Verifying Voting Schemes

Bernhard Beckert[a,1,*], Rajeev Goré[b], Carsten Schürmann[c,2,*], Thorsten Bormer[a], Jian Wang[c]

[a] *Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany*
[b] *Research School of Computer Science, The Australian National University, Australia*
[c] *IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark*

Abstract

The possibility to use computers for counting ballots allows us to design new voting schemes that are arguably fairer than existing schemes designed for hand-counting. We argue that formal methods can and should be used to ensure that such schemes behave as intended and conform to the desired democratic properties. Specifically, we define two semantic criteria for single transferable vote (STV) schemes, formulated in first-order logic over the theories of arrays and integers, and show how bounded model-checking and SMT solvers can be used to check whether these criteria are met. As a case study, we then analyse an existing voting scheme for electing the board of trustees for a major international conference and discuss its deficiencies.

Keywords: Voting schemes, verification, bounded-model checking, single transferable vote, SMT solvers

1. Introduction

The goal of any social choice function is to compute an "optimal" choice from a given set of preferences. Voting schemes in elections are a prime ex-

*Corresponding author

Email addresses: beckert@kit.edu (Bernhard Beckert), Rajeev.Gore@anu.edu.au (Rajeev Goré), carsten@itu.dk (Carsten Schürmann), bormer@kit.edu (Thorsten Bormer), jwan@itu.dk (Jian Wang)

[1] Phone: +49-72160844025

[2] This research has been funded in part by the DemTech research grant CCR-0325808 awarded by the Danish Council for Strategic Research, programme commission Strategic Growth Technologies.

ample of such choice functions as they compute a seat distribution from a set of preferences recorded on ballots. By *voting scheme* we refer to the method for counting ballots and computing who won – as opposed to an actual computer implementation of such a scheme or a scheme describing the process of casting votes via computer. The difficulty in designing preferential voting schemes is that the optimisation criteria are not only multi-dimensional, but multi-dimensional on more than one level. On one level, we want to satisfy each voter, so each voter is a dimension. On a higher level, there are desirable global criteria such as "majority rule" and "minority protection" that are at least partly inconsistent with each other. It is well-known that "optimising" such theoretical voting schemes along one dimension may cause them to become "sub-optimal" along another.

This observation is not new and voting specialists have proposed a series of mathematical criteria [1] that can be used to compare various voting schemes with one another. A classic example is the notion of a Condorcet winner, defined as the candidate who wins against *each* other candidate in a one-on-one contest. Such a winner exists provided that there is no cycle in the one-to-one contest relation. A voting scheme is said to satisfy the Condorcet criterion if the Condorcet winner is guaranteed to be elected when such a winner exists. Another is the *monotonicity criterion* which requires that a candidate who wins a contest will also win if the ballots were changed uniformly to rank that candidate higher.

In practice, theoretical voting schemes are often simplified in many ways when used in real-world elections, typically to reduce their complexity to allow counting by hand. Such practical schemes may not satisfy general properties such as the Condorcet criterion simply because it is intractable to compute the Condorcet winner by hand, but they may satisfy some weaker version of "optimality" that is specific to that particular scheme. It may even happen that one among the optimal winners is chosen at random [2] (as allowed by the Australian Capital Territory's Hare-Clark Method) or that someone other than the optimal winner is elected.

Voting schemes also evolve over time – for national elections in the large, and local elections, union elections, share holder elections, and board of trustee elections in the small. Incremental changes to the electoral system, the tallying process and the related algorithms challenge the common understanding about what the voting scheme actually does. For example, since 1969 some local elections in New Zealand adopted Meeks' method [3], which is a voting scheme for preferential voting that uses fractional weightings in

its computations and is too complex to count by hand. This also required an adjustment of understanding about who will now be elected. In general, it is often not clear whether changes to the electoral system improve or worsen the overall quality of a voting scheme with regard to the various dimensions of optimisation. Changes to the electoral system in Germany, for example, have created paradoxical situations where more votes for a party translate into fewer seats and fewer votes into more seats, and have prompted Germany's Supreme Court to intervene at several occasions (see, e.g., [4]).

Many jurisdictions around the world are now using computers to count ballots according to traditional voting schemes. Using computers to count ballots opens up the possibility to use voting schemes which really are optimised along multiple dimensions, while retaining global *desiderata* such as the Condorcet criterion. The inherent complexity of counting ballots according to such schemes means that it may no longer be possible to "verify" the result by hand-counting, even when the number of ballots is small. It is therefore important to imbue these schemes with the trust accorded to existing schemes. Note that our focus is on trust in the voting scheme, not trust in the computer-based process for casting votes.

One way to engender trust in such complex yet "fairer" voting schemes is to specify the *desiderata* when the scheme is being designed, and then formally check that the scheme meets these criteria before proposing changes to the legislation to enact the scheme. Such formal analyses could contribute significant unbiased information into the political discussions that typically involve such legislative changes and also assure voters that the changes will not create paradoxical situations as described above.

Formal analysis, however, is only practicable when we possess formal specifications of the voting scheme. We argue that it is important to give declarative specifications of the properties of a voting scheme for two reasons: (1) For understanding their properties and how they change during the evolution process, so that improving a scheme in one aspect does not by accident introduce flaws w.r.t. other aspects. (2) For checking the correctness of the scheme from both an algorithmic and implementation perspective. We also argue that general criteria are not sufficient and criteria are needed that are tailor-made for specific (classes of) voting schemes.

The properties in question are difficult to state, to formalise, to understand, to analyse, and to describe declaratively (as opposed to algorithmically) because: the final voting scheme may have to compromise between the conflicting demands of multiple individual desirable properties; the voting

scheme may evolve and we may have to revisit these desiderata; even when the properties can be made mathematically precise, the resulting mathematical statement cannot serve as a specification if the electoral law defines a voting scheme that does not (always) compute the optimal solution.

Contributions Here, we show that seemingly innocuous revisions to a voting scheme can have serious implications on the desired properties of the system and how analysis techniques employing Satisfiability Modulo Theories (SMT) solvers [5] can be used to discover them. As a running example, we use the preferential voting schemes single transferable vote (STV) that is used in large national elections world-wide, but also for smaller professional elections.

In Section 3, we define two tailor-made criteria to establish the desired properties of the voting scheme. Both criteria are formulated using first-order logic and are amenable for bounded model checking with Z3, which is the tool of choice for our formal analysis (Section 4). Besides the experiments, we also discuss advantages and disadvantages of different verification techniques. Subsequently, we discuss (Section 5) a particularly interesting variant of the Single Transferrable Vote Algorithm (CADE-STV) for the board of trustees of the International Conference on Automated Deduction (CADE). We explain its oddities and differences to standard STV, and give a historical account of the conception and the stepwise refinement of the algorithm. This paper extends our previous work on the specification and verification of voting schemes [6] and also our system description of a bounded model checking system for analysing voting schemes and its application to CADE-STV [7].

Related work Voting schemes have been investigated by social choice theorists for many decades. These tend to be mathematical analyses which prove various (relative) properties of different voting schemes: see [8, 9]. Such work tends to concentrate on what we have referred to as theoretical schemes and is often couched in terms of a formal theorem and its proof in natural language.

There is also a significant body of research on various properties of vote-casting schemes, particular security properties [10].

There does not appear to be much existing work on the formal analysis of voting schemes using methods and tools from the computer aided verification and automated deduction communities in our sense, although there is work on the formal analysis of actual implementations of such schemes [11, 12, 13].

2. Basic Definitions

Below, we define the basic notions related to voting schemes.

Definition 2.1. (Voting scheme, ballot, ballot box, election result) Given a non-empty set $\mathcal{C}$ of candidates, a voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$ is characterised by:

- a set $\mathcal{B}$ of possible ballots that can be cast by voters;
- a tallying function $\mathcal{T}$ assigning to each possible ballot box an election result, where a ballot box is a (finite) multiset of ballots and an election result is a (finite) duplicate-free sequence of candidates (the elected candidates).

Given a non-empty set $\mathcal{V}$ of voters we assume that each voter casts exactly one ballot in the election, allowing us to use the term voter and ballot interchangeably when identifying specific voters and ballots.

According to our definition of a voting schemes (Def. 2.1), the order of ballots in a ballot box is irrelevant and tallying functions are deterministic. Also, by definition, the order in which candidates are elected is part of the election result. These assumptions hold for all voting schemes considered in this paper, but the results and methods presented in the following apply as well to more general notions of voting schemes.

In this paper, we focus on preferential voting schemes.

Definition 2.2. (Preferential voting scheme) In a preferential voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$, the possible ballots $b \in \mathcal{B}$ are partial linear orders on candidates.

A ballot for a preferential voting scheme orders candidates according to the voter's preference.

