EasyChair Preprint

№ 15595

Reasoning About Vote Counting Schemes Using Light-Weight and Heavy-Weight Methods

Bernhard Beckert, Thorsten Bormer, Rajeev Gore, Michael Kirsten and Thomas Meumann

December 18, 2024

Reasoning About Vote Counting Schemes Using Light-weight and Heavy-weight Methods

Bernhard Beckert[1], Thorsten Bormer[1], Rajeev Goré[2],
Michael Kirsten[1], Thomas Meumann[2]

[1] Dept. of Informatics, Karlsruhe Institute of Technology (KIT)
[2] Research School of Computer Science, The Australian National University

Abstract

We compare and contrast our experiences in specifying, implementing and verifying the monotonicity property of a simple plurality voting scheme using modern light-weight and heavy-weight verification tools.

1 Introduction

Most research in computer-supported voting focuses on the process of vote casting. But the dependability and trustworthiness of vote counting is just as important – if not more so. In this paper, we focus on the problem of verified vote-counting whereby the code used to count the votes is formally verified against its specification. In countries which use complex single-transferable voting schemes, such as Australia and Ireland, this question is of paramount importance.

Humans are notoriously error-prone when counting votes using these complex schemes, and most electoral laws deliberately simplify the theoretical vote-counting scheme to make it easier for human counting. The ability to count votes using computers opens up the possibility to design new, even more complex, voting schemes which guarantee various theoretical desiderata, and to use them in real elections. How can we be sure that the new schemes enjoy the desired properties while remaining practical for counting by computer for large numbers of votes?

One way to ensure these properties is to develop the voting scheme incrementally and iteratively. We start with a simple implementation and specification and gradually add complexity as we iron out errors in the implementation and specification, and gain insights into the practicality of the desired theoretical desiderata. So what form of verification should we use?

Available options range from light-weight, fully automatic methods like software bounded model checking (SBMC) to full functional software verification using annotation-based program verification tools or interactive higher-order theorem provers.

Both SBMC and annotation-based program verification tools involve adding the properties to be checked as pre and post condition annotations to the actual code, turning these annotations automatically into proof obligations by a compiler, and discharging the proof obligations automatically by some theorem prover. Note that we are interested in high-level theoretical properties of voting schemes, not low-level correctness properties for which SBMC is normally used, such as the absence of runtime exceptions, buffer overruns etc.

While the light-weight SBMC technique does not need additional user interaction to check properties of a program, it only allows to check properties for a small portion of the program inputs. A *universal* correctness property can instead be derived using, e.g., annotation-based deductive program verification tools like VCC. Here, additional information such as intermediate annotations is indispensable and must be provided by the user for the successful proof search.

Heavy-Weight verification involves encoding both the implementation and the specification into the logic of some theorem prover, and then proving that the encoding of the implementation implies the encoding of the specification using that theorem prover, usually interactively.

The difference in these methodologies means that each has its pros and cons, and it is not at all obvious which is the better methodology to follow.

Here, we outline our experiences in specifying, implementing and verifying the monotonicity property of a simple plurality voting scheme using modern "light-weight", "medium-weight" and "heavy-weight" methods. The light-weight verification method is to apply the automatic software model checking tool LLBMC [12] to a C implementation annotated with a simple assertion-based representation of the monotonicity property. The "medium-weight" method of auto-active verification is where we apply the deductive program verification tool VCC, using a first-order logic specification of monotonicity. The heavy-weight path consists of an ML implementation, an encoding of the ML implementation into higher-order logic, an encoding of the monotonicity property in higher-order logic, and interactive proofs in the HOL4 theorem prover that the encoding of the implementation obeys the encoding of the monotonicity property.

2 Correctness of Voting Schemes

Voting schemes are difficult to get right. Even when the desired correctness properties are clear in our minds, tailoring an existing voting scheme to meet the new correctness properties may easily produce unintentional side-effects – a recent example being the voting scheme to elect the board of trustees of the International Conference on Automated Deduction [6].

We argue that in designing voting schemes, we need both tailor-made properties for the particular scheme that capture the intended qualities, as well as a range of verification techniques for the different development stages: "light-weight" formal methods for early feedback for initial implementation prototypes to guide us in the right direction, complemented by "heavy-weight" verification techniques to ensure that the final version of the voting scheme is correct. Between the two is a medium-weight approach which can take both roles, but we do not know yet whether it is good for one of them, both, or none.

Voting schemes are an interesting target for verification as they are amenable to a wide range of formal methods: the implementation of a typical voting scheme is comparatively small and relevant correctness properties can be formalized using first order logic with theories. Also, the program execution is often highly symmetrical,with the potential to be exploited by automated software verification tools.

For the rest of the paper we have chosen a simple voting scheme, together with a correctness criterion as running example, with the advantage that the functional behavior for the different parts of the voting scheme is well understood and can be easily specified. This allows us to compare a range of different verification methods.

A Simple Plurality Voting Scheme First-past-the-post plurality voting is a voting scheme wherein each voter may vote for one candidate only, usually by marking a cross or a tick next to the desired candidate on the ballot paper. The number of votes for each candidate is tallied, and the candidate with the most votes (a relative majority) is declared elected. Note that the candidate does not need an absolute majority. Real-world voting systems vary in the way they deal with a tie, but in our simple case, no candidate is elected in the case of a tie.

This is a simple version of one of the most comprehensible and widely-used voting schemes and the vote counting process is easily observable as it holds only two simple procedures. These are firstly the summation of the votes cast for each candidate, and secondly the determination of the candidate with the highest aggregation of votes. More sophisticated versions of this voting scheme might have some additional procedure in order to break a tie. Not only is the voting scheme easily observable, but generated counterexamples are also easily verifiable. The

generation of counterexamples (in the case of properties not fulfilled by the voting scheme) is part of the light-weight verification techniques employed within this work.

The Monotonicity Criterion The monotonicity criterion was originally posited by Arrow as a desirable property of social welfare functions as follows [2]:

> "If an alternative social state x rises or does not fall in the ordering of each individual without any other change in those orderings and if x was preferred to another alternative y before the change in individual orderings, then x is still preferred to y."

A social choice procedure, such as a voting scheme or a market mechanism, can be said to either satisfy this condition or not. Reducing the available social choice procedures to preferential voting schemes or a subset thereof allows us to narrow the definition and put it in more tractable language. Thus for our purpose: "social state" is the election of a particular candidate; and "x is preferred to y" refers to a societal preference and can be changed to "x is elected".

