# Automated Verification for Functional and Relational Properties of Voting Rules

Bernhard Beckert, Thorsten Bormer, Michael Kirsten,
Till Neuber, and Mattias Ulbrich

**Abstract**

In this paper, we formalise classes of axiomatic properties for voting rules, discuss their characteristics, and show how symmetry properties can be exploited in the verification of other properties. Following that, we describe how automated verification methods such as software bounded model checking and deductive verification can be used to verify implementations of voting rules. We present a case study, where we use and compare different approaches to verify that plurality voting satisfies the majority and the anonymity property.

## 1 Introduction

Axiomatic definitions of characteristic properties for voting rules are important both for understanding existing voting rules and for designing new ones. For the trustworthiness of elections, it is also important to analyse the *algorithmic* descriptions of voting rules and to prove that they have the desired properties, i.e., they satisfy the *declarative* description.

There are two classes of voting rule properties, namely (a) functional (absolute) properties, such as the majority criterion, which refer to single voting results, and (b) relational properties, such as anonymity, which are defined by comparing two (or more) results.

In this paper, we first formally define these two classes of properties and discuss their characteristics (Section 2). We then show how symmetry properties – which are a particular kind of relational properties – can be exploited to more efficiently verify voting rules w.r.t. other (functional) properties (Section 3).

Proving properties of voting rules by hand is tedious and error-prone – in particular, if the rules and properties evolve during design and various versions have to be verified. In Section 4, we describe how automated verification methods like software bounded model checking and deductive verification can be used to verify that implementations of voting rules satisfy functional and relational properties. Implementations can be pseudo-code algorithms derived from informal (e.g., legal) text or programs written in languages like C or Java.

In Section 5, we present a case study, where we use and compare different approaches to verify that plurality voting satisfies the majority and the anonymity property.

We discuss related work in Section 6 and, finally, draw conclusions in Section 7.

## 2 Describing Properties of Voting Rules

### 2.1 Voting Rules

We consider voting rules as instances of preference aggregation problems, where the individual preferences of voters are combined to produce an election result. The input for the voting rule is modelled as a finite sequence of ballots, with one ballot for each voter.

**Definition 1.** Given a set $\mathcal{B}$ of possible ballots and a set $\mathcal{W}$ of possible election results, a *voting rule* is a total function $T : \mathcal{B}^* \to \mathcal{W}$, assigning an election result to each profile, where a *profile* $B \in \mathcal{B}^*$ is a finite sequence of ballots.

The special election result $\perp$ may be included in $\mathcal{W}$ to denote that there is no "valid" result (e.g., in case of a tie).

An individual pair $(B, W) \in (\mathcal{B}^* \times \mathcal{W})$ consisting of a profile and an election result is called an *evaluation.* The set of all evaluations is $\mathcal{E} = \mathcal{B}^* \times \mathcal{W}$.

The concrete structure of ballots $b \in \mathcal{B}$, how ballots encode voters' preferences, and the structure of possible election results in $\mathcal{W}$ depend on the investigated voting rule. In examples throughout this paper and in the case study in Section 5, we use preferential voting rules with single winners:

**Definition 2.** Let $\mathcal{C}$ be a finite set of *candidates.* A voting rule is called *preferential* if the possible ballots $\succ_b \in \mathcal{B}$ are linear orders on $\mathcal{C}$ and the possible results are single candidates or a tie, i.e., $\mathcal{W} = \mathcal{C} \cup \{\perp\}$.

## 2.2 Functional and Relational Properties of Voting Rules

In the following, we define and discuss different types of properties for voting rules. In particular, we distinguish between *functional* and *relational* properties. In social choice theory, functional and relational properties are known as *intra-* and *interprofile* conditions respectively [17]. More precisely, functional properties denote the class of *active* intra-profile conditions and relational properties the subclass of *two-profile* conditions (further dividing the class of interprofile conditions).

**Definition 3** (Functional property)**.** Given a set $\mathcal{B}$ of possible ballots and a set $\mathcal{W}$ of possible results, a *functional property* $F$ for voting rules is a set of evaluations, i.e., $F \subseteq \mathcal{E} = (\mathcal{B}^* \times \mathcal{W})$ is a relation between profiles and results. The set of all functional properties is denoted by $\mathcal{F} = \mathbb{P}(\mathcal{E})$.

A voting rule $T : \mathcal{B}^* \to \mathcal{W}$ *satisfies a functional property* $F$ iff $T \subseteq F$, i.e., all evaluations of $T$ are elements of $F$. Intuitively, a functional property $F$ is the set of those evaluations that a voting rule may contain if it is to have that property.

*Example* 1 (Majority criterion, majority winner)*.* We consider preferential voting rules (Def. 2). Given a profile $B \in \mathcal{B}^*$, a majority winner for $B$ is a candidate $c \in \mathcal{C}$ that is preferred over all other candidates in more than half of the ballots:

$$|\{\succ_b \in B \mid c \succ_b c' \text{ for all } c' \in \mathcal{C}, c' \neq c\}| > \tfrac{|B|}{2} \ .$$

A voting rule satisfies the majority criterion iff, for all profiles $B$, either the majority winner for $B$ is elected or there is no majority winner for $B$. This criterion is formalised by the functional property

$$\mathit{Maj} \quad = \quad \{(B, c) \mid \text{there is no majority winner } c' \text{ for } B \text{ with } c' \neq c\} \ .$$

**Definition 4** (Relational property)**.** Given a set $\mathcal{B}$ of possible ballots and a set $\mathcal{W}$ of possible results, a *relational property* $R$ for voting rules is a set of pairs of evaluations, i.e.,

$$R \subseteq \mathcal{E} \times \mathcal{E} = (\mathcal{B}^* \times \mathcal{W}) \times (\mathcal{B}^* \times \mathcal{W}) \ .$$

The set of all relational properties is denoted by $\mathcal{R} = \mathbb{P}(\mathcal{E}^2) = \mathbb{P}((\mathcal{B}^* \times \mathcal{W}) \times (\mathcal{B}^* \times \mathcal{W}))$.

A voting rule $T : \mathcal{B}^* \to \mathcal{W}$ *satisfies a relational property* $R \in \mathcal{R}$ iff $T \times T \subseteq R$, i.e., if for all evaluations $e \in T$ and $e' \in T$, the pair $(e, e')$ is in $R$.