Suppose that $\mathtt{C}$ candidates, numbered $1, 2, \ldots, \mathtt{C}$, are competing. Then, we use the notation $[c_1, c_2, \ldots, c_k]$ for a ballot that ranks a subset of the candidates in decreasing order of preference, where $c_i \in \{1, 2, \ldots, C\}$ and $c_i \neq c_j$ for $i \neq j$.

3. Semantic Criteria for Analysing Voting Schemes

We distinguish between general criteria that preferably each voting scheme should satisfy and tailor-made criteria that distinguish between different classes of schemes and capture the essence of particular classes. Both kinds of criteria are important for the specification and analysis of voting schemes.

Below, we first describe a few important examples of general criteria found in literature. We then give two examples for tailor-made criteria applicable to the STV family of voting schemes.

3.1. General Criteria

Many general criteria that voting schemes preferably should satisfy have been proposed (for an overview see [1]). Note that, even though these basic criteria seem obvious and indispensable for voting schemes on first sight, they are in fact not always satisfied by each reasonable voting scheme. Most real-word voting schemes violate at least some basic criteria for some possible ballot box input.

An "obvious" and widely used criterion is the *majority criterion*, which states that, if a candidate c is ranked first by a majority of voters, then c must be elected. This is indeed satisfied by all reasonable preferential voting schemes that use votes ranking candidates. However, the majority criterion can be violated by preferential voting schemes where voters can attach a numerical preference to candidates instead of just ranking them (Borda count scheme).

Another "obvious" criterion is the *monotonicity criterion* [14]. Assume that there are two ballot boxes b and b' where b' results from b by raising the preference for a candidate c in one or more of the votes and leaving the votes otherwise unchanged (i.e., a vote of the form $[c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots, c_k]$ is replaced by $[c_1, \ldots, c_{j-1}, c, c_j, \ldots, c_{i-1}, c_{i+1}, \ldots, c_k]$ $(j < i)$. The monotonicity criterion states that, if c is elected using the ballot box b, then c must also be elected using b'. Surprisingly, some real-world voting schemes – including STV – do not satisfy monotonicity [14].

A third simple criterion is the *fill-all-seats criterion*, which states that all available seats are filled provided that there are sufficiently many candidates, i.e., $\mathtt{C} \geq \mathtt{S}$. In practice, this criterion is often used in a restricted form, e.g., candidates can be elected only if they reach a certain minimal quota.

The majority criterion fully describes the election result for the simple case of a single seat and a candidate with a majority of first preferences. But we desire criteria characterising the "right" result in increasingly complex situations.

An example is the *Condorcet criterion*. A candidate c is a Condorcet winner if c wins a one-to-one comparison against all other candidates, i.e., for all $c' \neq c$ there are more voters preferring c over c' than there are voters preferring c' over c. The Condorcet criterion states that a Condorcet winner c must be elected if there is one. And, as long as there are open seats and there are Condorcet winners among the remaining candidates, these must also be elected.

The majority and the Condorcet criteria present each an example of a *conditional criterion*. They apply to a ballot box only in the case the given condition (e.g., the existence of a Condorcet winner in the Condorcet Criterion) is satisfied; in this case attesting the property (e.g., a Condorcet winner must be elected). Otherwise they hold trivially. This means, that there are two degenerate cases of a conditional criterion: the first is when the condition is never satisfied by a ballot box, in which case the criterion will always hold, no matter which ballot boxes we apply it to. For the second case, when the property of the criterion simply yields true, the condition of the criterion becomes irrelevant and a similar observation applies. We therefore propose to analyze criteria according to *coverage* and *restrictiveness*.

Coverage Should apply to as many different ballot boxes as possible.

Restrictiveness The number of possible election results for ballot boxes to which the criterion applies should be as restricted as possible.

Returning to our criteria, we note that the majority and Condorcet criteria are very restrictive (they specify exactly one winner), but they do not have good coverage (they only apply if there is a clear winner). The fill-all-seats criterion, on the other hand, has full coverage (it restricts the possible outcome for all ballot boxes), but it is not very restrictive. But, even criteria with poor coverage, such as the Condorcet criterion, provide important insights into voting schemes: It is well known, for example, that STV (which we use as a case study) does not satisfy the Condorcet criterion.

Ideally, one would like to characterise schemes by a criterion that allows exactly one result for every possible ballot box, i.e., has full coverage and is fully restrictive. But for many voting schemes used in practice, such criteria do not exist. In these cases, we rely on tailor-made criteria that strike a compromise between coverage and restrictiveness.

In summary, voting schemes can be analyzed according to many different criteria, some criteria are considered important others less important. We remark, however, that no voting scheme exists that would satisfy all reasonable general criteria simultaneously. In the case of preferential voting, Arrow's impossibility theorem [9] states that no scheme can be designed to satisfy the three fairness criteria:

Unanimity. If all voters prefer candidate A over candidate B, then A is ranked over B in the election result.

Independence of irrelevant options. If some voters change their ballot but keep the relative position of candidates A and B in their ballot, then the relative position of A and B remains unchanged in the election result.

Non-dictatorship. There is no single voter whose preferences always prevail in the election result.

3.2. Tailor-made Criteria for Preferential Voting Schemes

As stated previously, many more voting scheme criteria have been developed and are described in the literature. So, as a first approach to specifying and analysing a particular voting scheme, one could select some of these to characterise the scheme's properties. For a detailed analysis, however, that is not sufficient. General criteria cannot distinguish between variants of the same voting scheme (or the number of available general criteria would have to be very high).

For example, for our analysis of preferential voting, we have devised two tailor-made criteria that capture the essence of *preferential* voting (Criterion 2) with *proportional representation* (Criterion 1) and are applicable to our case study STV:

(1) There must be enough votes for each elected candidate.

(2) If the preferences of *all* voters w.r.t. two particular candidates are consistent, then that collective preference is not contradicted by the election result.

The first criterion only considers number of votes and ignores preferences, while the second criterion only considers preferences and ignores number of votes. This separation of the two dimensions (number of votes and preferences) is the key to finding strong criteria that can be described declaratively.

The two criteria compromise in different ways on the two goals of generality and restrictiveness: Criterion 1 has full coverage. It applies to all ballot-boxes without being too restrictive (as the order of preferences is not considered). Criterion 2 has lower coverage. It only applies if the voters' preferences are not contradictory. In that case, however, it is rather restrictive as only a small number of election results are permissible.

Criterion 2 is a weaker version of the Condorcet criterion that, in contrast to Condorcet, is satisfied by STV. It assumes a preference to be collective

if *all* voters agree (or at least not disagree), while the Condorcet criterion assumes a preference to be collective if it is supported by a *majority* of voters.

These two criteria may not cover all the desired properties that we expect to hold, however, they offer a good starting point for a formal analysis of voting schemes, especially those based on STV. For additional properties currently not covered, the two criteria may need to be refined accordingly.

3.3. Criterion 1: Enough Votes for each Elected Candidate

One core element of any STV system is the transfer of surpluses from elected to remaining candidates. Here, a surplus is defined by the number of votes of the elected candidate above the required quota. We note, that in some variants of STV, surplus votes are transfered as a whole, whereas other variants of STV transfer votes at a fractional value. To argue that a particular transfer is sensible and justifiable, some regulations require that each ballot can only be used once to elect a candidate marked on the ballot, which leads us to the definition of the first criterion. It says, that the entire ballot box can be partitioned into (disjoint) groups of (used) ballots such that each elected candidate is supported by exactly one group.

Definition 3.1. (Criterion 1: Enough votes for each elected candidate) Let $\langle \mathcal{B}, \mathcal{T} \rangle$ be a preferential voting scheme (Def. 2.2). Let $\mathtt{Q}$ be the quota, $\mathtt{C}$ the number of candidates, $\mathtt{V}$ the number of voters, and $\mathtt{S}$ the number of seats.

We define that $\langle \mathcal{B}, \mathcal{T} \rangle$ satisfies Criterion 1 iff, for all ballot boxes

$$\mathcal{b} = \{\!|\beta_1, \ldots, \beta_{\mathtt{V}}|\!\} \text{ with } \beta_i = [c_1^i, \ldots, c_{k_i}^i] \in \mathcal{B}$$

and corresponding election results $\mathcal{T}(\mathcal{b}) = [r_1, \ldots, r_\sigma]$ ($\sigma \leq \mathtt{S}$ is the number of elected candidates), there is a partition

$$\mathcal{b} = \mathcal{b}_1 \dot\cup \ldots \dot\cup \mathcal{b}_\sigma \dot\cup \mathcal{b}_{rest}$$

such that, for $1 \leq i \leq \sigma$, the following holds:

1. $|\mathcal{b}_i| = \mathtt{Q}$ (there are exactly $\mathtt{Q}$ votes in each class that supports an elected candidate).
2. $r_i \in \beta$ for all $\beta \in \mathcal{b}_i$ (each vote β in the class $\mathcal{b}_i$ supports candidate r_i, i.e., the candidate occurs somewhere among the preferences of β).