In our plurality system, voters may only vote for one candidate, ie. rank one candidate above all others (rejecting all others equally). Thus monotonicity can be rewritten as:

> If each voter either changes his or her vote to a vote for candidate x or maintains his or her vote unchanged, and if x won before any votes changed, then x will still win after the changes.

3 Bounded Verification of Plurality Voting

Here, we compare two different software verification approaches working on the source code level of a particular voting scheme. The first technique, software bounded model checking (SBMC), is able to show correctness properties for program runs up to a certain length fully automatically without user support. As the length of program runs analyzed this way is bounded, of course no general correctness guarantee can be derived so we rely on the small scope hypothesis [16] to argue that it is unlikely that a large number of bugs has escaped this analysis.

In contrast, the second technique, annotation-based deductive verification, is able to provide general correctness properties. With tools following this paradigm, the requirement specification is usually given in the form of method contracts and invariants on data structures. Correctness of the program w.r.t. this specification is then established without further interaction, provided enough *auxiliary annotations* are given by the user beforehand – this kind of interaction pattern is also called *auto-active*. As providing the right annotations for these deductive verification systems is time consuming or even infeasible in some cases, as a third option, we try to combine the advantage of full automation of SBMC and the possibility of abstracting from the implementation using annotations in a kind of modular bounded verification approach.

Verification Setup For bounded program checking, the tools used in our case study take C programs (annotated with specifications) as input. Fig. 1a shows the C implementation of the majority voting scheme. Given an array of size `V` containing the votes where each vote is represented by an integer between 1 and `C`, first the vote count for each candidate is computed in the for-loop in line 6. If the resulting array `res` contains a unique maximum (the for-loop starting in line 10), the function `voting` returns the corresponding candidate number, and zero otherwise. The variables `V` and `C` are parameters of the implementation and their values are set later in the experiments depending upon the capabilities of the verification tool used.

```
uint voting(uint votes[V]) {
  uint res[C + 1];
  for (uint i = 0; i <= C; i++)
    res[i] = 0

  for (uint i = 0; i < V; i++)
    res[votes[i]] = res[votes[i]] + 1;

  uint max = 0, elect = 0;
  for (uint i = 1; i <= C; i++) {
    if (max < res[i]) {
      max = res[i];
      elect = i;
    } else if (max == res[i])
        elect = 0;
  }
  return elect;
}
```

(a) C implementation of the majority voting scheme

```
void monotonicity(uint v1[V],
                  uint v2[V],
                  uint m, uint n) {
  //pre: v1[n] ≠ m ∧ v2[n] = m ∧
         ∀i. 0 ≤ i < V ∧ i ≠ n
           → v1[i] = v2[i]
  uint elect1 = voting(v1);
  uint elect2 = voting(v2);
  //post: m = elect1 → elect1 = elect2
}
```

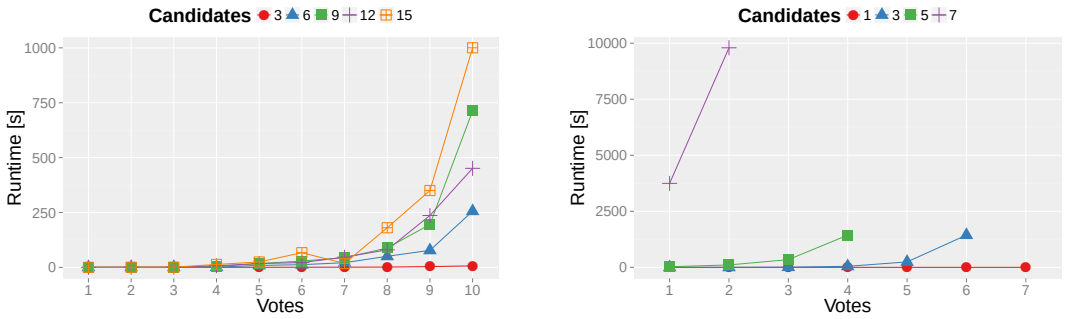
(b) Encoding the monotonicity property

Figure 1: C verification setup

The desired monotonicity property compares two executions of `voting`, so we introduce a helper function in Fig. 1b, which is later annotated with the appropriate instance of the contract given in comments marked `pre` and `post`. To simplify the verification task, we use a different representation of the monotonicity criterion from Sec. 2: whereas the original property allows *any number* of votes to change in favour of the the winning candidate, we show that monotonicity holds when only *one* vote is changed to favour the winner. We also introduce variables `m` and `n` for the values of the vote at position `n` in the array `votes` that is changed to count for candidate with number `m`. These variables are then used in the contract for `monotonicity`. As no concrete values for `n` and `m` are fixed in the precondition of `monotonicity`, the resulting correctness proof holds for arbitrary assignments to these variables (as long as they denote valid candidates resp. votes).

3.1 Software Bounded Model Checking

The software tool employed for bounded verification within this work is the bounded model checker LLBMC. It is actively developed by the research group *Verification meets Algorithm Engineering* at Karlsruhe Institute of Technology (KIT) with the main purpose of finding bugs and runtime errors in sequential C/C++ programs. As such, LLBMC does not directly support expressions in full first order logics, but programs can be specified with any legal C/C++-expression of type boolean as assumptions and assertions. The memory model is bit-precise and it operates on an intermediate compiler representation instead of directly on the C source code. It then converts this intermediate representation into a logical representation including some simplification steps by rewrite rules. For the actual verification process, it uses an SMT solver for the theory of bitvectors and arrays. By default, this is set to the SAT solver MiniSat [10] supplemented by the decision procedure STP [13]. LLBMC does bounded verification by unrolling loops in the given program and exploring all its possible execution



(a) LLBMC (b) VCC (all loops unrolled)

Figure 2: Verification performance checking monotonicity for Plurality Voting (bounded).

branches up to a given bound. This bound is determined by parameters to bound the numbers of loop unrollings and invocations of recursive functions. It achieves competitive results in its field and participated successfully in major competitions on software verification in various categories.

Using the software bounded model checking approach, C function `voting` does not have to be annotated for verification – the only user interaction necessary is the representation of the contract of `monotonicity` using specification primitives of LLBMC (i.e., simple assertions and assumptions about the state of the program). Some of the preconditions in our example are valid C expressions and can be adopted one-to-one as LLBMC specification statements – other specification constructs containing logical connectives or special keywords not present in C expressions have to be translated with the help of C statements (e.g., implications can simply be mapped using if-statements, together with LLBMC assertions). Quantified formulas over a bounded domain, as e.g., in the precondition of the monotonicity function are replaced by a conjunction of the instances of the quantified formula for each value of the bound variable (which can be succinctly expressed using a C for-loop).

