

The specification (in this case the translation of the legislation into HOL4's logic) is performed *prior* to the implementation of a counting program. This is intended to ensure the specification remains as independent of the implementation as possible. Thus ensuring any shortcuts or misconceptions adopted during the implementation process are not carried through to the specification.

Rather than producing an SML program and translating that to HOL4, the program is produced first in HOL4's formal logic, then translated to SML. Programming directly in HOL4's formal logic also helps to ensure that the non-functional features of SML are avoided.

The astute reader may notice there are certain gaps in this methodology that cannot be filled: there is no *proof* that the SML program (`IMP`) is the same as the HOL4 translation (`IMPHOL`), and there is no *proof* that the HOL4 encoding of the legislation (`SPECHOL`) is logically the same as the legislation itself.

5 Translating Legislation into Higher-order Logic

The following list of HOL4 syntax may be helpful.

HOL4	`\x y. A`	`T`	`F`	`~`t	t_1 `\/` t_2	t_1 `/\` t_2	t_1 `==>` t_2	t_1 `=` t_2	`!`$x.t$	`?`$x.t$
Logic	$\lambda xy.\ A$	verum	falsum	$\neg t$	$t_1 \vee t_2$	$t_1 \wedge t_2$	$t_1 \rightarrow t_2$	$t_1 = t_2$	$\forall x.t$	$\exists x.t$

The translation of the Tasmanian House of Assembly vote counting legislation is a non-trivial task. Theoretically, if there is only one way to interpret the legislation logically, then higher-order logic is expressive enough to capture the legislation's meaning. When examined closely, however, the legislation contains various ambiguities and contradictions that prevent a direct "translation". In many cases the intended *meaning* of the legislation must be encoded in HOL4 rather than a direct logical translation of each predication.

For example, clause 12 deals with the case in which there is a tie amongst the weakest candidates and one of them must be eliminated. The legislation specifies

that the tie is to be broken by deferring to *"the last count or transfer at which [the candidates involved in the tie] had an unequal number of votes"*. When more than two candidates are concerned, there are three different ways of interpreting which candidate should be excluded:

(a) the candidate who has the lowest count at the last count or transfer at which all of the candidates concerned had pairwise unequal counts;
(b) the candidate who has the lowest count at the last count or transfer at which one candidate had a count less than all of the other candidates concerned;
(c) the candidate who has the lowest count in a lexicographical ordering of all of the previous counts for the candidates concerned (with the most recent count being the first element of the lexicographical combination and the next-most recent count being the next element of the lexicographical combination etc.).

Option (a) appears to most closely mirror the wording of the legislation, but causes deferral to counts older than the other two options. This one is the most likely to defer all the way back to the initial count and result in a lot-based elimination. Option (b) causes deferral to counts more recent than option (a), but may result in the exclusion of a candidate who had a higher count than some or all of the other candidates concerned at a more recent count. Option (c) is the intuitively fairest option, but appears to reflect the legislation least. The ACT has a similar issue with their Hare-Clark legislation, in which the clauses regarding tie breaking are similarly worded. The ACTEC interprets their legislation according to option (c), so we also used this option.

Ambiguities such as this increase the difficulty of the formalisation process. Nevertheless, it is a testament to the rigorousness of our approach that it results in ambiguities such as this being discovered and properly questioned. This is a positive outcome if it results in a tightening of the legislation.

Another issue encountered whilst formalising the legislation is that the legislation is written in a procedural manner. In particular, the legislation makes regular reference to various "stages" of the count, and what should happen if certain conditions are met at various stages. This implies a mutable representation of the count, where the state changes over time (at each stage of the count) and is a side-effect of the legislation specifying *how* the votes should be counted, not what the result of the count should be. This is in direct contradiction with the ideal of functional programming, which is to have a declarative representation of computation (effectively stateless).

The procedural nature of the legislation forces `SPECHOL` to make statements about `IMPHOL`'s "state". In lieu of an existing `IMPHOL`, the `SPECHOL` must be built based on assumptions about `IMPHOL`'s structure. This results in a certain level of coupling between `SPECHOL` and `IMPHOL`, but cannot be avoided when the legislation is written in a procedural manner.