Note, that here the actual order of preferences is not taken into consideration.

Example 3.1. Assume there are four candidates A, B, C, D for two vacant seats, the votes to be counted are $[A, B, D], [A, B, D], [A, B, D], [D, C], [C, D]$, and the quota is $\mathtt{Q} = 2$. The election result $[A, D]$ satisfies Criterion 1 using the partition $\{[A, B, D], [A, B, D]\}$, $\{[C, D], [D, C]\}$, $\{[A, B, D]\}$, where the first group supports candidate A and the second supports candidate D.

Example 3.2. Criterion 1 does not capture election results exactly, it over-approximates them: Since we ignore the order of preference markings, the criterion may hold for unintended election results. The result $[B, D]$ is not a valid election result because it violates the majority criterion: A, despite its majority of first preferences, is not elected. Nevertheless, it satisfies Criterion 1, by virtue of the same partition from the previous example, because the first group also supports B as elected candidate.

Example 3.3. But, Criterion 1 also rules some election results as invalid. The result $[A, B]$, for example, which contradicts proportional representation, is not supported by this or any other partition (which shows that this criterion is indeed related to the requirement of proportional representation).

Formalisation. To formalise the criteria, we use first-order logic over the theories of natural numbers and arrays with the following notation in addition to the notation defined previously:

- b: is a two-dimensional array representing the ballot box ($\mathcal{b}$ in Def. 3.1), where $b[i, j] \in \{1, \ldots, \mathtt{C}\}$ represents the number c_i^j of the candidate that is ranked by vote i in the jth place. Thus, i's preference is $[b[i, 1], b[i, 2], \ldots]$. If vote i ranks only $k \leq \mathtt{C}$ candidates, then $b[i, j] = 0$ for $k < j \leq \mathtt{C}$.

- r: is an array representing the result $\mathcal{T}(\mathcal{b})$, where $r[i]$ is the ith candidate that is elected ($1 \leq i \leq \mathtt{S}$). If less than $\mathtt{S}$ candidates are elected, then $r[i] = 0$ for the empty seats.

Our criterion is formalised by a formula ϕ in which all the above (free) variables occur. We also use an existentially quantified variable a of type array that represents the partition and the assignment of classes in the partition to elected candidates as follows:

- $a[i] = k$ if the ith vote supports the kth elected candidate $r[k]$. If the ith vote does not support any elected candidate, then $a[i] = 0$. Using this

representation, a class $\mathcal{b}_k$ (as introduced in Def. 3.1) can be written as: $\mathcal{b}_k = \{\beta_i \mid a[i] = k, 1 \leq i \leq \mathtt{V}\}$; votes not supporting any candidate are collected in $b_{rest} = \{\beta_i \mid a[i] = 0\}$.

Then, the formula $\phi = \exists a(\phi_1 \wedge \ldots \wedge \phi_4)$ is the existentially quantified conjunction:

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow 0 \leq a[i] \leq \mathtt{S}\big) \tag{ϕ_1}$$

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow (a[i] \neq 0 \rightarrow r[a[i]] \neq 0)\big) \tag{ϕ_2}$$

$$\forall i\big((1 \leq i \leq \mathtt{V} \wedge a[i] \neq 0) \rightarrow \exists j(1 \leq j \leq \mathtt{C} \wedge b[i,j] = r[a[i]])\big) \tag{ϕ_3}$$

$$\begin{aligned}
&\forall k\big((1 \leq k \leq \mathtt{S} \wedge r[k] \neq 0) \rightarrow \\
&\quad \exists \mathit{count}(\mathit{count}[0] = 0 \wedge \\
&\qquad \forall i(1 \leq i \leq \mathtt{V} \rightarrow (a[i] = k \rightarrow \mathit{count}[i] = \mathit{count}[i-1] + 1) \wedge \\
&\qquad\qquad (a[i] \neq k \rightarrow \mathit{count}[i] = \mathit{count}[i-1])) \wedge \\
&\qquad \mathit{count}[\mathtt{V}] = \mathtt{Q})\big)
\end{aligned} \tag{ϕ_4}$$

Formulas ϕ_1 and ϕ_2 express well-formedness of the partition. Formula ϕ_3 expresses that only votes can support a candidate in which that candidate is somewhere ranked. Formula ϕ_4 expresses that each class supporting a particular elected candidate has exactly $\mathtt{Q}$ elements. To formalise this, we use an array *count* such that $\mathit{count}[i]$ is the number of supporters among votes $1, \ldots, i$ that support the kth elected candidate.

Note, that this criterion assumes all seats to be filled and has to be relaxed if a voting scheme does not satisfy the fill-all-seats criterion or there are not enough candidates that can reach the quota.

3.4. Criterion 2: Election Result Consistent with Preferences

We also consider a second criterion that can be considered orthogonal to Criterion 1. We check that the election result respects the preferences of the majority, as stated in the following definition.

Definition 3.2. (Criterion 2: Election result consistent with preferences) Given a ballot box $\mathcal{b}$, let $R = \bigcup \mathcal{b}$ be the union of the partial linear orders given by the votes $\beta \in \mathcal{b}$, and let R^+ be the transitive closure of R. Then, there is an argument for ranking candidate c_1 over c_2 iff $c_1 R^+ c_2$.

We define Criterion 2 to hold for a preferential voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$ iff, for all ballot boxes $\mathcal{b}$ and for all candidates x, y, the following holds:

If there is an argument for ranking x over y but not for ranking y over x then y must not be ranked higher than x in the election result $\mathcal{T}(\mathcal{b})$.

Note that, in the above definition, R and R^+ may not be order relations.

Formalisation. That there is an argument for ranking x over y implies that there is a sequence c of candidates such that $x = c[0], \ldots, c[k] = y$ and there is a sequence of votes $v[1], \ldots, v[k]$ such that $v[i]$ prefers $c[i-1]$ over $c[i]$ $(1 \leq i \leq k)$.

We formalise that vote $v[i]$ prefers candidate c_1 over candidate c_2 by:

$$\begin{aligned}\phi(v, i, c_1, c_2) \;=\; & \exists j(1 \leq j \leq \mathtt{C} \wedge b[v[i], j] = c_1 \wedge \\ & \forall j'(1 \leq j' < j \rightarrow b[v[i], j'] \neq c_2))\end{aligned}$$

The first line of the above formula says that voter $v[i]$ gives the preference j to candidate c_1. The second line says that v does not give a higher preference $j' < j$ to c_2: i.e., gives c_2 lower preference or no preference at all.

Now, we can formalise that there is an argument for ranking x over y by:

$$\begin{aligned}\Phi(x, y) \;=\; & \exists v \exists c \exists k(x = c[0] \wedge y = c[k] \wedge \\ & \forall i(1 \leq i \leq k \rightarrow (1 \leq v[i] \leq \mathtt{V} \wedge 1 \leq c[i] \leq \mathtt{C} \wedge \\ & \qquad \phi(v, i, c[i-1], c[i]))))\end{aligned}$$

In a similar way as with ϕ, we can formalise the fact that the voting result gives a higher ranking to candidate c_1 than to candidate c_2 as follows:

$$\begin{aligned}\psi(c_1, c_2) \;=\; & \exists j(1 \leq j \leq \mathtt{S} \wedge r[j] = c_1 \wedge \\ & \forall j'(1 \leq j' < j \rightarrow r[j'] \neq c_2))\end{aligned}$$

Using the formulas Φ and ψ, the criterion can be formalised as follows:

$$\forall x \forall y\big((1 \leq x \leq \mathtt{C} \wedge 1 \leq y \leq \mathtt{C} \wedge x \neq y \wedge \Phi(x, y) \wedge \neg\Phi(y, x)) \;\rightarrow\; \neg\psi(y, x)\big)$$

4. Checking Properties of Voting Schemes Using SMT Solvers

4.1. Overview: Different Approaches to Verification with SMT Solvers

In this section we discuss a range of methods that employ Satisfiability Modulo Theories (SMT) solvers for verifying that a voting scheme satisfies any of the aforementioned semantic criteria. Our motivation to chose SMT

solvers for this task is twofold: firstly, SMT solvers have evolved into powerful reasoning tools that are successfully used in model checking and software verification, and secondly, the theories supported by SMT solvers allow us to express semantic criteria easily.

Modern SAT solvers are programs that efficiently decide the satisfiability of a given set of formulas of classical propositional logic [15]. Although this problem is NP-complete, modern SAT solvers can easily solve problem instances with hundreds of propositional variables.