Experiment 1. For the implementation from Fig. 1a, LLBMC is able to show that majority voting satisfies the monotonicity criterion up to 15 candidates and 10 votes (within 1500s using at most 4 GB of memory). The run-times of the tool depending on the number of candidates and votes are shown in Fig. 2a – the graphs indicate an exponential time complexity in the number of votes. Curve fitting for the run-times plotted against the number of candidates (not shown here) suggests an at most quadratic time complexity in the number of candidates.

3.2 Deductive Verification

Having gained confidence that the voting scheme is correct for a small number of candidates and votes using bounded model checking with LLBMC, the next step is to prove the desired properties for larger bounds or even in the unbounded case. One appropriate formal method for finding general correctness proofs is deductive program verification, via tools such as VCC.

VCC [7] aims for full modular verification of concurrent C programs. It uses the intermediate verification language Boogie [3] and the SMT solver Z3 [8] with integer arithmetic. Modular verification is possible using the *Design by Contract* approach relying on contracts with pre- and

```
1  void monotonicity(_(ghost int v1[int]) _(ghost int v2[int])
                     _(out int elect1) _(out int elect2)
                     int m, int n)
       _(requires 0 < V && V < INT_MAX-1)
5      _(requires 0 < C && C < INT_MAX-1)
       _(requires \forall int i; (valid_voter(i) ==> valid_cand(v1[i]))
                              && (valid_voter(i) ==> valid_cand(v2[i])))
       _(requires valid_cand(m) && valid_voter(n))
       _(requires v1[n] != v2[n])
10     _(requires v2 == \lambda int i; i == n ? m : v1[i])
11     _(ensures (m == elect1) ==> (elect1 == elect2))
```

Figure 3: VCC contract for `monotonicity`.

post- conditions, state assertions, loop and type invariants as well as some specification defining the contract's framing (i.e. which memory locations can be changed). These annotations are mostly written in first order logic and other statements possible in C programs. VCC then attempts to prove the correctness of these annotations w.r.t. the program for every possible program execution. As this verification is in general undecidable, VCC sits within a larger framework that includes tools for monitoring proof attempts, tracking and examining failed proofs as well as generating partial counterexamples.

Examples for auxiliary annotations for imperative programming languages are loop invariants or (non-requirement) contracts of called methods. Finding sufficient auxiliary annotations is still non-trivial, despite the advances in the automation of the underlying proof search strategies as part of the verification tool-chain. The verification engineer must also consider the strengths and weaknesses of the tool at hand when choosing the form of auxiliary specification.

Even for an established specification and corresponding implementation of a voting scheme that is correct w.r.t. this specification, many iterations of adapting auxiliary annotations of different parts of the program are necessary [5] (e.g., especially in the case of loop invariants for nested loops or for contracts of nested method calls).

Reducing the requirement specification from full functional correctness to simpler, tailor-made properties of the program often does not simplify or reduce the number of auxiliary annotations needed – still, functional correctness of most of the program is likely to be necessary to verify the properties in question. For more complex voting schemes than plurality voting, the annotation-based verification approach is thus unlikely to scale.

One formulation of the contracts specifying the monotonicity property (as introduced in Fig. 1b) in the VCC specification language is shown in Fig. 3. Here, ghost parameters are used to simplify reasoning (e.g., aliasing of `v1` and `v2`) and for convenience (e.g., using lambda-expressions to describe contents of map `v2` in line 10, as well as passing more than one value out of the function by using ghost out parameters instead of integer pointers).

Abbreviations `valid_cand` and `valid_vote` define when an integer is in the appropriate range for the number of candidates respectively votes defined.

Proving the implementation of monotonicity (as given in Fig. 1a) correct w.r.t. its specification, the VCC methodology uses contracts for both invocations of function `voting` to deduce the monotonicity postcondition, abstracting from the concrete implementation of `voting`. This implies, however, that the user has to provide sufficient details about the functional behaviour in the form of pre- and postconditions, which already exists as imperative C implementation. As our goal is to reduce the user interaction needed in the verification process, we decided

instead to inline both calls to `voting` in order to being able to selectively replace parts of the implementation by a more declarative description in the form of VCC annotations.

Given this implementation, together with the method contract for `monotonicity` as shown in Fig. 3, the VCC tool can be used similarly to LLBMC in a fully bounded setup:

Experiment 2. For this experiment, the loops for counting votes, as well as determining the unique maximum are unrolled manually in both invocations of function `voting` – the number of loop iterations sufficient to detect errors is determined by the number of candidates and votes. VCC is run on the resulting C source containing annotations given in Fig. 3 with memory consumption limited to 1.5 GB and a timeout of 7 h. The resulting run-times for verification using VCC up to seven votes are shown in Fig. 2b (missing data points indicate a timeout).

In contrast to the bounded model checking experiment using LLBMC, our results indicate that run-times for VCC are (at least) exponential both in the number of candidates and votes.

Weaker performance of VCC in this experiment compared to the variant using LLBMC is unsurprising – the deductive verification methodology as used by VCC heavily relies on modularization and abstraction in order to give good results. Using instead VCC as intended, a full (i.e., unbounded) correctness proof could be obtained by annotating all loops with invariants which together with the rest of the program imply the monotonicity property. Specifying loop invariants that capture the effects of a loop as precisely as possible is not only helpful in proving the monotonicity property in the unbounded case, but also when combining deductive and bounded methods in Sec. 3.3.

As an example, Fig. 4 shows a version of such a set of invariants for the first loop of the code[1] of function `voting`, using the following recursive definition of `count`:

```
1 _(ghost _(pure) \integer count(int v[int], int to, int cmp)
    _(returns (to <= 0) ? 0 : count(v, to-1, cmp) + (v[to-1] == cmp ?  1 :  0)))
```

Invariants for the loops to count votes resp. to determine the (unique) maximum of the result are straightforward: besides stating the range of the loop index variable, for the first loop, the invariants relate the contents of array `res` to specification function `count` (line 3); the invariant in line 5 determines the range of the contents of `res` in relation to the loop index variable. This information is actually implied by the other loop invariants and the contract of `count` making the corresponding invariant in line 5 one of the non-essential auxiliary annotations [4].

Invariants of the second loop (not shown here) state that variable `max` is indeed the maximum of the array `res` up to index `i`. In case `elect1` is not zero, it is the index to a (not necessarily unique) maximum of `res`. A subsequent invariant then restricts the value of `elect1` further, i.e., `elect1` is not zero iff `res1` has a unique maximum. As we have defined `res` to be a map with C integers as domain, and the loop invariant of the first loop does not provide information about contents of `res` outside the range $0 \ldots$ `V`, these values are underspecified. It may thus be the case that `elect1` is outside the range $0 \ldots$ `V`, still fulfilling the invariants presented so far. To rule out this possibility, we restrict `elect1 <= C` by another invariant.