Intuitively, a relational property $R$ consists of those pairs of evaluations that – by definition of that property – are allowed to "co-exist" in a voting rule.

*Example* 2 (Monotonicity criterion [17]). Again, we consider preferential voting rules (Def. 2). For the monotonicity criterion, we need to compare profiles that are identical up to one ballot. By $b^{\uparrow c} \subset \mathcal{B}$ we denote the set of all ballots that are identical to $b$ except that now $c \in \mathcal{C}$ is raised in preference.

The relational property of monotonicity is

$$\begin{aligned} \mathit{Mono} \;=\; (\mathcal{E} \times \mathcal{E}) \setminus \{((B, c), (B', c')) \mid\; & B' \text{ results from } B \text{ by replacing} \\ & \text{a single ballot } b \in B \text{ by a ballot } b' \in b^{\uparrow c}, \\ & c \neq \perp \text{ and } c' \neq c\} \ . \end{aligned}$$

That is, *Mono* contains *all* pairs of evaluations *except* those where the winning candidate $c \neq \perp$ is given higher preference in one of the ballots and then some other candidate $c'$ different from $c$ wins.

A functional property consists of single evaluations, namely those evaluations that are considered "good" by the property. A voting rule is judged against the functional property for every evaluation separately. In contrast to that, a relational property sets evaluations of the voting rule into relation. Satisfaction of a relational property is judged by considering each of its evaluations in the context spanned by all other evaluations.

Thus, the concept of relational properties is stronger and more expressive. In fact, every functional property can also be represented as a relational property: For a functional property $F \in \mathcal{F}$, the relational property $R = F \times F \in \mathcal{R}$ is satisfied by any voting rule $T$ (Def. 4) iff $F$ is satisfied by $T$ (Def. 3).

The distinction between functional and relational properties does not cover all interesting properties of voting rules. Only those properties of voting rules are covered that can be checked by looking at one (functional) or two (relational) evaluations at a time. However, there are properties that require to compare three or more evaluations.

*Example* 3 (Consistency criterion [33]). A voting rule satisfies the consistency criterion if, for any three profiles $B, B_1, B_2$ such that $B$ is the concatenation of $B_1$ and $B_2$:

$$\text{if } \; T(B_1) = T(B_2) \; \text{ then } \; T(B) = T(B_1) = T(B_2) \ .$$

Properties such as consistency, which can (only) be defined by the comparison of three evaluations, are called 3-properties. This concept can be extended to $k$-properties for $k \in \mathbb{N}$. In classic social choice theory, the class of *multiprofile* conditions [17] corresponds to the union of all $k$-properties for $k > 2$. However, this characterisation still does not cover all properties. For example, the non-imposition property, which requires that for each possible election result there is a profile leading to that result (surjectivity of the voting rule), is a rather simple property that is not a $k$-property for any $k$. Non-imposition is existential in nature, requiring the existence of (combinations of) certain evaluations, while all $k$-properties are universal in nature, requiring that all $k$-tuples of evaluations are "good" in some sense.

The taxonomy of properties can be further extended to such existential properties and combinations of existential and universal properties, for which Fishburn defined the separate class of *existential* conditions [17]. However, while a further investigation of the taxonomy of voting rule properties with respect to their arity and nature is very interesting, such an investigation is beyond the scope of this paper.

## 2.3 Symmetry Properties

An important kind of relational properties includes those expressing that, if two profiles are symmetric (or in some other way similar) to each other, then they lead to symmetric (similar) election results. Many fairness criteria for voting rules are of this type.

**Definition 5** (Symmetry property)**.** A *symmetry property* is a relational property

$$S_{\sim,\approx} \;=\; \{((B,c),(B',c')) \mid B \sim B' \text{ implies } c \approx c'\}\;\;,$$

where $\sim\, \subseteq \mathcal{B}^* \times \mathcal{B}^*$ and $\approx\, \subseteq \mathcal{W} \times \mathcal{W}$ are binary relations on profiles resp. election results.

A voting rule satisfying $S_{\sim,\approx}$ (Def. 4) is called *symmetric* w.r.t. $S_{\sim,\approx}$.

Thus, a voting rule $T$ is called *symmetric* w.r.t. $S_{\sim,\approx}$ if two $\sim$-related profiles yield $\approx$-related results. Please note that, according to our definition of symmetry, it is not a requirement for $\sim$ or $\approx$ to be symmetric relations.

*Example* 4 (Anonymity criterion [17])*.* A fundamental symmetry property, which most voting rules satisfy, is *anonymity*, where $\sim_{\text{anon}}$ is defined by

$$B \sim_{\text{anon}} B' \;\text{ iff }\; B' \text{ is a permutation of } B$$

and $\approx_{\text{anon}}$ is the equality relation on $\mathcal{W}$. The symmetry property $S_{\sim_{\text{anon}},\approx_{\text{anon}}}$ expresses that changing the order of the ballots – corresponding to changing which voter casts which vote – does not affect the election result. In this example, both $\sim_{\text{anon}}$ and $\approx_{\text{anon}}$ are symmetric.

It is often useful to consider families of symmetries that are parameterised by elements of some set $P$ (which may, for example, contain permutations):

**Definition 6** (Family of symmetry properties)**.** A *family of symmetry properties* w.r.t. a parameter set $P$ is a set $\{S_{\sim_p,\approx_p} \mid p \in P\}$ of symmetry properties that are induced by relational properties $\sim_p, \approx_p$ for each $p \in P$.

*Example* 5 (Neutrality criterion [17])*.* Again, we consider preferential voting rules (Def. 2). The neutrality criterion expresses that, if the candidates are permuted in a profile, then the voting rule permutes the candidates in the same way in the election result.

Neutrality can be formalised as a family of symmetry properties using the parameter set $P = \{\pi \mid \pi \text{ is a permutation on } \mathcal{C}\}$ and defining:

$$B \sim_\pi B' \;\text{ iff }\; B' = \pi(B) \qquad \text{and} \qquad c \approx_\pi c' \;\text{ iff }\; c' = \pi(c)\;\;,$$

(where the application of $\pi$ is extended to ballots and profiles in the obvious way). Thus,

$$S_{\sim_\pi,\approx_\pi} \;=\; \{((B,c),(B',c')) \mid B' = \pi(B) \text{ implies } c' = \pi(c)\}\;\;.$$

A voting rule satisfies the neutrality criterion iff it satisfies $S_{\sim_\pi,\approx_\pi}$ for all $\pi \in P$.