5.1 Assumptions About the Implementation

To have a concrete conceptualisation about which to build the logical statements of `SPECHOL`, some assumptions must be made about the form of `IMPHOL`. The

initial assumptions are explained below. Note that some of the assumptions now need revision due to unforeseen technical restrictions on `IMPHOL`. The revision process is yet to be undertaken, but the intention is to combine it with a general review of `SPECHOL` to remove any inconsistencies.

Inputs and Outputs. The inputs and outputs of the counting procedure must be defined. This is fairly straightforward. At a minimum, the procedure must take the set of ballots and the set of running candidates as input. These are assumed to be provided using lists: a mainstay of functional programming. The procedure is assumed to take as input a list of candidates and a list of ballots. Each ballot itself is assumed to be a list of candidates in order of preference (the head of the list being the first preference). It is also assumed the function takes as input the number of candidates to be elected since the number of seats per electorate has changed multiple times in Tasmania. The output of the function is assumed to be a list of elected candidates. Let us call the function `COUNT_HCT`, so we have the following type-definition to work with:

```
COUNT_HCT: num -> 'a list -> 'a list list -> 'a list
```

where the first argument represents the number of available seats, the second argument the list of running candidates and the third argument the list of ballots.

Stateful Representation. Some assumptions about the internal operation of the function are needed to capture the stateful or procedural nature of the legislation. It is necessary to assume that `COUNT_HCT` possesses some form of state, and that the state changes over time. Moreover the state must take a particular structure, so we can reason formally about its various components.

In a strictly functional programming language there is no implicit concept of time or state. The closest thing to a state is the set of values of all of the variables at a given level of recursion. In this conceptual representation of state, the "time" is given by the level of recursion. Naturally, a proper representation of time must be strictly monotonic. That is, with each recursion the time must increase. In other words, backtracking back up the recursion cannot be permitted until the final result is ascertained (and it becomes possible to backtrack all the way to the surface tail-recursively). Ultimately, within the Hare-Clark context, our concept of time need only capture the temporal difference between the stages of the count, not the assignment of individual variables or other small differences. Bundling the requisite variables into a "state" represented by a tuple allows us to recurse on the tuple and treat it as a close approximation of a mutable state.

Based on the properties referred to by the legislation, it is assumed that the state of the count is represented by a tuple of the following structure:

```
(time, seats, quota, elected, excluded, rem, surps, groups)
```

where...

`time` is a parameter representing temporality (the level of recursion). It increases by one with each recursion;

`seats` is the number of seats to be filled. Note that this value is not intended to change over the course of the count;
`quota` is the number of votes required by candidates in order to be declared elected. It is calculated at the beginning and remains unchanged throughout;
`elected` is a list of candidates who have been elected. Declaring a candidate elected (as specified in the legislation) means placing a candidate in this list;
`excluded` is the list of excluded candidates. Excluding a candidate is interpreted as placing a candidate in this list;
`rem` is the list of continuing candidates, along with their current vote counts and their transfer history. Each candidate in this list is represented by the tuple `(name, total, transfers)` where:
`name` is the identifier of the candidate (this can be any equality type);
`total` is the total value of votes assigned to the candidate;
`transfers` is a list of transfers assigned to the candidate and is of the form `(value, ballots, clause)` where
`value` is the transfer value associated with the transfer and is a tuple of the form `(numerator, denominator)`;
`ballots` is the list of ballots associated with the transfer.
`clause` represents the clause responsible for the transfer of the ballots to the candidate concerned. This will likely be removed when the specification is reviewed as the implementation does not use it;
`surps` is a list of pending transfers of surplus votes from elected candidate;
`groups` is a list of transfers pending from the exclusion of a candidate. Each member of both `surps` and `groups` is of the form `(value, ballots)` where `value` is a tuple `(numerator, denominator)` for the transfer value and `ballots` is the list of ballots awaiting transfer.

Assuming the function performing the recursion on the state tuple is called `FINAL_STAGE`, we have the following function type definition:

```
FINAL_STAGE: num # num # num # 'a list # 'a list #
   ('a # num # ((num # num) # 'a list list # num) list) list #
   ((num # num) # 'a list list # num) list
   -> 'a list
```

It can be argued that these assumptions are not necessary: that the functions can be quantified in each of the clauses. This would remove any dependency on naming conventions, but the clauses will still need to make statements relying on what form the functions take. This has the potential to blow out the complexity of the individual clauses as each clause will need to cover many more possibilities in terms of functional structure. Whether or not this would actually happen is unclear. Potentially, more experience is needed to truly take advantage of the expressibility of higher order logic.