However, the VCC annotations for our example presented so far show an idealized version of the relevant loop properties which are not sufficient in practice. The main issue with the auto-active proof for our example is the handling of quantifiers by the SMT solver Z3 used as back-end of VCC. Universally quantified variables are treated by instantiation with appropriate terms – which terms are worth considering is determined by so called *patterns*. While Z3, together with VCC, is able in many cases to come up with suitable patterns, for some annotations, patterns

[1] The presentation of this example has been simplified: in the original version, all variable declarations and statements are of "ghost-type". Here, we only show the ghost modifier when using special specification features.

```
1 for (int i = 0; i < V; i++)
      _(invariant valid_voteCount(i))
      _(invariant \forall int k;
4         (valid_candZero(k) ==> res1[k] <= i && res1[k] >= 0))
5     _(invariant \forall int k;
6         (valid_candZero(k) ==> (res1[k] == count(v1, i, k))))
  { //count the number of votes and store result in res[] }
```

Figure 4: Sample VCC loop invariant for function `voting`.

are either too general (leading to out-of-memory errors due to a huge number of unneeded quantifier instantiations) or do not match on relevant terms in proof search (with the result that VCC is not able to deduce some of the annotations).

Finding the right set of patterns is a non-trivial process. In our experiments, correct trigger selection was one of the more time consuming tasks. Also, as trigger matching depends on the terms introduced in the proof process, trigger annotations proved to be sensitive to even minor changes in other annotations. On a more general note, changing the order of seemingly unrelated annotations also sometimes adversely affects the proof search.

As a result, due to the interactions between the different annotations for the four loops, providing a sufficient set of annotations for our example – that allows for a *general* correctness argument – turned out to be too time consuming in order to be used for iterative development of voting schemes. Instead, experiments for VCC were performed with a set of annotations approximating a full functional correctness proof:

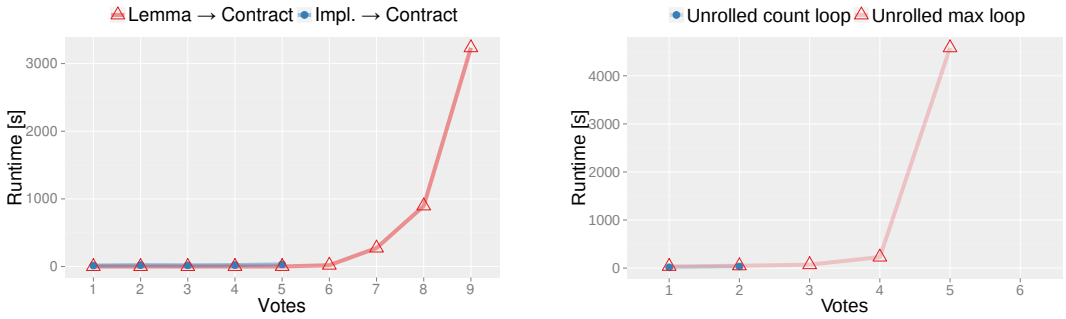
Experiment 3. Annotations given in Fig. 4, together with a moderate amount of further auxiliary annotations allows Z3 to prove the monotonicity property for a *bounded* number of *votes*, but in contrast to our experiments with LLBMC for an *unbounded* number of *candidates* with run-times as shown in Fig. 5a (labeled "Impl. $\rightarrow$ Contract"; missing data points indicate that the memory consumption exceeded the limit of 1.5 GB).

3.3 Bounded Verification

Assuming annotation completeness of VCC [4], a set of annotations exists to prove the monotonicity property in the unbounded case. Finding these annotations in practice is often infeasible. Thus, we now consider the goal of finding a small set of annotations that is easy to identify for the user and allows for verification of problem instances with a bound exceeding the outcomes of the SBMC experiments.

The experiments show that neither SBMC nor deductive verification alone perform as desired. Abstracting the implementation of all loops via invariants allows us to show monotonicity for up to five votes without restricting the number of candidates, but incurs considerable annotation overhead. On the other hand, SMBC does not need user interaction in the form of annotations but only scales up to a moderate amount of candidates and votes.

Finding the right annotations for deductive verification is not only complicated due to technical difficulties as described above; we claim that in some cases, finding a more abstract description of the implementation in form of, e.g., a loop invariant is virtually impossible. For these parts of the implementation, we propose to use SBMC techniques – a natural candidate is, e.g., unrolling loops instead of providing an invariant. The decision between both options



(a) Fully annotated

(b) With unrolling

Figure 5: Verification performance checking monotonicity for Plurality Voting using VCC.

depends not only on the level of difficulty to come up with the specification but also on the impact of the depth of unrolling to scalability of bounded model checking.

Our experiments with LLBMC show that scalability of the approach when used for majority voting differs between the number of candidates and the number of votes, the latter having a more pronounced effect on the run-times of LLBMC.

For the majority voting scheme, we examined performance of a combination of the bounded model checking approach and annotation-based deductive verification by unrolling selected loops instead of providing invariants for verification with VCC:

Experiment 4. For each of the two loops of function `voting`, the chosen loop is unrolled depending either on the number of candidates or votes, while the other loop is annotated with the invariants from experiment 3 (as shown exemplary in Fig. 4).

By retaining invariants for one of the loops, we abstract either from the concrete number of candidates or, analogously, from the number of votes. The resulting run-times of VCC for exactly five candidates and up to six votes are shown in Fig. 5b (missing data points indicate out of memory errors of VCC).

The results for selective unrolling of loops partially reflect the outcome of the experiments in the fully annotated version of experiment 3). For the latter, VCC was able to prove the monotonicity property independently of the number of candidates. The same holds true when unrolling the loop counting the votes which is independent from the number of candidates. Run-times for this proof are comparable to those given in Fig. 5b for only five candidates. The ability to prove the monotonicity property only for a small number of votes in the fully annotated case re-appears: unrolling the counting-loop in the bounded verification experiment only aggravates this effect.

In the case of unrolling the loop finding the maximum of the vote counts, we give up the loop invariant that allowed us to abstract from the number of votes, making the run-time of VCC depend from both the number of candidates and votes.

Improving Performance of the Verification Approach. Orthogonal to the combination of bounded techniques and deductive verification, there are further options to improve on the performance of verification in our context, by modularizing reasoning about the program.