*Example* 6*.* The monotonicity property *Mono* from Example 2 also corresponds to a family of symmetry properties, using the parameter set $P = \mathcal{C}$ (the candidates are the parameters) and defining $\sim_{\uparrow p}, \approx_{\uparrow p}$ by:

$$\begin{aligned} B \sim_{\uparrow p} B' \quad &\text{iff} \quad B' \text{ results from } B \text{ by replacing a single ballot } b \in B \text{ by a ballot } b' \in b^{\uparrow p} \\ c \approx_{\uparrow p} c' \quad &\text{iff} \quad c = p \text{ implies } c' = p\;. \end{aligned}$$

A voting rule satisfies property *Mono* if and only if it satisfies $S_{\sim_{\uparrow p},\approx_{\uparrow p}}$ for all $p \in \mathcal{C}$, i.e., $\mathit{Mono} = \bigcap_{p\in\mathcal{C}} S_{\sim_{\uparrow p},\approx_{\uparrow p}}$.

Note that the relation $\sim_{\uparrow p}$ is not an equivalence relation for monotonicity.

This formalisation using a parametric property family is necessary for *Mono*, it cannot be described as a single symmetry property. In fact, every relational property can be phrased as a family of symmetry predicates for some parameter set.

## 3 Exploiting Symmetry Properties for Proofs of Functional Properties

In practice, the number of possible ballots is very large and the number of possible profiles even larger. Correspondingly, there is a huge number of possible execution paths through implementations of voting rules. That makes testing of voting rules difficult. Also, formal verification is infeasible for many combinations of voting rules and properties.

A possible solution is to exploit a symmetry property $S$ for proving (or testing) a voting rule $T$ w.r.t. a functional property $F$. The idea is to prove that $T$ satisfies $F$ for only a small subset $X \subseteq \mathcal{B}^*$ of the possible profiles and to then make use of the symmetry property $S$ to conclude that, if $(x, T(x)) \in F$ for all $x \in X$, i.e., $T$ has the property $F$ for inputs $x$, then the same holds for all $b \in \mathcal{B}^*$, i.e., $T$ has property $F$ in general. This, of course, is only useful if the subset $X$ is much smaller than $\mathcal{B}^*$ and if it is easy to prove that $T$ is symmetric w.r.t. $S$ – or if we can assume an existing proof because the symmetry is an interesting property in its own right (anonymity, neutrality, monotonicity etc.).

In addition to proving that (1) the function $T$ is symmetric w.r.t. $S$ and (2) $T$ has the property $F$ for inputs from $X$, the approach requires proving that (3) the set $X$ is spanning (Def. 7), i.e., all profiles can be reached from elements of $X$ with zero or more $\sim$-steps. And, finally, one has to prove that (4) the property $F$ is preserved under the symmetry $S$, i.e., $F$ is symmetry-resilient w.r.t. $S$ (Def. 8). It is important to note that (3) and (4) do not depend on the voting rule $T$ but only on the property $F$, the symmetry $S$, and the spanning set $X$. Thus, the proofs for (3) and (4) can be reused for other voting rules or when (the implementation of) a voting rule is changed.

**Definition 7** (Spanning set)**.** Let $\sim \subseteq \mathcal{B}^* \times \mathcal{B}^*$ be a binary relation on profiles. A set $X \subset \mathcal{B}^*$ of profiles is a *spanning set w.r.t.* $\sim$ iff, for every profile $B \in \mathcal{B}^*$, there is a profile $x \in X$ with $x \sim^* B$, where $\sim^*$ is the transitive, reflexive closure of $\sim$.

**Definition 8.** Let $F \in \mathcal{F}$ be a functional property, and let $S_{\sim,\approx} \subseteq \mathcal{R}$ be a symmetry property. Then, $F$ is called *symmetry-resilient w.r.t.* $S_{\sim,\approx}$ iff, for all $B, B' \in \mathcal{B}^*$ and $W, W' \in \mathcal{W}$,

$$(B, W) \in F,\, B \sim B',\, W \approx W' \qquad \text{implies} \qquad (B', W') \in F \ .$$

The following theorem formalises the approach of exploiting symmetry properties to help with proving functional properties.

**Theorem 1.** *Let $F \in \mathcal{F}$ be a functional property and $\{S_{\sim_p,\approx_p}\}$ a family of symmetry properties with parameter set $P$; further, let $X \subset \mathcal{B}^*$ be a set of profiles.*

*Then, every voting rule $T : \mathcal{B}^* \to \mathcal{W}$ such that*

$$T \text{ is symmetric (Def. 5) w.r.t. } S_{\sim_p,\approx_p} \text{ for all } p \in P \tag{1}$$

$$(x, T(x)) \in F \text{ for all } x \in X \tag{2}$$

$$X \text{ is spanning (Def. 7) w.r.t. the union } \textstyle\bigcup_{p \in P} \sim_p \text{ of the relations } \sim_p \tag{3}$$

$$F \text{ is symmetry-resilient (Def. 8) w.r.t. } S_{\sim_p,\approx_p} \text{ for all } p \in P \tag{4}$$

*satisfies the property $F$, i.e., $T \subseteq F$.*

*Proof.* Let $B \in \mathcal{B}^*$ be an arbitrary profile; we have to show that $(B, T(B)) \in F$. Because of (3), there exist $B_0 \sim_{p_0} B_1 \sim_{p_1} \cdots \sim_{p_{n-1}} B_n = B$ with $B_0 \in X$. It is, thus, sufficient to show that $(B_k, T(B_k)) \in F$ for all $k$, which we do by induction on $k$. The base case $(B_0, T(B_0)) \in F$ follows from (2) as $B_0 \in X$. For the step case, the induction hypothesis is $(B_k, T(B_k)) \in F$. As $B_k \sim_{p_k} B_{k+1}$, we have also $T(B_k) \approx_{p_k} T(B_{k+1})$ by (1). From that and the hypothesis, we can derive $(B_{k+1}, T(B_{k+1})) \in F$ using (4). $\square$

The following corollary is a version of Theorem 1 in which all definitions are expanded. We include it to show in one place the four proof obligations that arise for verifying that the voting rule $T$ satisfies the functional property $F$ using the symmetry property $S$.