With the assumptions in hand, it becomes possible to translate the legislation into HOL4's formal syntax. The translation of one example clause, is given in Section 5.4. An example function and statements that are used by the clauses are given in Section 5.2 below. Sanity checks are given in section 5.3. Note that

the definitions, sanity checks and clausal statements will need to be revised to take into account the final form of the implementation. They will also need to be reviewed for their accuracy.

5.2 Example Definitions

The function shown in Listing 1.1 is used to simplify the clauses in Section 5.4. Such functions are intended to be executable and translatable into SML so that they may be used by the counting program should this be necessary.

Listing 1.1: Executable function definitions.

```
(* Returns list of ballots whose first preference is cand *)
val FIRSTS_FOR_DEF = Define `
    FIRSTS_FOR cand ballots =
        FILTER (($= cand) o HD) ballots`;
(* Sums the number of ballots with a first preference for
   each of the running candidates.  This is needed simply
   because the legislation specifies that this is how the
   quota should be calculated. *)
val SUM_FIRSTS_DEF = Define `
    (SUM_FIRSTS []      ballots = 0)
 /\ (SUM_FIRSTS (c::cs) ballots =
                            LENGTH (FIRSTS_FOR c ballots)
                            + SUM_FIRSTS cs ballots)`;
```

5.3 Sanity Checks

In addition to the clauses of the Tasmanian Hare-Clark legislation, some proof obligations have been defined as sanity checks. These checks can be assumed in the clausal statements, reducing their complexity, as shown next.

Listing 1.2 specifies that it must be impossible to introduce candidates to the list of continuing candidates after the count has begun. In other words, if a candidate is in the list of continuing candidates, then that candidate must have been in the list of candidates in all preceding states of the count.

Listing 1.2: Candidates cannot be introduced partway through the count.

```
!seats cands ballots state state'.
    (COUNT_HCT seats cands ballots = FINAL_STAGE state)
    /\ (COUNT_HCT seats cands ballots = FINAL_STAGE state')
    /\ TIME_VAR state' > TIME_VAR state
    ==> !cand. IS_REM_CAND state' cand
               ==> IS_REM_CAND state cand
```

5.4 Example of a Clause in Higher-order Logic

Each of the 14 clauses in the Tasmanian Hare-Clark legislation was thus hand-translated into higher-order logic. We give just one example below.

Clause 2: First preference votes to be counted

> The number of first preferences recorded for each candidate, on ballot papers which are not informal ballot papers, is to be counted.

This is somewhat abstract in the context of our counting procedure and leaves little to specify concretely. The proof obligation for this clause in HOL4 instead specifies how the counts should be incorporated into the initial state tuple. See Listing 1.3 below.

Listing 1.3: Clause 2

```
!seats cands ballots cand rem_cands quota.
 (COUNT_HCT seats cands ballots =
           FINAL_STAGE (t0,seats,quota,[],[],rem_cands,[]))
 /\ (MEM cand cands =
        MEM (cand,
             LENGTH (FIRSTS_FOR cand ballots),
             [((1,1), FIRSTS_FOR cand ballots, clause2)])
            rem_cands)
```

Note that the "count" of the first preferences is given by `LENGTH (FIRSTS_FOR cand ballots)`. The function `LENGTH` is a predefined function in HOL4, and `FIRSTS_FOR` is defined in Listing 1.1 and `MEM` is the member predicate on lists.

6 From HOL4 to an SML Implementation

The implementation is first written in HOL4, then translated into SML. The translation is performed iteratively, ironing out any features used in one language that are not available in the other. Since the implementation is initially programmed in HOL4, the features that need removal are primarily those available in HOL4 but not SML (a lambda calculus interpreter for instance).

The semantic equivalence of these two implementations is not rigorously guaranteed. A visual comparison is still convincing for this larger case study, however, thanks to the strict functional nature of the implementations and the restricted feature set they use.

The implementation breaks ties using the lexicographical ordering interpretation (option (c) on page 7). It does this by merge-sorting the list of remaining candidates according to candidate counts at each stage of the recursion. Merge sort is stable, allowing it to maintain the lexicographical ordering discussed without further interference.

6.1 Testing the SML Implementation for Efficiency

The implementation was tested both for preliminary correctness and to ascertain whether it could handle the input sizes likely in real public elections. All of the tests were conducted using PolyML (`http://www.polyml.org/`) on GNU/Linux with an Intel Core i7-3740QM processor and 16 GiB of RAM.