One obvious approach is to introduce lemmas, as demonstrated by the following experiment:

Experiment 5. Using the annotations for `monotonicity` from experiment 3, we were easily able to compile a set of annotations taken from the loop invariants and precondition which (a) hold at the end of the program and which (b) imply the monotonicity postcondition. In contrast to the original verification task, successful up to five votes, property (a) could be proven for the unbounded case. This leaves proof obligation (b) as the bottleneck, as the run-times labeled "Lemma → Contract" in Fig. 5a show.

Another form of modularization is to prove contracts for implementation parts in a first verification pass but only making use of a *subset* of the contract in the form of assumptions in the second pass. This takes account of the fact that some annotations are only used "locally", e.g., to prove that a loop invariant is preserved by execution of the loop body, but never needed to prove the postcondition of the program. In fact, some of these unnecessary annotations are a burden in the proof search when added to the set of assumptions, e.g., by triggering quantifier instantiations. Resolving the dependencies between the different specification blocks in this manner thus possibly allows to simplify each block.

4 Heavy-Weight Verification

The heavy-weight verification method involves producing a logical formalisation of both the software's requirements and the software itself, then constructing a formal proof that the software matches the specification using an LCF-style theorem-proving assistant. Unlike the light-weight methodologies above, this method pushes all of the proof obligations onto the user since the user must "tell" the verification tool how to construct the proof. The benefit is that the produced proof is highly trustworthy and can be more general. LCF-style theorem-provers rely only on the correctness of a very small kernel of code which has been scrutinised for correctness by a huge number of people. All theorems or rules that exist in the system have been proved by deduction from this kernel. Thus there effectively cannot be a flaw in the system's logic once we accept that the kernel is correct. We used the HOL4 theorem prover (`http://hol.sourceforge.net/`).

The proof procedure involves producing the following, step-by-step:

Implementation: An implementation in SML of the plurality vote-counting scheme.

Translation: A translation of the implementation into HOL4's formal logic.

Specification: An encoding of MC in HOL4's formal logic.

Proof: A proof acceptable to the HOL4 theorem prover that the specification (of MC) holds of the translation (of the implementation).

4.1 Implementation

The implementations used in the other methodologies mentioned are procedural, and written in C. By contrast, the heavy-weight methodology we are concerned with here cannot be applied directly to a program written in an imperative programming language. As such, a plurality counting algorithm has been written in StandardML (SML), a strict functional programming language. The SML code for the plurality counting algorithm is given in Figure 6.

This implementation makes use of the `option` type operator. Specifically, `ELECT` returns a value of type `num option`. `WINNER` also makes use of the `num option` datatype. The `option` type operator is acting in both cases as a wrapper around type `num` to allow the program to

```
local
    (* Counts the number of votes in the given list for candidate c *)
    fun COUNTVOTES c []     = 0
      | COUNTVOTES c (h::t) = if h = c then 1 + COUNTVOTES c t
                                       else 0 + COUNTVOTES c t;

    (* Finds winner from all candidates numbered c or lower. *)
    fun WINNER 0 v = (SOME 0, COUNTVOTES 0 v)
      | WINNER c v = let val numvotes = COUNTVOTES c v in
                         let val (w, max) = WINNER (c-1) v in
                             if numvotes > max then (SOME c, numvotes)
                             else if numvotes = max then (NONE, max)
                             else (w, max)
                         end
                     end;
in
    (* C is the number of candidates, v is the list of votes *)
    fun ELECT C v = if C <= 0 then NONE else #1 (WINNER (C-1) v)
end;
```

Figure 6: Implementation of a plurality counting algorithm in SML.

return either a number (as $\texttt{SOME}\, c$) or the lack thereof (NONE). Do not confuse the statement $\texttt{SOME}\, c$ as being shorthand for "there exists some c".

For simplicity, each candidate is represented by a number from 0 to $(C-1)$, and the set of votes by a list of numbers: each representing a vote for the numbered candidate. Let c_i be the i^{th} candidate and v_j be the j^{th} vote. A vote v_j is a vote for c_i iff the j^{th} member of the list v is equal to i. If $v_j < 0$ or $v_j \geq n$ where n is the number of candidates, then v_j is invalid.

Our implementation runs in $O(cv)$ time with number of candidates c and number of votes v. An $O(c+v)$ implementation is possible, but it was kept this way to maintain the program's functional purity and simplicity (thereby making it easier to reason about). Theoretically, the same results are provable of a $O(c+v)$ implementation but we have not explored this possibility.

4.2 Translation into HOL4

Figure 7 shows the implementation translated into recursive definitions in HOL4. Syntactically speaking, the translation between SML and HOL4 is purely mechanical. is examined in section 4.5.

4.3 Specification

Formally stated in higher-order logic, the definition of monotonicity given on page 3 becomes:

$$\forall C\, w\, v\, v'.\, \Big((\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\, n < \texttt{LENGTH}\, v \Rightarrow (\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v')\big) \\ \wedge\, (\texttt{ELECT}\, C\, v = \texttt{SOME}\, w)\Big) \Rightarrow (\texttt{ELECT}\, C\, v' = \texttt{SOME}\, w) \quad (1)$$

where: v is a list representing the set of prior votes; v' is a list representing the set of posterior votes; w is a number representing the winning candidate; C represents the number of candidates;

```
val COUNTVOTES_def = Define ‘
    (COUNTVOTES c []     = 0) /\
    (COUNTVOTES c (h::t) = if (h = c) then 1 + COUNTVOTES c t
                           else 0 + COUNTVOTES c t)‘;

val WINNER_def = Define ‘
        (WINNER 0 v = (SOME 0, COUNTVOTES 0 v)) /\
        (WINNER c v = let (numvotes = COUNTVOTES c v) in
                      let ((w, max) = WINNER (c-1) v) in
                          if numvotes > max then (SOME c, numvotes)
                          else if numvotes = max then (NONE, max)
                          else (w, max))‘;

val ELECT_def = Define ‘
    ELECT C v = if C <= 0 then NONE else FST (WINNER (C-1) v)‘;
```

Figure 7: Translation of the plurality counting algorithm into recursive definitions in HOL4.

$\texttt{LENGTH}\, l$ is the length of list l; and $\texttt{EL}\, n\, l$ is the n^{th} element of list l, where $0 \leq n < \texttt{LENGTH}\, l$. Note that LENGTH and EL are predefined recursive functions in HOL4 and $\texttt{EL}\, 0\, (h :: t) = h$.

4.4 Proof

The entire proof was completed using the HOL4 theorem prover. Rather than explaining the syntax of HOL4 and how it corresponds to higher-order logic, all of the formulae in this section are given using standard higher-order logic syntax.