**Corollary 1.** *Let $F \in \mathcal{F}$ be a functional property and $\{S_{\sim_p,\approx_p}\}$ a family of symmetry properties with parameter set $P$; further, let $X \subseteq \mathcal{B}^*$ be a set of profiles.*

*Then, every voting rule $T : \mathcal{B}^* \to \mathcal{W}$ such that*

$$\forall B, B' \in \mathcal{B}^*, p \in P : \quad B \sim_p B' \implies T(B) \approx_p T(B') \tag{1'}$$

$$\forall x \in X : \quad (x, T(x)) \in F \tag{2'}$$

$$\forall B \in \mathcal{B}^* : \exists x \in X : \quad x \sim^* B \tag{3'}$$

$$\forall (B, W), (B', W') \in \mathcal{E}, p \in P : \quad (B, W) \in F \wedge B \sim_p B' \wedge W \approx_p W' \Rightarrow (B', W') \in F \tag{4'}$$

*satisfies the property $F$, i.e., $T \subseteq F$.*

*In (3'), $\sim^*$ denotes the transitive, reflexive closure of $\bigcup_{p \in P} \sim_p$.*

## 4 Automated Verification of Voting Rules

In the following, we apply the insights gained in Sections 2 and 3 in the field of computer-aided automated verification of voting rules, where these rules are proved to be correct w.r.t. functional and relational properties. We argue that this is an important step towards a process for the design and development of verified tailor-made voting rules with clear and trustworthy axiomatic characterisations.

### 4.1 Verification of Functional Properties

Using program verification techniques, we analyse properties for concrete implementations of a voting rule $T$ in an imperative programming language (in our case, C or Java), operating on a concrete representation of ballots and voting results as a data structure in that language (e.g., arrays). In order to identify the semantics of a program that implements a voting rule $T$ correctly with the function $T$ itself, we assume that every such algorithm is (a) total (i.e., terminating for all inputs; to be shown in a separate verification) and (b) deterministic (guaranteed by the programming language semantics). This allows us to use the definitions of properties from Section 2 on both the abstract level of voting rules and the concrete level of imperative implementations.

We describe the target properties in first-order predicate logic, which comes with a clear declarative notation and can be directly used in automated deduction tools. However, instead of directly quantifying over both the profiles and election results, program verification makes use of symbolic execution with preconditions – specifying requirements to the input, i.e., the profile – and postconditions – specifying the resulting property of the election result, e.g., the winning candidate. In case of interactive verification techniques, also auxiliary specifications such as loop invariants are needed to guide the verification tool.

Symbolic execution does not require concrete values, but symbolically executes the program for any possible input value. In general, this leads to a high computational complexity, entailing a possible state space explosion, which is a major obstacle for an efficient use of formal program verification techniques. Therefore, in the following, we derive effective techniques from the insights in Section 3 to reduce the effects of state space explosion for verifying functional properties of voting rules.

## 4.2 Verification of Relational Properties

Relational program properties relate the evaluation of two independent inputs. For their verification, two runs of the same program $P$ need to be analysed and their results compared. To formalise this, we assume that there are two variants $P_1$ and $P_2$ of the program that are identical up to variable names. Program variant $P_1$ operates on variables $\bar{x}_1$ while $P_2$ operates on variables $\bar{x}_2$. The results are stored in the variables $r_1$ and $r_2$. A common technique (called *self-composition* [2, 12]) for proving a relational property for program $P$ is to show a functional property for the composed program $P_1 \,; P_2$, combining the behaviour of both variants. To prove that program $P$ satisfies relational property $R$, we prove that, if $P_1 \,; P_2$ is run with inputs $\bar{x}_1, \bar{x}_2$ and gives the results $r_1, r_2$, then $((\bar{x}_1, r_1), (\bar{x}_2, r_2)) \in R$. In other words, we require that $\{\text{true}\}P_1 \,; P_2\{R((\bar{x}_1, r_1), (\bar{x}_2, r_2))\}$ is a valid Hoare triple [21].

Formal verification of relational properties using self-composition is challenging in general since it requires static analysis of two independent program runs; the exploration space that needs to be analysed is potentially exponentially larger than that for analysing a single program run. Moreover, for this type of relational verification, sufficiently strong program specifications (in particular, loop invariants and postconditions) are required to prove non-trivial relational properties (such as, e.g., interesting symmetries).

Relational verification is much easier to handle by verification tools if the two program runs to be compared are not simply executed consecutively but have their statements weaved into a common program in a more flexible fashion. Since $P_1$ and $P_2$ have disjoint variable sets, reordering statements cannot have an effect on the result state as long as the execution order amongst the statements of only $P_1$ resp. $P_2$ is preserved. Details about the possibilities of flexibly weaving programs can be found in [2, 16].

Consider for instance the program `while(`*cond*`) {` *body* `}` consisting of a single while-loop. It is easy to see that, instead of concatenating two variants of this code (one with $cond_1/body_1$ and one with $cond_2/body_2$), one can use the single-loop program

$$\texttt{while(}cond_1 \texttt{ || } cond_2\texttt{) \{ if(}cond_1\texttt{)\{}body_1\texttt{\} if(}cond_2\texttt{)\{}body_2\texttt{\} \}}$$

This weaved program does not require separate loop invariants for the loops in $P_1$ and $P_2$ but only a single so-called *coupling invariant* for the weaved loop that sets variables $\bar{x}_1$ and $\bar{x}_2$ into relation. In many cases, the coupling invariant is significantly simpler than the loop invariants. As long as the two loop executions behave similarly, it is easier to express how the two states are related after each step than to specify what it is that the loops actually compute. Thus, relational verification using weaved programs is ideal for proving symmetry properties, as symmetric inputs typically induce similar runs of the program.