SMT solvers are SAT solvers that are extended by theories meaning they provide domain-specific and highly optimised solvers for arithmetic, arrays, uninterpreted functions, and so on. There are many SMT solvers under active development, such as CVC, MathSAT5, Yices, Z3, etc. and there are annual SMT solver competitions continuously driving the progress regarding theoretical and engineering aspects of SMT solver technology. Among all SMT solvers, Z3 has emerged as a powerful and comprehensive tool that is also practical in that it provides APIs for various programming languages, for example Python. Z3's support for first-order logic over the theories of natural numbers and arrays is outstanding, which we applied to formalize semantic criteria of voting systems. These are the main reasons why we chose Z3 to conduct our experiments.

In the following, we first give an overview of the different ways to apply SMT solvers for checking semantic criteria of voting schemes and present experimental results for the applicability of one of these methods. We address the question of how to represent semantic criteria in the input language of Z3 (using Z3's Python API) and demonstrate feasibility of expressing common semantic criteria at several examples.

To choose a particular technology for the task of verification, we must strike a balance between the ease of use of a particular tool and the quality of the resulting proof. In general, it is not possible to combine both full automation and a full verification, which would be the most desirable result.

Full verification is to provide a general correctness argument for arbitrary vote instances of any size. In order to achieve full verification using SMT solvers, we first need to provide the solver with a logical representation of the voting algorithm (the implementation of the voting scheme). These representations can be derived using off-the-shelf methods, such as weakest precondition generation [16]. As these tools usually cannot derive all loop invariants automatically, they may have to be assisted by a manual, time intensive, error-prone, and sometimes unsuccessful process (see Sec. 4.3). If,

however, the loop invariants are known, SMT solvers can be used to discharge first-order proof obligations,

If we unroll loops in the voting algorithm to a specified finite bound, we speak of *bounded verification*. The advantage over full verification is that, as there are no loops left after unrolling, no manual assistance is required. The disadvantage is that we provide a proof of correctness only for a subset of vote instances.

Perhaps the most automatic but in general least precise method of verification is *bounded model checking*, which exhaustively checks properties for finitely many concrete vote instance up to a certain size. While the scalability of this method is restricted in general, its main strengths are that it neither relies on explicit loop invariants nor weakest precondition generation. It is thus a good candidate for the initial examination of a voting scheme.

4.2. Bounded Model Checking

We examine voting schemes for their semantic criteria by exhaustively testing voting instances using a bounded model checker. That is, we exhaustively run the voting algorithm on the fixed set of input data defined by a given bound and test whether the result produced by each concrete execution of the algorithm satisfies the criterion. If the bounded model checker does not find a bad state, we have established that the criteria are satisfied, which by the small scope hypothesis [17] is not a proof but indicates the absence of programming bugs and conceptual problems. If the model checker finds a bad state, it is possible to extract a counter example for future inspection. Bounded model checking is well understood, and its application to voting schemes was discussed in an earlier paper [7], where linear logic was used to express voting schemes, and bounded model checking was performed via proof search within linear logic.

Here we check the criteria using Z3 by encoding the semantic criteria in first-order logic (plus theories supported by Z3) instead of a linear logic framework. In addition we encode input and output of the individual execution of the voting algorithm, as well as relevant intermediate values of program variables. The generated formula is satisfiable iff the semantic criteria hold for the result produced by the voting algorithm. If Z3 reports that the formula is unsatisfiable, the voting instance serves as a counter-example.

We comment on two scalability issues when using exhaustive testing to verify properties of a voting scheme: (a) the size of the input given to a single test run (e.g., the number of ballots or candidates) and (b) the number

of different possible ballot boxes (and thus test runs needed) given an upper bound on the number of ballots and candidates. Unsurprisingly, exhaustively testing all voting instances for a large number of votes or candidates is intractable. But even the result of a single test run may be difficult to check if the input is large and we require quantification over array elements of arrays whose size depends, e.g., on the number of votes.

If a criterion does not only relate a single ballot box to the result produced by the voting algorithm, but involves multiple ballot boxes, applicability of exhaustive testing is restricted even further. One example for such a criterion is monotonicity (see Sec. 3.1), which relates the ballot box B, the input to the voting algorithm, to another, slightly changed ballot box B'. As a prerequisite to be able to determine whether the semantic criterion holds for a particular voting instance, for all such ballot boxes B' we need to know the result of applying the voting algorithm to B'. While we can still apply instance checking as before by simply running the checks for vote instance B with all possible ballot boxes B', this clearly aggravates the scalability issues.

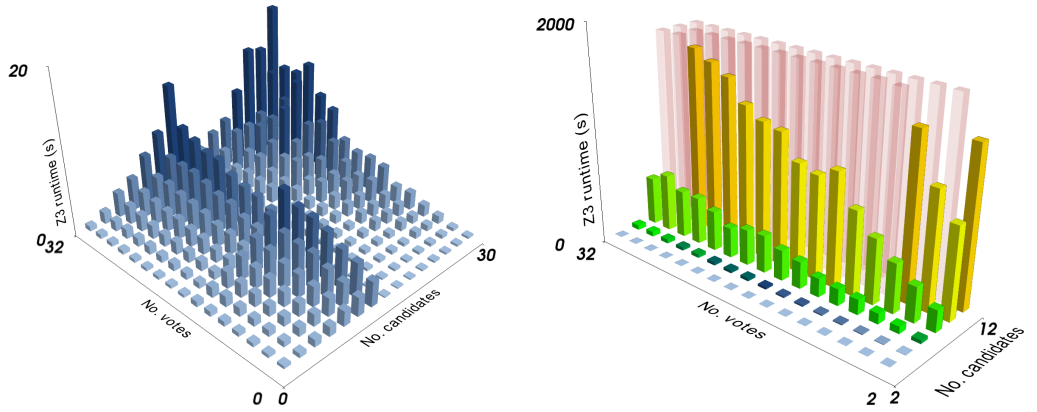
One improvement to this "brute force" approach is to narrow down the search space by generating only ballot boxes B' that actually relate to the vote instance B as required by the semantic criterion (e.g., for monotonicity, enumerate only the B' which result from B by solely raising the preference for a candidate c in one or more of the votes). A different solution to this problem is to use weakest precondition generation as described in Sec. 4.3 to capture the effect of the voting algorithm in first-order logic with theories.

Furthermore, concerning scalability of instance checking, the performance of the SMT solver depends crucially on the way semantic criteria and relevant data of the program run are encoded in first-order logic over theories. The effects of different encodings on performance are shown in the following.

4.2.1. Experiments: Checking Tailor-made Criteria for STV using Z3

To demonstrate that the use of bounded model checking tools is viable, we report here on our experiments on STV implemented in Python using the SMT-based model checker Z3 on the two semantic criteria of STV described in Sections 3.3 and 3.4.

We first produced a straightforward encoding in Z3 of the semantic properties as given in first-order logic. This one-to-one encoding preserves the structure of the original first-order formulas (including all quantifiers) and uses Z3's theories of integers and arrays. Unfortunately, Z3 was not able to handle quantification over the integers in the one-to-one encoding, i.e., Z3



(a) Encoding over bitvectors (no quantifier elimination)

(b) Encoding over integers with quantifier elimination

Figure 1: Z3 performance for checking both semantic criteria (as stated in Sec. 3.3 and Sec. 3.4) of single STV vote instances. The z-axis shows average Z3 run-time in seconds for a single, randomly chosen ballot box with fixed number of candidates and votes. Red, translucent bars indicate that test runs exceeded a timeout of 1800s. Test runs executed on 12-Core AMD Opteron Processor at 2.1 GHz, with 32 Z3 instances running in parallel.

could not determine whether the formula in question was satisfiable or not.

In a second experiment, we eliminated most of the quantifiers in the formula given to Z3 by replacing a universally quantified formula by a conjunction over all instances, and replacing some existentially quantified formulae by providing witnesses (e.g., the array variable a in the first STV criterion), which is possible as the quantifiers range over a small bounded domain.

With quantifiers mostly eliminated, Z3 was able to check the properties for concrete voting instances as shown in Fig. 1b.

A third encoding variant takes advantage of the fact that we are in the bounded case, which means that the integers used in a concrete instance are also of bounded size and can thus be represented by bitvectors of a fixed length. For bitvectors, Z3 provides a strategy good enough to handle formulas with the original structure and without eliminating quantifiers. In fact, as shown in Fig. 1a, for bitvectors, Z3 achieves clearly better performance than reasoning over the integers with quantifiers eliminated. The drop of Z3's runtime at the boundary of 14 candidates respectively votes indicate a change in Z3's proof search strategy – whether this can be further exploited

to improve scalability of our model checker is future work.