To test for bugs, we compared this implementation against an implementation of the ACT's Hare-Clark system produced previously by Dawson. Several randomly generated examples were produced with lists of votes ranging from 50 thousand to 300 thousand in length and between 10 and 40 candidates covering a range of possible scenarios.[1] The two programs produced the same results for each example, giving preliminary indications that the program is correct.

However, there are differences between Hare-Clark ACT and Hare-Clark Tasmania. The main difference that might lead to a different outcome at an election is that the transfer value is calculated differently. Tasmanian Hare-Clark calculates the transfer value based on the total number of votes in the transfer leading to the surplus whereas the ACT calculates it based on a subset of those: the unexhausted ballots. The following small example illustrates the difference.

Imagine an election between 3 candidates (A, B and C), with two available seats and a total of 5 votes. Let's say the votes were as follows: `[A,B] [A,B] [A] [A] [C]` where the vote can be read from left to right in order of preference (so the first vote has A as its first preference and B as its second). In the first round, A will be elected with 4 votes and a surplus of 2 (the Droop quota is 2 votes). B and C remain unelected with counts of 0 and 1 respectively. In the second round, 2 of A's votes will be counted towards B, but the transfer value differs between the ACT and Tasmanian systems:

TAS = (surplus / total votes in prev. transfer) = 2 / 4 = 1/2
ACT = (surplus / total unexhaust. votes in prev. transfer) = 2 / 2 = 1

Thus,the ACT would transfer the votes in full, whereas in Tasmania they would transfer them as half votes. So the result after round 2 would be:

TAS: 1 for C, 1 for B so B eliminated as C had more votes in previous round;
ACT: 1 for C, 2 for B so C eliminated.

The programs confirmed that this example leads to different results.

The program was tested separately for its ability to handle large numbers of ballots. The tests ranged from 250 thousand votes with 10 candidates to 15 million votes with 40 candidates. Note that every example took less than 80 seconds to count, and consumed less than 10 GiB of memory.

There are 5 electorates in Tasmania used to elect the House of Assembly using Hare-Clark, and these each have approximately 72,000 enrolled voters (as

[1] If a large number of ballots are generated naïvely, they become spread too evenly between the candidates. This results in no candidate being elected until the final stages of the count, which is unrealistic. The candidates were given random popularity ratings to produce uneven distributions of ballots to avoid this issue.

at September 2013) [21]. Our implementation is more than adequately equipped to handle counts of this size. The largest electorate used in any PR election in Australia is New South Wales (NSW), with an enrolment of just under 5 million voters (as at August 2014) [18]. Once again, our implementation is well able to handle counts of this size.

An initial analysis shows that our SML code has computational complexity O(num_candidates * num_candidates * num_votes), possibly worse. We are confident that we can remove at least one of these occurrences of num_candidates by using SML arrays, and setting up a HOL4 theory formalising the appropriate extra correctness properties. An alternate view is that the verified code only has to be run once, and it doesn't matter if it takes a week to run, even if the faster unverified code has already produced a result which has been announced.

We are currently investigating whether it can handle hundreds of candidates as occurred in the 2015 NSW State Election. Incidentally, being functional, and moving bits of data all over the place, it depends crucially on real memory.

7 Proving Termination of Functions and Properties of the Results of those Functions

HOL4 requires function definitions to terminate, because the underlying logic of computable functions requires that all functions be total. So HOL4 does not actually allows us to state termination as a formula of higher-order logic: rather the evidence of it is that HOL accepts the definition of a function.

Once we input a function definition, HOL4 automatically attempts to generate a termination proof using in-built strategies based upon term-rewriting. If HOL4 cannot produce a termination proof automatically, it outputs the statement of a lemma which would allow it to complete the proof. If the user proves the lemma interactively, then HOL4 completes the proof of termination itself.

Once HOL4 accepts a function as terminating, it outputs, automatically, an induction principle which can be used to prove an arbitrary property `P` of the function. By instantiating this property in various ways, we can prove interesting properties of the function as illustrated next.