*Example* 7. To prove that plurality voting is monotonic, i.e., that it satisfies the relational property *Mono* from Example 2, we weave two variants of the program (see Listing 1) such that the corresponding loops in the variants are combined. The first loop of the program counts the ballots while the second loop computes the winner. The essential part of the coupling invariant that can be used to prove monotonicity reads

$$(\texttt{i}_1 \leq \texttt{c} \rightarrow \texttt{max}_1 \geq \texttt{max}_2) \wedge (\texttt{elect}_1 = \texttt{c} \rightarrow \texttt{elect}_2 = \texttt{c} \wedge \texttt{max}_2 \geq \texttt{max}_1)$$

In comparison, the essence of a sufficient full functional loop invariant for a single (non-weaved) loop is considerably larger and reads

$$\begin{gathered}(\forall t : 1 \leq t < \texttt{i} \rightarrow \texttt{max} \geq \texttt{res}[t]) \wedge \\ \big(\texttt{elect} \neq 0 \rightarrow \texttt{max} = \texttt{res}[\texttt{elect}] \wedge (\forall t : 1 \leq t < \texttt{i} \wedge t \neq elect \rightarrow \texttt{max} > \texttt{res}[t])\big) \wedge \\ (\texttt{elect} = 0 \wedge \texttt{max} > 0 \rightarrow (\exists u, w : 1 \leq u < w < \texttt{i} \wedge \texttt{res}[u] = \texttt{res}[w] = \texttt{max}))\ .\end{gathered}$$

While the details of this invariant do not matter at this point, it is plausible that finding the required specification and conducting the verification is simpler using the weaving approach.

### 4.3 Verification Using Symmetry Properties

Once a voting rule has been proved to satisfy a (relational) symmetry property, this fact can be exploited to reduce the effort for proving other properties. Theorem 1 says that, if voting rule $T$ satisfies a symmetry property $S$, then it suffices to examine a spanning set $X$ of $S$ for the analysis of $T$ w.r.t. a functional property $F$ (resilient to $S$).

Corollary 1 leaves us with four proof obligations of which only two (1' and 2') are program verification tasks that involve (the implementation of) $T$. We assume that the voting rule has already been proved to satisfy the symmetry $S$ (1'). In (4'), it must be shown that $F$ is compatible with $S$. Many sensible properties are resilient to symmetries; every property that only refers to numbers of votes is indifferent to the actual ordering of ballots, i.e., it is resilient to anonymity (cf. Example 4). Unless the target property makes use of the actual order of the ballots in $B$, resilience w.r.t. anonymity can be proven either manually or by using a theorem prover which is able to reason about ordered sets and first-order logic.

In the specification used for verification, a spanning set $X$ for $S$ is represented in the form of a first-order logic predicate $\varphi$, i.e., a formula with a free variable $x \in \mathcal{B}$. Then, the set $X_\varphi \subseteq \mathcal{B}$ is the set of ballots which satisfy $\varphi$. If $X_\varphi$ is a spanning set, then $\varphi$ is called a *symmetry-breaking predicate* (SBP), a term originating from the field of constraint satisfaction [10]. When a candidate $\varphi$ for an SBP is given, proof obligation (3') requires showing that it is indeed a spanning set. The concrete form of (3') results from expanding the definitions of $\varphi, \sim$ and $\approx$. It does not depend on the voting rule and can be verified either via a manual proof, or using an automated theorem prover that can deal with first-order logic and set theory (including transitive closure).

Note that we use a generalised concept of symmetries or symmetry operations, which does not require an actual equivalence relation. Hence, the requirements for an SBP are slightly different from those usually required for symmetry breaking as defined in [10]. Generally, the task of finding an SBP is far from trivial and an automatic generation seems intractable.

Finally, the functional program verification (2') can be done using program verification and the SBP can be integrated into the precondition using its first-order logic form.

## 5 Case Study

In this case study, we exemplarily apply the techniques described in Section 4 to verify that a simple *first-past-the-post* (FPTP) plurality voting rule satisfies the functional *majority property* (Example 1) and the relational *anonymity property* (Example 4).

FPTP plurality voting is already thoroughly analysed and axiomatised (cf. [28]). But, even though this is a simple and well-understood example, the insights and results of this case extend to more complex voting rules and properties as well.

The function `voting` in Listing 1 implements plurality voting, the constants `C` and `N` encode the numbers of candidates resp. ballots and the array `ballots` encodes the profile. Candidates are represented by natural numbers $1, \ldots, \mathtt{C}$; the result 0 indicates a tie.

### 5.1 Verification Tools and Techniques

We consider two approaches for source-code-level program verification: software bounded model checking (SBMC) and deductive theorem-proving.

SBMC statically analyses programs up to a predefined number of loop iterations and recursions. Programs are fully symbolically executed and checked for errors up to the given bound. Beyond the bound, no correctness guarantee can be obtained, but we rely on the small-scope hypothesis [22] claiming that typical faults manifest themselves already in small instances. We use the highly automated software model checker CBMC [9] which takes

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

Listing 1: Implementation of plurality voting

C/C++ programs as input, annotated with specifications in the form of assertions and assumptions. Since quantifiers are not supported by CBMC, quantified expressions need to be expressed as assertions within a loop. CBMC internally models all data structures as bit vectors. The symbolically executed programs are translated to statements over bit vectors, which are then processed by a specialised satisfiability solver for bit vectors. For our experiments, we use CBMC 5.3 with the built-in solver based on MiniSat 2.2.0 [15], combined with an efficient bit-vector refinement procedure [6]. All experiments are performed on an Intel(R) Core(TM) i5-3360M CPU at 2.80 GHz with 4 cores and 16 GB of RAM with a hard time-out of 30 minutes for each experiment.

In program verification based on deductive theorem proving, loops are not analysed up to a bound, but verified for *any* number of iterations using inductive loop invariants. Without upper bounds, the guarantees of deductive verification are stronger – yet, there is a price to pay: loop invariants need to be found and specified, which can be a labour-intensive task. For deductive verification, we use the semi-interactive theorem-prover KeY [1], which operates on Java programs. Specifications for KeY are written in the Java Modeling Language (JML) [23]. In JML, methods are annotated with pre- and postconditions. The successful verification then proves that the method, started in a state satisfying the precondition, terminates in a state satisfying the postcondition. Just as for CBMC, properties need to be encoded into the specification language. Since JML supports a variety of quantifiers (e.g., sum operators), less creative energy for the encoding is required. Program analysis with KeY considers unbounded program iterations and unbounded datastructures (unlike CBMC where finite bounds are given on both the number of iterations and on datastructures). Therefore, conducting proofs in KeY is less automatic and requires additional specifications like loop invariants and further user interaction.

## 5.2 Functional Verification Using CBMC

For plurality voting, the majority criterion can be put in simpler terms than in Example 1 as voters can only select a single choice. Thus, a profile $B \in \mathcal{B}^*$ is a sequence of candidates (one for each voter).