Let ϕ be defined as follows:

$$\phi = \Big((\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \ \wedge \big(\forall n.\, n < \texttt{LENGTH}\, v \Rightarrow (\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v')\big)\Big) \quad (2)$$

This allows us to rewrite the proof obligation (1) as:

$$\forall C\, w\, v\, v'.\, \big(\phi \wedge (\texttt{ELECT}\, C\, v = \texttt{SOME}\, w)\big) \Rightarrow (\texttt{ELECT}\, C\, v' = \texttt{SOME}\, w) \quad (3)$$

C is either 0 or the successor to some number (ie. $\texttt{SUC}x$). Examining these cases and applying some basic substitution allows us to rewrite the proof obligation (3) in terms of WINNER:

$$\forall c\, w\, v\, v'.\, \big(\phi \wedge (\texttt{FST}\, (\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w)\big) \Rightarrow (\texttt{FST}\, (\texttt{WINNER}\, c\, v') = \texttt{SOME}\, w) \quad (4)$$

The new obligation is that at any stage of the recursion: if w beats all other candidates examined so far with the prior votes, then w beats the same candidates with the posterior votes.

To get to the core of the problem, it is desirable to go one step further and rewrite the proof obligation in terms of COUNTVOTES. In order to do this, we need a formula relating WINNER and COUNTVOTES. Hence the following lemma:

$$\forall c\, v\, w.\, w \leq c \ \Rightarrow \big((\texttt{FST}\, (\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w) \iff \forall c'.\, c' \neq w \ \wedge \ c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v\big) \quad (5)$$

This lemma states that if w beats all candidates numbered c or less, then w also has more votes than all of the said candidates and vice versa. The proof of this lemma relies upon inductive proofs of various properties of WINNER. See Appendix A for more specific details.

The proof obligation (4) can thus be rewritten in terms of `COUNTVOTES` as follows:

$$\forall c\, w\, v\, v'.\ \big(\phi \wedge (\forall c'.\ c' \neq w\ \wedge\ c' \leq c \Rightarrow \texttt{COUNTVOTES}\ w\ v > \texttt{COUNTVOTES}\ c'\ v)\big) \\ \Rightarrow (\forall c'.\ c' \neq w\ \wedge\ c' \leq c \Rightarrow \texttt{COUNTVOTES}\ w\ v' > \texttt{COUNTVOTES}\ c'\ v') \quad (6)$$

In other words we need to prove that if w has more votes than the set of lesser-numbered candidates using the prior votes, and the conditions in ϕ hold, then w also has more votes than all the aforementioned candidates using the posterior votes. A structural case analysis of v and v' can now be performed (the lists being either empty or having a head and tail).

In order to make the proof fall all the way through it is necessary to prove the monotonicity of `COUNTVOTES` (both for the winner *and* for other candidates). These two monotonicity properties are given below:

$$\forall w\, v\, v'.\ \phi \Rightarrow \texttt{COUNTVOTES}\ w\ v' \geq \texttt{COUNTVOTES}\ w\ v \quad (7)$$

$$\forall w\, v\, v'.\ \phi \Rightarrow (\forall c.\ c \neq w \Rightarrow \texttt{COUNTVOTES}\ c\ v \geq \texttt{COUNTVOTES}\ c\ v') \quad (8)$$

Appendix A lists all the lemmas involved in the proof and a diagram of their inter-dependencies.

4.5 Summary and Discussion of the Heavy-Weight Approach

There are two aspects worth considering when evaluating the feasibility of the heavy-weight verification process: the effort involved and whether the proof actually covers everything that is required. The latter needs addressing.

We have proved that our recursive definition in HOL4 matches our encoding of MC. Syntactically, our SML program appears the same as our recursive definitions. Semantic equivalence is another matter. We have no evidence to show that our SML implementation is equivalent to our HOL4 translation, except for the fact that they look syntactically similar. This means we cannot extend our proved results to our concrete SML implementation.

A particularly illuminating example of this conundrum is the difference between HOL4's and SML's handling of numerical types. In both programs, the candidates are represented by numbers. SML uses integers by default, whereas HOL4 represents them as Peano numbers: `0`, `SUC 0`, `SUC (SUC 0)` etc. The underlying representation would not matter if the same operations were defined and those operations had the same effect. Unfortunately this is not the case. $0 - 1 = 0$ is provably correct in HOL4, whilst `0 - 1` will result in `~1` in SML (`~` is unary negation in SML so `~1` means -1). We are fortunate that our SML implementation deals only with positive integers. Nevertheless, we cannot be formally certain that the implementation and translation are equivalent. To be precise, the results have been proven for some semantically equivalent SML program that may or may not actually be *the* implementation in SML.

One way to get around this is to execute the HOL4 definitions directly. After all, the encoding in HOL4 is itself executable using HOL4's deductive rewriting engine. Unfortunately there is a large loss in efficiency when using this method. The SML implementation takes less than 7 minutes, using less than 10.5 GiB of memory, to count 250 million votes with 160 candidates. By contrast, with the same number of candidates, the HOL4 translation takes 40 minutes, using 14 GiB of memory, to count 25 thousand votes. There are also several issues with converting a list of votes into a logical statement acceptable to HOL4.

We therefore need a way to extend our results about the translation to a concrete SML program. One option would be to write the HOL4 recursive definitions, then translate the definitions into SML along with some associated proofs of semantic equivalence. CakeML(`https://cakeml.org/`) is a project that endeavours to make this feasible [17]. It is currently under development so is not explored here, but may in future provide the missing link required.

The entire process from implementation to complete verification took 3 weeks. Bear in mind that this was a learning process, with only 1–2 months-worth of prior experience with HOL4. Ultimately, 3 weeks is a short time to spend producing a piece of fully formally verified software. How this scales to more complex problems remains to be seen.

Another measure of the effort involved is the proof-to-implementation ratio, measured in lines of code (LoC). The implemented algorithm spans 24 lines whilst the proof spans 590. This gives at least 24 lines of proof for each line of implementation. Unfortunately, the final LoC measurement does not take into account the effort expended in exploring unproductive proof strategies. This makes its applicability here questionable. Nevertheless, it may be helpful when comparing the procedure to other verification methods. Assuming the ratio can be extrapolated to to larger programs, verifying a 100-line program would require 2400 lines of verification. The complexity of such a proof may make this impractical.

It is also worth noting that the methodology here is not well suited to rapid prototyping. In particular, if you try to prove an invalid property, you can spend an indeterminate amount of time trying before you realise it is not possible.