7.1 Properties of the function `MERGE`

Definition 1 (`MERGE_def`)**.** *The function* `MERGE` *(used to define* `MERGE_SORT`*) is*

```
    (MERGE R []      right   = right)
 /\ (MERGE R left    []      = left)
 /\ (MERGE R (l::ls) (r::rs) = if R l r
                               then l::(MERGE R ls (r::rs))
                               else r::(MERGE R (l::ls) rs))
```

where `[]` *is the empty list,* `x::xs` *is a list with head* `x` *and tail* `xs` *and* `R` *is a function that returns* `true` *if its first argument is "less than" its second.*

Theorem 1. *The* `MERGE` *function terminates for all inputs*

Proof. HOL4 is able to deduce termination automatically because in successive recursive calls, one list argument gets smaller while the other remains the same.

Some function definitions, however, require the user to prove a termination condition: in general, that there is some well-founded relation for which the argument(s) to the function get "smaller" in successive function calls.

HOL4 generates, automatically, an induction principle (lemma) called `MERGE_ind` for proving properties `P` of the result of the `MERGE` function:

Lemma 1 (`MERGE_ind`). *For all properties* `P`*, if the following conditions hold*

1. `P R left right` *holds whenever* `left` *or* `right` *is empty*
2. `P R (l::ls) (r::rs)` *holds whenever* `~ R l r` *and* `P R (l::ls) rs` *hold*
3. `P R (l::ls) (r::rs)` *holds whenever* `R l r` *and* `P R ls (r::rs)` *hold*

then `P v v1 v2` *holds for all values of* `v`*,* `v1` *and* `v2`*:*

```
!P. (!R right. P R [] right) /\ (!R v4 v5. P R (v4::v5) [])
 /\ (!R l ls r rs. ~ R l r /\ P R (l::ls) rs ==> P R (l::ls) (r::rs))
 /\ (!R l ls r rs. R l r /\ P R ls (r::rs) ==> P R (l::ls) (r::rs))
==> !v v1 v2. P v v1 v2
```

Note how HOL4 has reformulated the first clause of `MERGE_ind` to avoid overlapping cases by using `v4::v5` instead of `left` to enforce that the left argument is a non-empty list since the other part of this clause already handles the case where `left` is the empty list.

As an example of a proof by induction using `MERGE_ind`, we prove that the result of `MERGE`, viewed as a set, is the union of the lists `l` and `r`, viewed as sets.

Theorem 2. `!v v1 v2. set (MERGE v v1 v2) = (set v1) UNION (set v2)`

Proof. We instantiate `P` of the theorem `MERGE_ind` to

```
\c l r. set (MERGE c l r) = set l UNION set r
```

HOL4 then sets out the framework for a proof by induction, where the inductive steps and their assumptions match the structure of `MERGE_def` (Definition 1). Intuitively, each step in the definition of `MERGE` preserves the desired property.

8 Proving Sanity Checks, Difficulties and Errors Found

8.1 That the list of candidates remains unchanged.

We showed that the list of elected, excluded and remaining candidates is unchanged. This needs to be formulated precisely, since these lists are changed by moving candidates from one list to another, and by re-ordering the remaining candidate list according to the number of votes each candidate has.

We use the built-in function `PERM`, which means that one list is a permutation of the other. The definition of `PERM` is provided by HOL. So we show that at each iteration, the concatenation of these lists is permuted.

8.2 Conditions which need to be proved

These are examples of conditions which seem obvious, and are assumed by the code (and, indeed, by the legislation), but proving that they hold requires several steps of reasoning and tracing through the code. Their proof is a lower priority since whenever they are not satisfied, the code as written will not complete without error, but for completeness, we intend to prove all such conditions.

The condition that there be "remaining" candidates. Since the counting program chooses the lowest ranking candidate to be eliminated, it requires that the list of "remaining" candidates be non-empty. We found that to prove this from the code as written would be very convoluted, since the part of the code which excludes a candidate requires that there be a candidate to exclude, and will have an undefined effect otherwise. That is, to avoid reasoning about an undefined effect, we have to prove that the list of remaining candidates is non-empty: leading us back to where we started!

We "solved" this problem by adding an extra termination condition: stop if the list of remaining candidates becomes empty. This avoids having to reason about undefined effects, but defers the problem since we now need to prove that this extra termination condition never has effect. However, doing so is significantly simpler since we never have to reason about undefined effects.

That transfer values do not have denominator zero. The code requires the denominator of a fractional transfer value to be non-zero. This in turn requires that the final parcel of votes which elects a candidate is non-empty and that a candidate can get only one new parcel of votes in each iteration of the algorithm.