**Corollary 2** (Majority criterion for single-choice voting rules)**.** *A single-choice voting rule $T : \mathcal{C}^* \to \mathcal{C} \cup \{\bot\}$ satisfies property Maj (Example 1) iff, for all $N \geq 1$,*

$$\forall (b_1, \ldots, b_N) \in \mathcal{C}^* \, \forall c \in \mathcal{C} : (|\{b_i \mid b_i = c, i \in \{1, \ldots, N\}\}| > \tfrac{N}{2} \to T(b_1, \ldots, b_N) = c) \ .$$

In this case study, we use the majority criterion as formalised in Corollary 2 and the plurality voting rule as implemented in Listing 1.

```
void anonymity(int ballots1[N], int ballots2[N], int c, int d, int v, int w) {
    assume (0 < c ≤ C ∧ 0 < d ≤ C);
    assume (0 < v ≤ N ∧ 0 < w ≤ N ∧ v < w);
    for (int i = 1; i < N; i++) {
        assume (0 < ballots1[i] ≤ C ∧ 0 < ballots2[i] ≤ C);
        if (i != v && i != w) { assume (ballots1[i] == ballots2[i]); }
    }
    assume (ballots1[v] == ballots2[w]);
    assume (ballots2[v] == ballots1[w]);
    unsigned int elect1 = voting(ballots1);
    unsigned int elect2 = voting(ballots2);
    assert (elect1 == elect2);
}
```

Listing 2: Anonymity property as a Java/C program

Using CBMC in a straightforward manner, verifying that plurality voting has the majority property is possible for small bounds on the number of voters and candidates, but becomes infeasible for higher numbers. The time-out of 30 minutes is reached with 5 candidates and 45 voters resp. with 10 candidates and 20 ballots. Considering the small-scope hypothesis and the simple structure of plurality voting, these bounds are high enough. The run-times (in seconds) for up to 15 candidates and 15 voters are shown in Fig. 2a.

Preliminary experiments for further voting rules have shown that reachable maximum bounds prove to be even lower for more complex voting rules. This is particularly unfortunate, since our experiments provide also evidence that these complex voting rules require an analysis up to higher bounds in order to produce reliable evidence.

## 5.3 Relational Verification Using CBMC

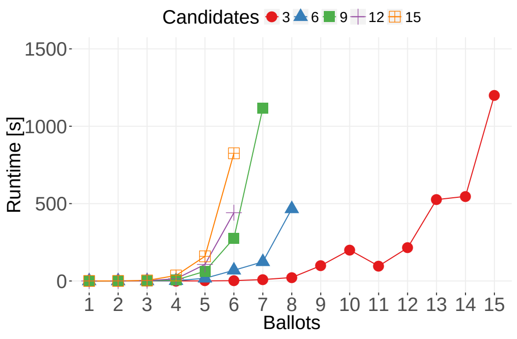
As explained in Section 4.2, relational verification with weaved programs and coupling invariants is more efficient than just composing two variants of the program. We evaluate the impact of coupling invariants on performance and feasibility using as an example the verification of plurality voting w.r.t. the anonymity property (Example 4).

**Corollary 3** (Anonymity property for single-choice voting rules). *A single-choice voting rule $T$ satisfies the anonymity property (Example 4) iff, for all $N \geq 1$,*

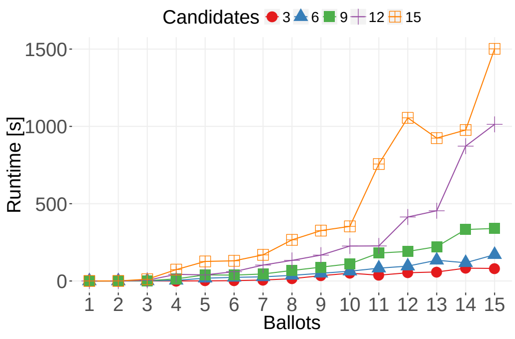
$$\forall (b_1, \ldots, b_N), (b'_1, \ldots, b'_N) \in \mathcal{C}^* \; \forall v, w \in \{1, \ldots, N\} : \\ (b_v = b'_w \land b_w = b'_v \land (\forall i \in \{1, \ldots, N\} : v \neq i \land w \neq i \rightarrow b_i = b'_i)) \rightarrow (T(b) = T(b')) \; .$$

The corresponding CBMC specification of anonymity is shown in Listing 2. Note that the formula in Corollary 3 has four outer universal quantifications (the two profiles that are compared and the indices $v, w$ of the ballots that are exchanged) as well as an inner universal quantification of variable $i$ (a further ballot index). For specifying the property in CBMC, where only C/C++ expressions are allowed, we make use of array variables `ballots1`, `ballots2` and integer variables `v`, `w`. We restrict their ranges with the CBMC `assume`-statement. As nothing further is assumed about their initial values, they are effectively universally quantified (without loss of generality we also add `v` < `w` as assumption). The quantification of variable `i` is expressed using a `for`-loop containing `assume`-statements.

The implementation of plurality voting has two loops, one counting the ballots and one finding the maximum of votes per candidate and, thus, determining the winner. The coupling invariant for the counting loop is:



(a) Without coupling invariants



(b) Weaved program with coupling invariants

Figure 1: Verification of the anonymity property for plurality voting

```
0 < i1 ≤ N+1 ∧ i1 == i2 ∧ (∀ int t: 0 < t ≤ C → res2[t] == res1[t] +
  (v < i1 ≤ w ? (t == ballots1[v] ? -1 : 0) + (t == ballots1[w] ? 1 : 0) : 0))
```

The two loop counters are `i1`,`i2`. This invariant captures several cases: If we did not see any switched ballots yet or we already counted both switched ballots, the count is the same in both variants (`res2[t] == res1[t]`). Otherwise, there are three sub-cases depending on the variable `t`: the count in `res2[t]` is either `res2[t]`, `res2[t]+1`, or `res2[t]-1`.

The coupling invariant for the loop finding the winner is:

```
0 ≤ i1 ≤ C+1   ∧   0 ≤ elect1 < i1   ∧ i1 == i2   ∧   elect2 == elect1   ∧
max2 == max1   ∧   (∀ int t: 0 < t ≤ C → res2[t] == res1[t])
```

It is simpler because at the point where the second loop is started, the count is already the same in both variants. Note that we allow for a tie, in which case `elect1` and `elect2` both have the value 0.