The procedure for fully formally verifying properties of vote counting algorithms is feasible for small simple algorithms. It remains to be seen whether the procedure will scale to complex proportional representation systems. In any event, there is a gap in our proof system that needs filling in order for the proofs to be used in practical real-world situations. There is an ongoing effort to plug this gap and this needs further investigation.

5 Related Work

Regarding the analysis of voting schemes, voting rules or voting algorithms, there does not appear to be much research using methods of formal logics. However, there has been extensive research in the domain of social choice theory analyzing theoretical voting schemes by the means of natural language [18]. Other research in the field of mathematics and computer science covers algorithmic analysis and classification of voting schemes [11]. This analysis defines mathematical distances using rather general voting scheme properties such as neutrality and consistency, which are usually pre-assumed in our analysis. Furthermore, there has been research on verification of voting systems, i.e. considering a concrete voting software [9].

Some research on the trade-offs between bounded and modular verification has focused on how to make automatic verification more efficient and powerful. This involves adding and effectively using some external decision procedures or SMT solver e.g. for quantifiers or integer arithmetics [1] or even using a whole different logical foundation. There has also been research on the combination of bounded and modular verification techniques [15], but the scope of the analysis is still limited, or focussed on different logical theories [14].

6 Conclusion and Future Work

At first sight, the results of our experiments are twofold: while SBMC is normally used to detect bugs, our experiments show that the technique can also be successfully applied to demonstrate correctness of a simple majority voting scheme w.r.t. to its specification, in our case the monotonicity property – albeit only for a small number of candidates and votes. On the other end of the spectrum of verification tools, interactive, higher order theorem provers allow for a general and rigorous correctness proof, but incur a substantial amount of user interaction.

However, the experiments also indicate potential issues when using the different verification methods on realistic, more complex voting schemes. Even in our simple example, experiments indicate that proving monotonicity with SBMC has exponential runtime in the number of votes – this effect is likely to be even larger for complex voting schemes (e.g., due to nested loops in the implementation). Heavy-weight verification heavily depends on user interaction, as demonstrated by the proof-to-implementation ratio for our example – whether the proof size scales favorably for more complex voting schemes, or whether the user is able to find the necessary abstract properties of the implementation at all remains an open question.

To improve the scalability of the SBMC approach, we examined the combination of SBMC and deductive verification on source code level: by providing invariants for some of the loops in the implementation, we removed the necessity to unroll all loops. The idea behind this combination is to take advantage of the automation of SBMC for program parts which are difficult to specify when using deductive verification tools and the benefit of abstracting from the implementation using contracts for those parts where SBMC does not scale well.

First experiments with the combination of both techniques showed performance worse than using SBMC alone – however, the previously described scalability issues with the SBMC tool indicate implementation parts that could benefit from an annotation-based abstract specification. Further work is needed to improve the methodology to unite both verification approaches.

The experiments also showed that annotation-based verification required far more user interaction than anticipated. Indeed, the extra interaction made it impossible to use this approach in an iterative development process – we have identified and presented some of the issues in this paper. These problems can partially be addressed by better modularization of the proofs, which we believe would also benefit the combination of bounded methods and deductive verification.

Instead of superimposing the bounded model checking approach on VCC, an alternative may be to incorporate modularization concepts into LLBMC. Another way to benefit from deductive verification is to verify the implementation of the voting scheme correct w.r.t. general properties which allow, e.g., to exploit symmetries in program execution when checking with SBMC.

The heavy-weight approach took roughly 10 weeks of full time work: 7 weeks of learning HOL4 and 3 weeks to specify and verify the code. Given that the specification of "monotonicity" was more general than the one used in the light-weight approach and that the proof covers any number of candidates and any number of votes, this seems a small price to pay. However, the following caveats apply. We verified a HOL-encoding of an SML program, not the SML program itself, so we have no proof of their equivalence. A visual comparison is compelling for the simple case we examined here, but might not be for a complex STV voting scheme such as Hare-Clark. The HOL4 encoding of plurality voting is itself executable, but is only feasible for small-scale elections. The CakeML project, currently under active development, may provide a solution that could be used to bridge this gap. Also, the interactive proof methodology does not lend itself to rapid prototyping since it does not provide counter-examples. Indeed, one can spend an inordinate amount of time trying to prove false conjectures before realising that they are indeed false.

Can we combine annotation-based light-weight verification tools and heavy-weight interactive theorem provers? The different programming paradigms of the implementation prevent a direct interchange of the different code versions, but specifications inferred by one method could be exported as assumptions to modularize or simplify the proofs in another tool.

References

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[2] K. J. Arrow. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58(4):pp. 328–346, 1950.

[3] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. Boer, M. Bonsangue, S. Graf, and W.-P. Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin Heidelberg, 2006.

[4] B. Beckert, T. Bormer, and V. Klebanov. Improving the usability of specification languages and methods for annotation-based verification. In B. Aichernig, F. S. de Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects, 9th International Symp. FMCO 2010. State-of-the-Art Survey*, volume 6957 of *LNCS*. Springer, 2011.

[5] B. Beckert, T. Bormer, F. Merz, and C. Sinz. Integration of bounded model checking and deductive verification. In *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*, volume 7421 of *LNCS*, pages 86–104. Springer, 2012.

[6] B. Beckert, R. Goré, and C. Schürmann. Analysing vote counting algorithms via logic - and its application to the CADE election scheme. In M. P. Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 135–144. Springer, 2013.

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009.

[8] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340, 2008.

[9] G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. In N. Shankar and J. Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[10] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[11] E. Elkind, P. Faliszewski, and A. M. Slinko. On the role of distances in defining voting rules. In W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, and S. Sen, editors, *AAMAS*, pages 375–382. IFAAMAS, 2010.

[12] S. Falke, F. Merz, and C. Sinz. The bounded model checker LLBMC. In *ASE*, pages 706–709. IEEE, 2013.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.

[14] O. Grumberg and D. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[15] Y. Hashimoto and S. Nakajima. Modular checking of C programs using SAT-based bounded model checker. In S. Sulaiman and N. M. M. Noor, editors, *APSEC*, pages 515–522. IEEE Computer Society, 2009.

[16] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[17] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ML. In *POPL*, pages 179–192, 2014.

[18] E. Pacuit. Voting methods. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, winter 2012 edition, 2012.

A Lemmas

The following is a full listing of each lemma proved during the HOL4 proof. Figure 8 shows the dependencies between the various lemmas. See Section 4.4 for an explanation of the proof.