Combining the use of bitvectors with quantifier elimination does not lead to further increases in performance. This indicates that Z3's internal handling of quantifiers for the bitvector theory is more sophisticated than what is achieved by our elimination of quantifiers in the formula given to Z3.

Regardless of the encoding, the bounded model checking technique does not scale to the size of ballot boxes found in real elections. It is, however, still useful (due to the small model hypothesis) to pinpoint some of the possible errors in the voting scheme, and to check specific larger instances of interest.

4.2.2. Using Z3 and Its Interface

In this section, we give a brief introduction to Z3, and explain how we use Z3 to check encodings of semantic criteria introduced in Sec. 3.1.

Z3 provides APIs for C/C++, .Net, Python, and OCaml. In this paper, we consider only the Python version. The Z3 API comprises many classes and functions, only some of which are explained here (a complete manual can be found on the Z3 website[3]).

At the class level, the `Solver` class plays a central role. It allows to add assertions, check assertions for consistency, and generate models for consistent assertions.

The API provides functions to construct formulas. For example, the `Int()` and `BitVec()` functions create an integer and a bit-vector constant respectively. The functions `Array()` and `K()` create array variables resp. constants. And the functions `ForAll()`, `Exists()`, `And()`, `Or()`, `Not()` provide logical operators.

Generating a Z3 input formula consists of two parts. One is encoding the voting instance from electoral raw data; the other is formalising the criteria.

A voting instance consists of the voting settings (e.g., the number of candidates, the number of voters, the number of vacant seats, etc.), the ballots, the voting result (elected candidates), and sometimes the intermediate variables such as each ballot's support candidate in STV. Entities such as candidates are represented by integers instead of names.

Examples for constructing voting instances in Z3 using bitvectors and integers, respectively, are shown in Fig. 2. The function `to1DimZ3Array` converts a list of integers into bit-vectors, the function `to2DimZ3Array` is

[3]http://research.microsoft.com/en-us/um/redmond/projects/z3/z3.html

— Voting instance with bitvectors —

```
def to1DimZ3Array(a):
    AllZero = K(Char, BitVecVal(0, BITVECSIZE))
    res = AllZero
    for i, val in enumerate(a):
        res = Update(res, i + 1, val)
    return res

b = to2DimZ3Array(ballots_instance)
r = to1DimZ3Array(result_instance)
```

— Voting instance with bitvectors —

— Voting instance with integers —

```
ballot_sort = ArraySort(IntSort(), IntSort())
b = Array('ballots', IntSort(), ballot_sort)
for i in range(V):
    one_ballot = Array('ballot', IntSort(), IntSort())
    for j in range(C):
        one_ballot = Update(one_ballot, j, ballot_instance[i][j])
    b = Update(b, i, one_ballot)

r = Array('result', IntSort(), IntSort())
for i in range(S+1):
    r = Update(r, i, result_instance[i])
```

— Voting instance with integers —

Figure 2: Formalisation of Voting Instance

defined analogously and hence omitted.

The integer library provides a function `Update()` that we use set values of voting instances. Alternatively, we can use assignments in formulas that are subsequently executed by the solver.

Next we explain the construction of Z3 input formalising criteria. The majority criterion and the first criterion from Sec 3.3 are shown as an example in Figure 3 and 4, respectively. We explain each in turn.

In Figure 3, the definition of `b` (ballot) and `r` (result) are transcribed literally from the formulation of the criterion. In the formula, the universal quantifiers are unfolded into conjunctions (using `And()`), and similarly, existential quantifiers are unfolded into disjunctions (using `Or()`). In line 7, `for c in range(1, C+1)` is used to represent "for all candidates". The body of this loop reads as follows: If a candidate accumulates more than 50% of the votes, he or she will appear in the in the final elected result `r`.

Fig. 4 depicts the formalisation of the first criterion from Sec 3.3 in Z3.

— Majority criterion

```
[ Implies(Exists(count, // count is a counter
                 And(count[0] == 0, count[V] > V/2, // V is total number of voters
                     And([And(Implies(b[i][0] == c, count[i+1] == count[i]+1),
                              Implies(b[i][0] != c, count[i+1] == count[i]))
                          for i in range(V)]))),
          Or([r[j] == c for j in range(1, S+1)])) // S is the number of elected candidates
  for c in range(1, C+1) ] // C is the total number of candidates
```

Majority criterion —

Figure 3: Encoding of the majority criterion

— First criterion

```
F1 = [And(a[i] >= 0, a[i] <= S) for i in range(V)]

F2 = [Implies(a[i] != 0, r[a[i]] != 0) for i in range(V)]

F3 = [Implies(a[i] != 0, Or([b[i][j] == r[a[i]] for j in range(C)]))
      for i in range(V)]

F4 = [ForAll(k,
             Implies(And(1 <= k, k <= S, r[k] != 0),
                     Exists(count,
                            And(count[0] == 0,
                                count[V] == Q,
                                And([And(Implies(a[i] == k, count[i+1] == count[i]+1),
                                         Implies(a[i] != k, count[i+1] == count[i]))
                                     for i in range(V)])))))]
```

First criterion —

Figure 4: Formalisation of First Criteria for STV

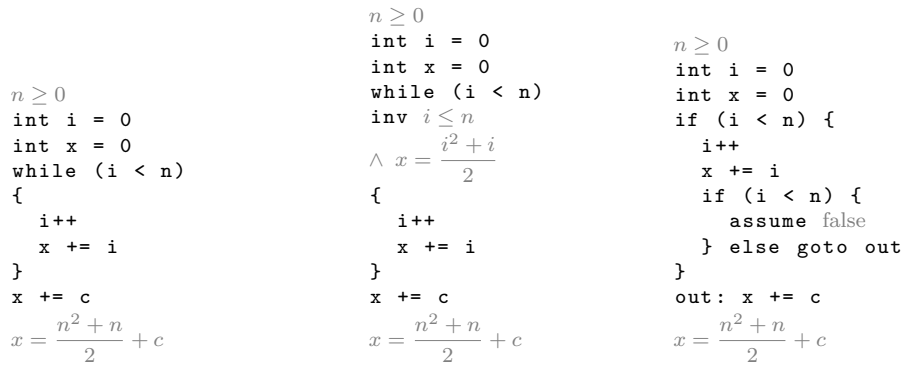
The formalisation is a one-to-one transliteration. Consider, for example, the Z3-formula `F1` that encodes the corresponding first order formula

$$\forall i \big(1 \leq i \leq \mathtt{V} \rightarrow 0 \leq a[i] \leq \mathtt{S} \big) \ .$$

Note that there is an index shift. While the voters' index starts from 1 in the original formalisation, it starts from `0` in the Z3 representation.

4.3. Full and Bounded Verification

Scalability issues of the bounded model checking approach, which requires exhaustive instance checking, call for other techniques that allow to analyze either all voting instances up to a considerable size, or even guarantee correctness independent of the size or concrete content of the ballot box. In



(a) Original program

(b) Program with annotated loop invariant

(c) Program with loops unrolled once

Figure 5: Verification example

the following, we demonstrate two established methods for this purpose on a small sample program.

For this, consider the program to compute the sum of the first n integers plus a constant c (both c and n are input parameters), as shown in Fig. 5a. Besides the actual program (in black), we label (in gray) properties of the program state during program execution. The property $n \geq 0$ is the precondition of the program and $x = (n^2 + n)/2 + c$ its postcondition. The precondition captures in which states we intend this program to be invoked, and the postcondition states what property to expect after the program terminates.

With full verification, we are able to prove that the program adheres to its pre- and postcondition pair (using weakest precondition computation, as explained below). For the unbounded case, the user needs to supply information describing the effect of executing (a variable number of) loop iterations in the program via loop invariants. For our example program, the appropriate invariant is shown in Fig. 5b. Finding the right loop invariant for this example is trivial – for concrete voting algorithms, however, this is a difficult and sometimes infeasible task, as there might not even be a suitable abstraction in form of a concise invariant that describes the loop's effects.

Another option which does not need further user-supplied information is to use bounded verification. Loops in the program are dealt with by examining only program executions up a small number of loop iterations (the *bound*). This allows us to transform loops into if-cascades – as depicted

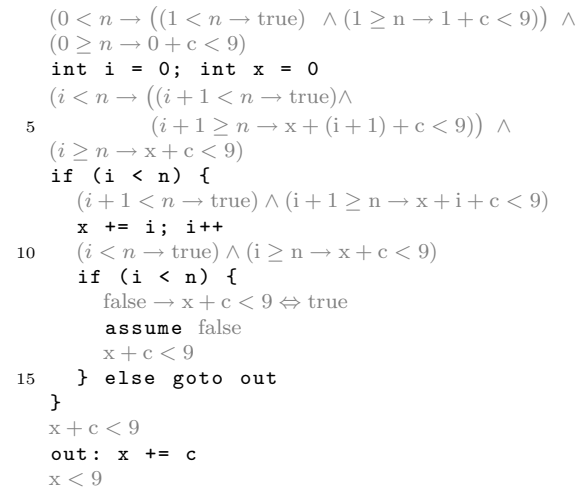


Figure 6: Weakest precondition computation for property: $x < 9$

in Fig. 5c for executions containing at most one loop iteration. The assume statement inserted by the transformation in the second if-block is used for weakest precondition computation: independently from the properties that actually hold at this point in the execution, this statement adds *false* as an assumption so that from this point on, *any* property holds, effectively treating all executions of the original program that pass this point as if they unconditionally establish the postcondition.