Using CBMC to verify the anonymity property for plurality voting using (a) simple composition of two variants without coupling invariant and (b) weaved programs with coupling invariants, we get the run-times (in seconds) shown in Fig. 1a resp. Fig. 1b (for candidate and ballot numbers between 1 and 15). For the missing data points, the run-times exceed our predefined time-out of 30 minutes.

As can be seen, the verification without weaving and coupling invariants becomes infeasible for rather small bounds. Verification with coupling invariants fares considerably better; the time-out, here, is finally reached for about 10 candidates and 25 ballots, which is similar to the bounds reachable for functional properties (see Section 5.2).

## 5.4 Relational Verification Using KeY

If the bounds that can be reached using bounded model checking are insufficient, or the application requires an unbounded proof, deductive verification is a viable alternative.

Proof search for deductive verification is not fully automatic. Nevertheless, using the deductive verification tool KeY, verifying the anonymity property for plurality voting is possible for unbounded numbers of ballots and candidates with an almost automatic verification, requiring very little user interaction. Here as well, coupling invariants are very useful – we were not able to construct a KeY verification proof without them.

To further investigate the applicability of this approach, we performed a variety of unbounded proofs with the KeY system. We verified various single-winner voting rules for various relational properties (see [25, 31] for definitions of the voting rules and properties).

Table 1: Verified properties and voting rules using KeY (required lines of specification).

| | Plurality voting | Approval voting | Range voting | Borda count |
|---|---|---|---|---|
| Anonymity | 33 | 43 | 44 | 44 |
| Neutrality | 42 | 56 | 57 | 57 |
| Monotonicity | 46 | 47 | 48 | 52 |
| Participation | 28 | 50 | 51 | 50 |
| Homogeneity | 53 | 70 | 71 | 71 |

Note that approval and range voting are not preferential voting rules according to Definition 2. We applied the relational analysis to these weighted voting rules with the relational properties adapted accordingly. Table 1 gives an overview of these experiments. The numbers shown are lines of specification, which includes both the formalisation of the property to be verified and the coupling loop invariant. This indicates that not too much effort is required for writing the specification.

Note that for Borda Count, we show *monotonicity ceteris paribus*, i.e., the preference orders are only changed w.r.t. the raised candidate. For the remaining voting rules, we use the criterion *mono-raise*, where the preference orders of the other candidates may also be changed. The definition of the properties in Table 1 is according to Woodall [31].

In all cases, proof construction required less than ten user interactions, most of them quantifier instantiations. The results are promising, as further automation seems possible.

## 5.5 A Symmetry-Breaking Predicate for Anonymity

For the anonymity property, which is a symmetry (according to Definition 5), we formulate a simple symmetry-breaking predicate by defining the spanning set to consist of those profiles where the ballots are ordered by the chosen candidate. This is, for any ballot $b_i$ in the profile, which is not the first ballot, the index of the candidate in the directly preceding ballot $b_{i-1}$ is less or equal to the candidate in $b_i$. Written in first-order logic, this reads as follows:

**Definition 9** (Symmetry-breaking predicate for the anonymity property)**.** For profiles $B = (b_1, \ldots, b_N) \in \mathcal{B}^*$, where $\mathcal{B} = \mathcal{C}$ (the candidates are the possible ballots), the symmetry-breaking predicate $\varphi_{\text{anon}}(B)$ is defined by:

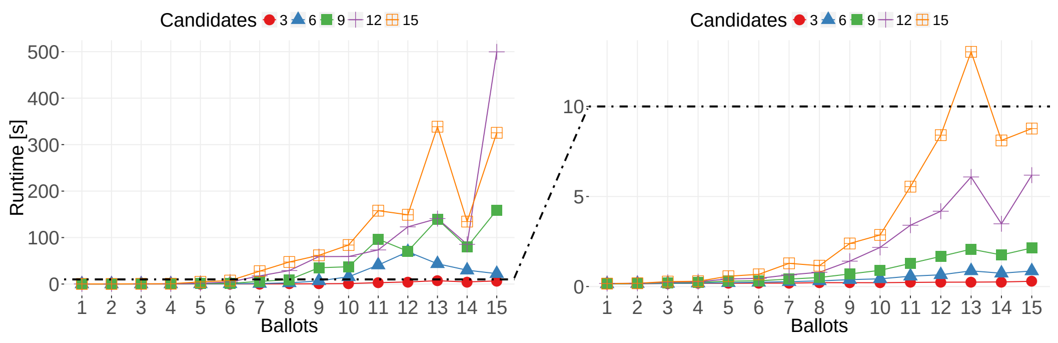
$$\forall i \in \{2, \ldots, N\} : b_{i-1} \leq b_i$$

To prove that $\varphi_{\text{anon}}$ is indeed a symmetry-breaking predicate, we have to show that for any profile $B \in \mathcal{B}^*$, there is a profile $x$ which satisfies Definition 9 and can be transformed into $B$ through the application of zero or more symmetry operations. The symmetry operation $\sim$ for the anonymity property is permutation on profiles. Hence, we need to show that any profile is reachable via zero or more permutations from a profile satisfying $\varphi_{\text{anon}}$. This holds since any profile can be transformed into a sorted profile via a single permutation.

## 5.6 Proving the Majority Criterion Using the SBP for Anonymity

As explained in Section 5.6, if CBMC is applied to verify that plurality voting has the majority property without the help of a symmetry reduction, then bounds of about 20 ballots are feasible. The run-times (in seconds) for this approach are shown in Fig. 2a.

Using the symmetry-breaking predicate from Definition 9, the situation improves dramatically. The much lower run-times are shown in Fig. 2b (please note the different time



(a) Without symmetry reduction (b) Using symmetry reduction

Figure 2: Verification of the majority criterion for plurality voting

scales in the two diagrams). Experiments for bounds beyond 15 ballots show that handling 100 ballots for 10 candidates becomes feasible.

These findings are supported by an analysis for this particular symmetry: the number of sorted profiles, which are those that need to be considered using symmetry reduction, is only $\frac{(|\mathcal{C}|+N-1)!}{N! \cdot (|\mathcal{C}|-1)!}$ compared to the $|\mathcal{C}|^N$ profiles in the original set $\mathcal{B}^N$ (for $\mathcal{C}$ the number of candidates and preference profiles of size $N$).