CV_LE_W:

$$\forall c\, v\, c'.\ c' \leq c \Rightarrow \texttt{COUNTVOTES}\ c'\ v \leq \texttt{SND}\ (\texttt{WINNER}\ c\ v) \tag{9}$$

CV_GT_W:

$$\forall v\, c\, c'.\quad c' < \texttt{SUC}\ c\ \wedge\ \texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v > \texttt{SND}\ (\texttt{WINNER}\ c\ v) \Rightarrow \texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v > \texttt{COUNTVOTES}\ c'\ v \tag{10}$$

W_BOUNDED:

$$\forall c\, v\, c'.\ c' > c \Rightarrow (\texttt{FST}\ (\texttt{COUNTVOTES}\ c\ v) \neq \texttt{SOME}\ c) \tag{11}$$

W_CV:

$$\forall c\, v\, w\, m.\ (\texttt{WINNER}\ c\ v = (\texttt{SOME}\ w, m)) \Rightarrow (\texttt{COUNTVOTES}\ w\ v = m) \tag{12}$$

W_CV_2:

$$\forall c\, v\, w.\ (\texttt{SOME}\ w = \texttt{FST}\ (\texttt{WINNER}\ c\ v)) \Rightarrow (\texttt{COUNTVOTES}\ w\ v = \texttt{SND}\ (\texttt{WINNER}\ c\ v)) \tag{13}$$

NEXT_C:

$$\forall v\, w\, c.\ (\texttt{SOME}\ w = \texttt{FST}\ (\texttt{WINNER}\ c\ v)) \wedge \texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v < \texttt{SND}\ (\texttt{WINNER}\ c\ v) \Rightarrow \texttt{COUNTVOTES}\ w\ v > \texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v \tag{14}$$

W_IMP_GT_C:

$$\forall c\, v\, c'\, w.\ \big((c' \neq w) \wedge (c' \leq c) \wedge (\texttt{FST}\ (\texttt{WINNER}\ c\ v) = \texttt{SOME}\ w)\big) \Rightarrow \texttt{COUNTVOTES}\ w\ v > \texttt{COUNTVOTES}\ c'\ v \tag{15}$$

C_HAS_MAX:

$$\forall v\, c.\ \exists c'.\ c' \leq c \wedge (\texttt{COUNTVOTES}\ c'\ v = \texttt{SND}\ (\texttt{WINNER}\ c\ v)) \tag{16}$$

DRAW:

$$\forall v\, c.\ (\texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v = \texttt{SND}\ (\texttt{WINNER}\ c\ v)) \Rightarrow \exists c'.\ c' \leq c \wedge (\texttt{COUNTVOTES}\ (\texttt{SUC}\ c)\ v = \texttt{COUNTVOTES}\ c'\ v) \tag{17}$$

GT_C_IMP_W:

$$\forall c\, v\, w.\ w \leq c \Rightarrow \big((\forall c'.\ c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\ w\ v > \texttt{COUNTVOTES}\ c'\ v) \Rightarrow (\texttt{FST}\ (\texttt{WINNER}\ c\ v) = \texttt{SOME}\ w)\big) \tag{18}$$

W_EQ_GT_C:

$$\forall c\, v\, w.\ w \leq c \Rightarrow \big((\texttt{FST}\ (\texttt{WINNER}\ c\ v) = \texttt{SOME}\ w) = (\forall c'.\ c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\ w\ v > \texttt{COUNTVOTES}\ c'\ v)\big) \tag{19}$$

W_LT_C:

$$\forall c\, v\, w.\, (\texttt{FST}\, (\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w) \Rightarrow w \leq c \tag{20}$$

MONO_CV_W:

$$\forall w\, v\, v'.\, (\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\, (n < \texttt{LENGTH}\, v) \Rightarrow ((\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v'))\big) \\ \Rightarrow \texttt{COUNTVOTES}\, w\, v' \leq \texttt{COUNTVOTES}\, w\, v \tag{21}$$

MONO_CV_C:

$$\forall w\, v\, v'.\, (\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\, (n < \texttt{LENGTH}\, v) \Rightarrow ((\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v'))\big) \\ \Rightarrow \forall c.\, c \neq w \Rightarrow \texttt{COUNTVOTES}\, c\, v \geq \texttt{COUNTVOTES}\, c\, v' \texttt{COUNTVOTES}\, w\, v \tag{22}$$

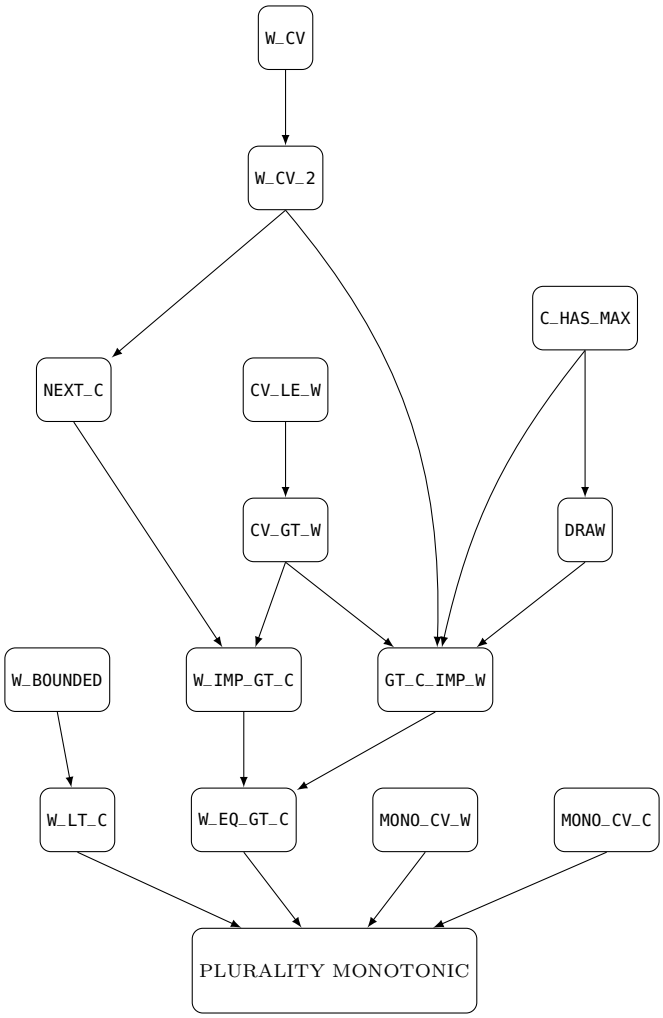


Figure 8: Dependencies between lemmas. The proof of a lemma at the destination of an arrow relies upon the lemma at the arrow's origin.