Starting from either the original program with annotated invariants or the unrolled program, the next step in showing correctness of the program is to generate the weakest precondition for the given postcondition w.r.t. the program, resulting in a first-order logic formula. The weakest precondition is a property of program states s.t., if the program is started in a state satisfying the weakest precondition, then it terminates in a state that satisfies the postcondition. If the actual precondition of the program implies the weakest precondition, then the program is correct w.r.t. the given pre-/postcondition pair. Whether this implication holds is typically checked automatically using an SMT solver.

For our example program, Fig. 6 shows the intermediate properties resulting from weakest precondition computation for a simple postcondition P: $x < 9$. For P to hold after execution of the final statement `x+=c` in line 18 of the program, P with all occurrences of x replaced by $x + c$ has to be true beforehand. Corresponding rules for the other statement types are applied to

all statements of the program to get the weakest precondition of the program w.r.t. P, as seen in lines 1-2 in Fig. 6. The constructed weakest precondition is equivalent to $(n = 1 \rightarrow c < 8) \wedge (n \leq 0 \rightarrow c < 9)$.

As explained, bounded verification only guarantees correctness up to a given number of loop iterations. Thus for parameters that affect the number of loop iterations in the program execution (in our example: n), we only get correctness results for a subset of values. In the weakest precondition for our example program in Fig. 6, the sub-formula $(1 < n \rightarrow \text{true})$ captures this: for any concrete value for n greater than one, the whole formula simply evaluates to true, although the postcondition $x < 9$ actually may not hold.

With the number of loop iterations depending on the values of input parameters, we get similar scalability issues as described for bounded model checking, if we want to examine larger parameter values. In contrast to bounded model checking, however, for variables not affecting the number of loop iterations, the result of bounded verification applies to *all* values, as variables are handled symbolically by the technique. Additionally, on this abstract level, analysis of program properties might be simplified by exploiting symmetries in the program behaviour. For these reasons, bounded verification is a promising approach to check properties of voting schemes.

5. Case Study: Variants of the Single Transferable Vote Scheme

Single transferable vote (STV) is a preferential voting scheme [18] for multi-member constituencies aiming to achieve proportional representation according to the voters' preferences.

5.1. The Standard Version of STV

There are many versions of STV, but most are an extension or variant of the standard version that is shown in Figure 7.

For input and output of the algorithm, we use the same notation and encoding as in Section 3. There are `V` voters electing `S` of `C` candidates, and:

- b: is the input ballot box, where $b[i, j]$ is the number of the candidate that is ranked by vote i in the jth place. If the vote does not rank all candidates, then $b[i, j] = 0$ for the empty places.
- r: is the output election result, where $r[i]$ is the ith candidate that is elected ($1 \leq i \leq$ `S`). If less than `S` candidates are elected, then $r[i] = 0$ for the empty seats.

We assume the input for the algorithm to satisfy the following conditions (which are pre-conditions for running the standard STV algorithm): (1) $\mathtt{C} \geq \mathtt{S}$, (2) $\mathtt{V} \geq 1$. and (3) votes are linear orders of a subset of the candidates, i.e., for all $1 \leq i \leq \mathtt{V}$ and all $1 \leq j, j' \leq \mathtt{C}$:

- $0 \leq b[i,j] \leq \mathtt{C}$,
- if $b[i,j] \neq 0$ and $j \neq j'$ then $b[i,j] \neq b[i,j']$,
- if $b[i,j] = 0$ then $b[i,j'] = 0$ for all $j' \geq j$.

The initialisation part of the STV algorithm, in particular, computes a quota necessary to obtain a seat (line 5). Different definitions of quotas are used in practice. The most common is the Droop quota $\mathtt{Q} = \lfloor \mathtt{V}/(\mathtt{S}+1) \rfloor + 1$.

To determine the election result, STV uses an iterative process, which repeats the following two steps until either a winner is found for every seat or the number of remaining candidates equals the number of open seats (lines 10–33).

1. If no candidate reaches the quota of first-preference votes, a candidate with a minimal number of first-preference votes is eliminated and that candidate is deleted from all ballots (lines 17–19).
2. Otherwise one of the candidates with `Q` or more first-preference votes is chosen (line 23) and declared elected (line 24). Of the first-preference votes for that candidate, `Q` are chosen and erased (lines 26–29). These are the votes that are considered to have been "used up". If the candidate has more than `Q` votes, the surplus votes remain in the ballot box. Finally, the elected candidate is deleted from all ballots still in the box.

The procedure for deleting a candidate c (lines 40–47) works by searching for the candidate in each vote and, if c is found to have preference j, then the candidate with preference $j+1$ moves to preference j, the candidate with preference $j+2$ moves to preference $j+1$, and so on.

When the main loop of the standard STV algorithm as shown in Figure 7 terminates, either (a) all seats are filled, or (b) the number `cc` of remaining candidates is equal to the number of open seats. In case (b), a further step is needed to distribute some or all of the remaining candidates to the equal number of remaining seats. The default is to fill all the remaining seats with the remaining candidates (line 37). Alternatively, one may continue the main STV loop to see if the further candidates get elected (which may leave seats open).

— Standard Version of STV —

```
1  // Initialisation
2  r  := [0, ..., 0];            // no one elected yet
3  e  := 1;                      // e is the next seat to be filled
4  cc := C;                      // cc is the number of (continuing) candidates
5  Q  := ⌊V/(S + 1)⌋ + 1;        // Droop quota

7  // Main loop: While not all seats filled and
8  //            there are more continuing candidates than open seats
9  // In each iteration one candidate is elected or one candidate eliminated
10 while (e ≤ S) ∧ (cc > S − e + 1) do
11     // QuotaReached is the set of candidates for which the number of
12     // first-preference votes reaches or exceeds the quota Q
13     QuotaReached := {c | 1 ≤ c ≤ C ∧ #{v | 1 ≤ v ≤ V ∧ b[v, 1] = c} ≥ Q};
14     if QuotaReached = ∅ then
15         // no one has reached the quota,
16         // eliminate a weakest candidate by deletion from the ballot box
17         Weakest := {c | 1 ≤ c ≤ C ∧ #{v | 1 ≤ v ≤ V ∧ b[v, 1] = c} is minimal};
18         choose c ∈ Weakest;
19         delete(c);
20     else
21         // one or more candidates have reached the quota,
22         // elect one of them
23         choose c ∈ QuotaReached;
24         r[e] := c;  // put c in the next free seat
25         e := e + 1; // increase the number e of the next seat to be filled
26         do Q times                                  // Q of the votes that
27             choose i ∈ {i | 1 ≤ i ≤ V ∧ b[i, 1] = c};  // give c top preference
28             for j = 1 to C do b[i, j] := 0; od      // get erased
29         od
30         delete(c);   // delete c from the ballot box
31     fi
32     cc := cc − 1;    // in any case we have one less continuing candidate
33 od

35 // Fill the empty seats
36 if e < S then
37     fill the remaining seats r[e, ..., S] with the remaining cc candidates

39 // procedure for deleting candidate c from votes in b
40 procedure delete(c) begin
41     for i = 1 to V do for j = 1 to C do
42         if b[i, j] = c then
43             for k = j to C − 1 do b[i, k] := b[i, k + 1] od;
44             b[i, C] := 0;
45         fi
46     od od
47 end
```

— Standard Version of STV —

Figure 7: The standard STV algorithm

Example 5.1. We consider the same situation as in Example 3.1, i.e., there are four candidates A, B, C, D for two vacant seats, and the votes to be counted are $[A, B, D], [A, B, D], [A, B, D], [D, C], [C, D]$. The Droop quota in this case is $\mathtt{Q} = \lfloor 5/(2+1) \rfloor + 1 = 2$.

In the first iteration of the main loop, candidate A meets the quota and is hence elected. Two of the votes $[A, B, D]$ are erased, the third is a surplus vote. It is transformed into $[B, D]$ by deleting A from the ballots.