It remains to be shown that the majority criterion is symmetry-resilient with respect to the anonymity property. This does not require program verification, and the proof is indeed trivial. Changing the order of the ballots does not affect the number of votes for any candidate. Hence, if there is a majority of ballots voting for a candidate $c$, i.e., $c$ is the majority winner, then any permutation on the profile preserves $c$'s majority. In the remaining case, where there is no majority winner as no candidate gets a majority of votes, permuting the ballots does not produce any new majority.

## 6 Related Work

While we use program verification technology based on first-order logic for the automated verification of voting rules, there are other approaches using tactical theorem provers and higher-order logic. Examples are proofs carried out by Dawson [13], Goré and Meumann [20], and Schürmann and Pattinson [26]. Verification using tactical theorem provers may lead to even higher confidence levels, but the task is inherently difficult and time-consuming, resulting in huge and laborious interactive proofs.

There exists extensive research analysing theoretical voting rules on a more intuitive or experimental level involving empirical experiments or comparisons of previous elections [3, 18, 27]. Also, there is research on the verification of concrete voting systems, i.e., considering a concrete voting software [14].

Furthermore, a multitude of theoretical work on proving and finding new incompatibilities of voting rule properties has been done using SAT solvers [4, 5, 8, 19, 30]. Another application of computer-aided techniques to social-choice theory is the use of machine learning for designing new social-choice mechanisms satisfying desired axiomatic properties [32].

We use weaved programs to reduce the required effort for relational verification to that of "standard" program verification. A general notion of product programs that supports such a reduction is provided by Barthe et al. [2].

There is also related work on breaking symmetries on the problem-specification level [7, 24] and on methods for automatically generating symmetry-breaking predicates for classes of combinatorial objects for search problems [29]. More applied work on this task provides tools which detect symmetries in structured graphs generated from CNF formulas [11].

## 7 Conclusion and Future Work

In this paper, we have formally defined the concepts of functional and relational properties as well as the special case of symmetry properties. We have shown that software bounded model checking and deductive verification can be effectively used for the verification of functional and relational properties, and that symmetry properties can be used to greatly reduce the effort needed for verification of other properties.

Our case study shows that bounded verification up to bounds of about 20–30 ballots is possible in practice, which can be increased to about 100 ballots using symmetry reduction techniques. With the small-scope hypothesis, these bounds are sufficiently high even if the structure of profiles and election results and the operations that make up the voting rule implementation are more complex than in the simple case of plurality voting. In addition, modularisation and decomposition techniques can be used to handle even more complex rules by verifying their components individually (e.g., phases or rounds in the counting process).

Future work includes a further investigation of the taxonomy of voting rule properties and an extension to existential properties and mixtures of existential and universal properties. Further, we plan to apply our approach in bigger case studies to further properties and, in particular, to more complex voting rules and implementations. We are confident that interesting properties can be verified for voting rules as they are used in real-world elections.

## References

[1] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, et al. The KeY platform for verification and analysis of Java programs. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 8471 of *LNCS*. Springer, 2014.

[2] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods*, volume 6664 of *LNCS*. Springer, 2011.

[3] S. J. Brams and P. C. Fishburn. Does approval voting elect the lowest common denominator? *PS: Political Science & Politics*, 21(02), 1988.

[4] F. Brandt and C. Geist. Finding strategyproof social choice functions via SAT solving. *J. Artif. Intell. Res. (JAIR)*, 55, 2016.

[5] F. Brandt, C. Geist, and D. Peters. Optimal bounds for the no-show paradox via SAT solving. *CoRR*, abs/1602.08063, 2016.

[6] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*. Springer, 2007.

[7] M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. In *AI* IA 2005: Advances in Artificial Intelligence*. Springer, 2005.

[8] S. Chatterjee and A. Sen. Automated reasoning in social choice theory: Some remarks. *Mathematics in Computer Science*, 8(1), 2014.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*. Springer, 2004.

[10] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Morgan Kaufmann, 1996.

[11] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04. ACM, 2004.

[12] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[13] J. E. Dawson, R. Goré, and T. Meumann. Machine-checked reasoning about complex voting schemes using higher-order logic. In *E-Voting and Identity, VoteID 2015*, volume 9269 of *LNCS*. Springer, 2015.

[14] G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. In *VSTTE*, volume 5295 of *LNCS*. Springer, 2008.

[15] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*. Springer, 2003.

[16] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, ASE '14. ACM, 2014.

[17] P. C. Fishburn. *The Theory of Social Choice*. Princeton University Press, 1973.

[18] M. Gallagher. Monotonicity and non-monotonicity at PR-STV elections, 2013.

[19] C. Geist and U. Endriss. Automated search for impossibility theorems in social choice theory: Ranking sets of objects. *CoRR*, abs/1401.3866, 2014.

[20] R. Goré and T. Meumann. Proving the monotonicity criterion for a plurality vote-counting program as a step towards verified vote-counting. In *Electronic Voting: Verifying the Vote (EVOTE)*. IEEE, 2014.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), Oct. 1969.

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

[23] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*. Springer, 1999.

[24] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Abstraction, Reformulation and Approximation*. Springer, 2005.

[25] H. Moulin. Condorcet's principle implies the no-show paradox. *Journal of Economic Theory*, 45(1), 1988.

[26] D. Pattinson and C. Schürmann. Vote counting as mathematical proof. In *AI 2015: Advances in Artificial Intelligence - 28th Australasian Joint Conference*, volume 9457 of *LNCS*. Springer, 2015.

[27] M. Regenwetter and B. Grofman. Approval voting, Borda winners, and Condorcet winners: Evidence from seven elections. *Management Science*, 44(4), 1998.

[28] F. S. Roberts. Characterizations of the plurality function. *Mathematical Social Sciences*, 21(2), 1991.

[29] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12), 2007.

[30] P. Tang and F. Lin. Computer-aided proofs of Arrow's and other impossibility theorems. *Artif. Intell.*, 173(11), 2009.

[31] D. R. Woodall. Properties of preferential election rules. *Voting Matters*, 3, 1994.

[32] L. Xia. Designing social choice mechanisms using machine learning. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13*. IFAAMAS, 2013.

[33] H. P. Young. An axiomatization of Borda's rule. *Journal of Economic Theory*, 9(1), 1974.

Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Mattias Ulbrich, Till Neuber
Institute for Theoretical Informatics
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Emails: {`beckert,bormer,kirsten,ulbrich`}`@kit.edu`, `till.neuber@student.kit.edu`