8.3 Errors discovered

We found some errors where conditions (expressed in HOL4), which we set out to prove, were in fact not provable. We have not yet found cases where this was due to errors in the code (that is, the specification, in HOL4, of the program's behaviour). Rather, the errors were all in the expression of the conditions which were to be proved. We surmise that this is because the "program" specification, in HOL4, was translated into Standard ML, and tested. No doubt there were errors which were found in the course of this testing. On the other hand, the correctness conditions were not tested in this way.

Taking the n^{th} member of a shorter list. The condition that the list of remaining candidates are distinct utilises the function `EL n list` which returns the n^{th} member of `list` but is undefined when `list` has fewer than n members.

Need to assume candidates distinct initially. To prove that the list of remaining candidates are distinct at any stage, it is necessary to assume that the list of candidates provided initially is distinct. This assumption was omitted.

References

1. AAP. AEC costs WA Senate election at $20M. http://www.sbs.com.au/news/article/2014/02/25/aec-costs-wa-senate-election-20m, February 2014.
2. P. Abate, J. Dawson, R. Goré, M. Gray, M. Norrish, and A. Slater. Formal methods applied to electronic voting systems. http://users.rsise.anu.edu.au/~rpg/EVoting/, 2003.
3. ACTEC. Hare-Clark electoral system. http://www.elections.act.gov.au, 2015.
4. K. J. Arrow. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58(4):pp. 328–346, 1950.
5. B. Beckert, T. Börmer, R. Goré, M. Kirsten, and T. Meumann. Reasoning about vote counting schemes using light-weight and heavy-weight methods. In *VERIFY 2014: Workshop associated with IJCAR 2014*, 2014.
6. B. Beckert, R. Goré, C. Schürmann, T. Bormer, and J. Wang. Verifying voting schemes. *J. Inf. Sec. Appl.*, 19(2):115–129, 2014.
7. J. Benaloh, T. Moran, L. Naish, K. Ramchen, and V. Teague. Shuffle-Sum: Coercion-resistant verifiable tallying for STV voting. *Information Forensics and Security, IEEE Transactions on*, 4(4):685–698, Dec 2009.
8. S. Bennett. Inglis Clark's other contribution: A critical analysis of the Hare-Clark voting system. http://samuelgriffith.org.au/docs/vol23/vol23chap5.pdf.
9. D. Cochran. *Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms*. PhD thesis, ITU, 2012.
10. D. Cochran and J. Kiniry. Formal model-based validation for tally systems. In *Proceedings of VoteID 2013*, volume LNCS 7985, pages 41–60. Springer, 2013.
11. H. DeYoung and C. Schürmann. Linear logical voting protocols. In *E-Voting and Identity*, pages 53–70. Springer, 2012.
12. D. M. Farrell and I. McAllister. *The Australian Electoral System: Origins, Variations and Consequences*. University of New South Wales Press, 2006.
13. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. CUP, 1993.
14. R. Goré and T. Meumann. Proving the monotonicity criterion for a plurality vote-counting program as a step towards verified vote-counting. In *6th International Conference on Electronic Voting: Verifying the Vote*, pages 1–7, 2014.
15. I. D. Hill, B. A. Wichmann, and D. R. Woodall. Algorithm 123 : Single transferable vote by Meek's method. *Computer Journal*, 30:277–281, 1987.
16. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL*, pages 179–192, 2014.
17. T. Newman. Hare-Clark system. http://www.utas.edu.au/library/companion_to_tasmanian_history/H/Hare-Clark%20system.htm, 2004.
18. NSWEC. Enrolment statistics. http://www.elections.nsw.gov.au/enrol_to_vote/enrolment_statistics, 2014. New South Wales Electoral Commission.
19. M. Poppleton. The single transferable voting system: Functional decomposition in formal specification. In *IWFM*, 1997.
20. Community Z tools. http://czt.sourceforge.net/, Accessed 2 June 2015.
21. TEC. Annual report 2013–2014. http://www.tec.tas.gov.au/pages/ElectoralInformation/PDF/2012-13\%20TEC\%20Annual\%20Report.pdf, 2013. Tasmanian Electoral Commission.
22. V Teague and J A Halderman. Thousands of NSW election online votes open to tampering. http://theconversation.com, 2015.
23. R. Wen. *Online Elections in Terra Australis*. PhD thesis, University of New South Wales, 2010.