In the second iteration no candidate reaches the quota, thus the weakest of the remaining candidates B, C, D is eliminated – which one depends on the kind of tie-breaker used as all three have exactly one first-preference vote at that point. (1) If the tie-break eliminates B, the aforementioned transformed vote $[B, D]$ will be transformed again and will become a vote for D, so that D will be elected in the next iteration. (2) If the tie-break eliminates C, the vote $[C, D]$ will be transformed into a vote for D, and thus D will be elected. (3) If the tie-break eliminates D, then C will be elected, analogously, in the next iteration. In summary, the algorithm reports either $[A, D]$ or $[A, C]$ as the election result but not, for example, $[A, B]$ or $[B, D]$. If the number of second-preference votes is used as a tie-breaker, then B is eliminated first (case 1 above).

The standard STV algorithm has three choice points that produce non-determinism. Different variants of STV resolve them in different way:

1. Who is eliminated if several candidates have the same minimal number of first preferences (line 18)?
2. Who is elected if several candidates have reached the quota (line 23)?
3. How are the votes chosen that are deleted when an elected candidate has more than quote votes (line 27)?

Choice points (1) and (2) are typically handled – to some extent at least – by defining various kinds of tie-break rules. They can also be handled by declaring all weakest candidates eliminated resp. declaring all strongest candidates elected. That, however, is not always possible (there may not be enough open seats). And it can affect the election result in unexpected ways.

Choice point (3) can be eliminated using the notion of fractional votes. Instead of erasing a fraction of the votes that needs to be chosen, the same fraction of each vote is erased and the remaining fraction remains in the ballot box. This is done in many versions of STV used in real-world elections.

The above considerations illustrate that the STV algorithm as presented in this section is not only one but an entire family of vote counting algorithms.

There are a number of parameters to play with: the quota, the choice of tie-breakers, placement of candidates once there are as many free seats as remaining candidates.

There are further options that – we argue in Section 5.2 – lead to election systems that can no longer be considered part of the STV family.

5.2. The CADE-STV Election Scheme

The bylaws of the Conference on Automated Deduction (CADE) specify an algorithm for counting the ballots cast for the election of members to its Board of Trustees [19]. The intention of the bylaws is to design a voting algorithm that takes the voters' preferences into account. The algorithm has been implemented in Java and used by several CADE Presidents and Secretaries in elections for the CADE Board of Trustees. It has also been used by TABLEAUX Steering Committee Presidents, including one of the authors, for the election of members to the TABLEAUX Steering Committee.

Pseudo-code for the CADE-STV scheme is included in the CADE bylaws [19], making it an interesting target for formal analysis. CADE-STV differs from the standard version of STV (shown in Figure 7) in several ways:

Quota Instead of the Droop quota, CADE-STV uses a quota of 50% of the votes – independently of the number of seats to be filled. That is, line 5 in Fig. 7 is changed to "`Q := `$round$(`V`/2)".

Empty seats CADE-STV does not fill seats that remain open at the end of the main loop, i.e., lines 36–37 are removed, and "cc > S - e + 1" in lines 10 is changed to "cc > 0".

Restart Each time a candidates c reaches the quota `Q` of first-preference votes and gets elected, the election for the next seat restarts with the original ballot box – with the only exception that the elected candidate c is deleted. Thus, (a) the `Q` votes used to elect c are not erased but are only changed by deleting c, and (b) weak candidates that have been eliminated are "resurrected" and take part in the election again. That is, (a) the code for erasing votes (lines 26–29) is removed and (b) replaced by code for resurrecting the eliminated candidates.

5.3. Applying Bounded Model Checking to CADE-STV

As already explained in Section 4.2, we have applied SMT-based bounded model checking to CADE-STV. There, we drew conclusion regarding the

applicability of bounded model checking and the use of Z3. In this section, we report what can be concluded from the experiments w.r.t. properties of CADE-STV.

First, the experiments validated that CADE-STV (like standard STV) satisfies Criterion 2 (Def. 3.2). But since an exhaustive model search was only done up to a small bound on the number of votes and number of candidates, this does not constitute a full proof.

Second, running our bounded model checker on CADE-STV confirms that, in difference to standard STV, CADE-STV does not satisfy Criterion 1 (Def. 3.1), which is closely related to proportional representation.

In addition, we have used SMT-based bounded model checking to check the correctness ratio of CADE-STV voting instances. In this experiment, we generated all possible ballot boxes for CADE-STV up to a certain size and used Z3 to check if the CADE-STV counting results meet Criterion 1 for that instance.

As an example, the number of candidates is fixed to 4, the number of vacant seats is varied from 1 to 3 and the number of voters is set from 3 to 5. The result of the experiment is shown in Tab. 1. All possible ballot instances for 4 voters and 2 vacant seats are tested, while for others only the first 100,000 ballot instances are checked. And when CADE-STV fails to seat all the vacant seats, the voting instance is not counted in the correct ratio.

For the one-vacant-seat case, the criterion is met by CADE-STV constantly, because there is no difference between standard STV and CADE-STV (unless no candidate reaches the quota, in which case standard STV picks a random candidate while CADE-STV leaves the seat empty). When there are more than two vacant seats, the criterion is unsatisfiable with CADE-STV because there not enough ballots to support all elected candidates as the quota is 50%. For the two candidates case, CADE-STV sometimes does not reuse the ballot when picking the second candidate, and in these cases the criterion can be satisfied.

5.4. Effects of the Differences between CADE-STV and Standard STV

5.4.1. Effects of Restart

To illustrate the effect of the restart mechanism in CADE-STV on the election result, we consider an example:

Voters \Vacant Seats	1	2	3
3	100%	0%	0%
4	100%	87.2%	0%
5	100%	0%	0%

Table 1: Test First STV Criterion on CADE-STV

Example 5.2. Let us run CADE-STV on Example 3.1. First, we compute the majority quota $\mathtt{Q} = 3$. In the first iteration, A has three first preferences, so that A is the majority winner and is seated. Since CADE-STV uses restart, A's votes are not deleted but are redistributed at the end of the first iteration. Now the ballot box contains $[B, D], [B, D], [B, D], [D, C], [C, D]$. Following the algorithm, we observe that now B is the majority candidate with 3 first preference votes and is seated. The election is over, and the election result is $[A, B]$ (which is different from the possible results $[A, D]$ or $[A, C]$ of standard STV).

Indeed, our bounded model checker finds smaller counter examples than the one shown in Example 5.2, but these are not as illustrative.

The effect of the differences between standard STV and CADE-STV is further clarified by the following theorem and its corollary: in certain cases, there is no proportional representation in the election results computed by CADE-STV. See also Example 5.3 below.

Theorem 5.1. *If a majority of voters vote in exactly the same way with ballot $[c_1, \ldots, c_k]$, then CADE-STV will elect the candidates $c_1, \ldots, c_k$ preferred by that majority in order of the majority's preference.*

Proof. Since a majority of voters choose c_1 as their first preference, no other candidate can meet the "majority quota". Thus c_1 is elected in the first round. When redistributing the ballots, each of the majority of ballots with c_1 as first preference have c_2 as second preference. All become first preferences for c_2. Thus candidate c_2 is guaranteed to have a majority of first preferences and is elected in round two, and so on until all vacancies are filled. □

Corollary 1. *If the electorate consists of two diametrically opposed camps that vote for their candidates only, in some fixed order, then the camp with a majority will always get their candidates elected and the camp with a minority will never get their candidate elected.*

Standard STV does not use the restart mechanism and so it will elect the first ranked candidate of the majority, but will then reuse only the surplus votes and not all votes as done by CADE-STV. Thus the second preference from the majority is not necessarily the second person elected. Consequently, majorities do not rule outright in standard STV.

5.4.2. Effects of High Quota and No Filling of Empty Seats

No matter how many candidates there are and how many seats need to be filled, a candidate can only be seated by CADE-STV if he or she accumulates more than 50% of the votes. Any candidate with less than 50% of the vote is defeated. Thus, CADE-STV obviously violates the fill-all-seats criterion. But because of the high quota it also prevents proportional representation as candidates supported by a large minority can neither be elected via reaching the quota nor via filling seats left empty at the end of the main loop.

In fact, if the high quota of 50% and no filling of empty seats were the only changes w.r.t. standard STV, only a single candidate could be elected because more than 50% of the votes would be used up by electing that candidate. CADE-STV requires the restart mechanism to elect further candidates.

Example 5.3. Assume there are 100 seats and two parties nominating candidates $A_1, \ldots, A_{100}$ and $B_1, \ldots, B_{100}$, respectively. Further assume that there are 51% of A-voters and 49% of B-voters. All A-voters vote $[A_1, \ldots, A_{100}]$ and B-voters vote $[B_1, \ldots, B_{100}]$. Standard STV elects $A_1, \ldots, A_{51}, B_1, \ldots, B_{49}$, i.e., the result is a perfect proportional representation.

With a quota of 50% and no filling of empty seats, only A_1 gets elected and then nothing further happens, which is clearly undesirable. But CADE-STV also uses the restart mechanism, therefore, like standard STV, it fills all seats. The result is different, however, because the votes used to elect $A_1, \ldots, A_{51}$ do not get erased. CADE-STV produces the election result $[A_1, ..., A_{100}]$.

The above example again shows that the majority can rule with CADE-STV and there is no proportional representation in that case (Corollary 1).

5.5. Observations on the History of CADE-STV

We discuss the history of the CADE-STV scheme because it illustrates the problem of evolving an election scheme without using formally specified semantic criteria and a formal definition of the input to the scheme. It is publicly known that there were lots of discussions among the CADE Trustees over a long period of evolving CADE-STV. But we do not know what the

non-public deliberations actually where. The following is based on our interpretation of the publicly available material.

5.5.1. The Violation of Proportional Representation

The CADE-STV voting scheme is the result of a long discussion among the board of trustees that took place in the years 1994–1996. David A. Plaisted published various concerns about the existing voting scheme which can be found on his homepage [20].

One of Plaisted's concerns was that a minority supporting candidates standing for re-election could re-elect these candidates against the wishes of the majority as that majority is not sufficiently coordinated in its behaviour to elect alternative candidates [20]:

> *Of course, one of the main purposes of a democratic scheme is to permit the membership to vote a change in the leadership if there is a need for this. However, the new bylaws make this more difficult in several ways. The problem is that those who are unsatisfied with the scheme will tend to split their votes among many candidates (unless they are so disgusted as to put the trustee candidates at the very bottom of the list), but those who are satisfied will tend to vote for the trustee nominees. This means that the trustee nominees tend to be elected even if only a minority is happy with the scheme.*

We believe that because of Plaisted's concerns the board introduced the high 50% quota and did not include a mechanism for filling seats that remain empty. On first sight, this seems good because it solves the problem illustrated in Plaisted's scenario. But, as explained above, this deviation from the standard STV setup not only violates the fill-all-seats criterion but also the goal of proportional representation (see Example 5.3). Thus, the CADE-STV scheme protects the majority at the expense of the minority.

Also, as explained above, if the high quota and the remaining empty seats were the only changes, only a single candidate could be elected. So, in effect, one was forced to change the algorithm further. The result was that the restart mechanism was added to the algorithm, that reuses the original ballot box for each seat and does not erase votes (because then more candidates can be elected, see Example 5.3).

There would have been a different solution than using a restart that would have solved Plaisted's problem without restricting proportional representa-

tion as much: One could have used Standard STV with an additional rule that – before the main algorithm is started – anybody who does not appear (with arbitrary preference) on at least 50% of the votes is immediately eliminated.

Example 5.4. Using the same input ballots as in Example 5.3, the algorithm would then elect $[A_1, ..., A_{51}]$, which still suppresses the B minority, but at least gives the A party only those seats that are proportional to the A votes.

5.5.2. Well-formedness and Interpretation of Input

Apparently, during some CADE elections, there was some confusion about the meaning of not listing a candidate at all on a ballot and how that should be translated into input for the CADE-STV voting scheme.

The instruction was given to the voters that not listing a candidate is the same as giving that candidate the lowest possible preference. But that is not the correct interpretation. It is easy to see that for both standard STV and CADE-STV, there is a difference between giving a candidate the lowest possible preference and not listing the candidate at all. For example, if there are candidates A, B, C, then $[A, B]$ is different from $[A, B, C]$. When candidates A and B get eliminated, $[A, B, C]$ turns into a vote for C and may help to elect C, which $[A, B]$ does not. One could transform a ballot of the form $[A, B]$ into an input vote $[A, B, C]$ (and, thus, make them equal by definition). But that only works if a single candidate is missing from the ballot. If more are missing, they would have to be put in the same spot on the ballot, which is not possible. Indeed, CADE-STV does not work correctly if input votes contain candidates with equal preference, i.e., if the pre-condition that a vote is a partial linear order is violated. As that pre-condition was never clearly specified, fixing the problem in CADE-STV was a lengthy process that took several years.

This shows that not formalising the pre-conditions which the input must satisfy is problematic. Besides the possibility of errors or unintended behaviour of the algorithm, it is important that the voters understand how their ballot is transformed into input for the algorithm.

6. Conclusion

We have discussed semantic criteria to formalize desired properties of voting schemes. Formal specification and verification do not provide a mech-

anism for deciding which properties are desirable for a particular vote. But they are methods for the analysis and development of voting schemes.

Our case study demonstrates the importance of formal criteria both for analysis of voting schemes and their evolution and the development process. Semantic criteria need to be explicitly stated. A discussion of voting schemes using anecdotal descriptions of individual voting scenarios is not a good basis for making electoral laws.

Furthermore, we demonstrated that SMT solvers are well-suited to discover bugs in voting schemes. In particular, we have shown that the formalisation of semantic criteria in first-order logic over the theories of integers and arrays is a good choice for SMT-based analysis.

Future work is to improve the reach and the efficiency of SMT-based analysis as described in Sec. 4. This will allow us to investigate larger classes of voting schemes and to use more complex criteria. We also plan to extend our analysis to criteria that measure the quality of election results based on difference measures [21] in addition to yes/no criteria.

References

[1] F. Brandt, V. Conitzer, U. Endriss, Computational social choice, in: G. Weiss (Ed.), Multiagent Systems, MIT Press, 2012, forthcoming. Available at `http://www.illc.uva.nl/~ulle/pubs/files/BrandtEtAlMAS2012.pdf`.

[2] S. Brams, R. Sanver, Voter sovereignty and election outcomes, Retrieved from `http://www.nyu.edu/gsas/dept/politics/faculty/brams/sovereignty.pdf`, accessed 21 March 2013 (2003).

[3] I. D. Hill, B. A. Wichmann, D. R. Woodall, Single transferable vote by Meek's method, Computer Journal 30 (3).

[4] F. C. Court, Provisions of the federal electoral act from which the effect of negative voting weight emerges unconstitutional, Press release no. 68/2008 (2008).

[5] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proceedings, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer-Verlag, 2008, pp. 337–340.

[6] B. Beckert, R. Goré, C. Schürmann, On the specification and verification of voting schemes, in: 4th International Conference, Vote-ID 2013, Guildford, UK, LNCS, Springer, 2013.

[7] B. Beckert, R. Goré, C. Schürmann, Analysing vote counting algorithms via logic. And its application to the cade election system, in: Proceedings, 24th International Conference on Automated Deduction (CADE), Lake Placid, NY, USA, LNCS, Springer, 2013.

[8] E. Pacuit, Voting methods, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, winter 2012 Edition, 2012.

[9] K. J. Arrow, A difficulty in the concept of social welfare, Journal of Political Economy 58 (1950) 328.

[10] Y. Sun, C. Zhang, J. Pang, B. Alcalde, S. Mauw, A trust-augmented voting scheme for collaborative privacy management, Journal of Computer Security 20 (4) (2012) 437–459.

[11] J. P. G. M McGaley, Electronic voting: A safety critical system, Tech. Rep. NUIM-CS-TR2003-02, Department of Computer Science, National University of Ireland, Maynooth (March 2003).

[12] D. C. J R. Kiniry, P. E. Tierney, Verification-centric realization of electronic vote counting, Tech. rep., School of Computer Science and Informatics, University College Dublin (2007).

[13] D. Cochran, Formal specification and analysis of danish and irish ballot counting algorithms, Ph.D. thesis, ITU (2012).

[14] Wikipedia, Monotonicity criterion, Retrieved from `http://en.wikipedia.org/wiki/Monotonicity_criterion`, accessed 21 March 2013.

[15] N. Eén, N. Sörensson, An extensible SAT solver, in: Theory and Applications of Satisfiability Testing, Springer, 2004, pp. 502–518.

[16] M. Barnett, K. R. M. Leino, Weakest-precondition of unstructured programs, in: ACM SIGSOFT Software Engineering Notes, Vol. 31, ACM, 2005, pp. 82–87.

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

[18] Wikipedia, Single transferable vote, Retrieved from http://en.wikipedia.org/wiki/Single_transferable_vote, accessed 20 Jan 2013.

[19] CADE Inc., CADE Bylaws (effective Nov. 1, 1996; amended July/August 2000), Retrieved from http://www.cadeinc.org/Bylaws.html, accessed 20 Jan 2013.

[20] D. A. Plaisted, A consideration of the new cade bylaws, Retrieved from http://www.cs.unc.edu/Research/mi/consideration.html, accessed 22 Mar 2013.

[21] T. Meskanen, H. Nurmi, Closeness counts in social choice, in: M. Braham, F. Steffen (Eds.), Power, Freedom, and Voting, Springer, 2008.