CHAPTER 14

# An Introduction to Voting Rule Verification

Bernhard Beckert, Thorsten Bormer, Rajeev Goré,
Michael Kirsten, and Carsten Schürmann

## 14.1 Introduction

Social choice functions (or more specifically voting rules) form the backbone of modern democracies. They are often expressed as algorithms palatable for machine implementation and execution, and simultaneously they control the transfer of power from the people to a government. Social choice functions vary in complexity and exhibit different behaviors and properties. For scientists, they are perfect objects of study, in particular, to predict the way they behave, to understand corner cases, or to check that they comply with the law. Conversely, for agents (or voters) the functions' behavior and which properties they satisfy or violate is often hard to discern and difficult to understand.

Program verification technology, which is based on formal logic and deduction, provides a powerful toolset for the analysis of algorithms and their properties. The reach and power of software verification methods and tools has increased tremendously over the last decade. Following their earlier successes, for example for hardware design, formal verification methods today are routinely applied to catch design errors at early stages of software and protocol development processes. There has been considerable progress in the verification of real-world software written in languages such as C and Java, as the core technologies of deductive program analysis have matured (Ahrendt et al., 2014; Clarke et al., 2004; Falke et al., 2013).

But as of yet, to our knowledge, little work has been done to combine formal program verification with social choice theory, even though the two go well together: Voting rules are mathematical constructs and formal program verification techniques are optimised to analyse those. Thus, these techniques can help to uncover the hidden secrets of voting rules. For example, they allow us to check whether a voting rule matches its specification, and help determine how many votes must be changed to cause a change in the outcome of an election.

In this chapter, we give an overview about the role that formal program verification can play in social choice research. In particular, we focus on one such technique, namely software bounded model checking (SBMC), which statically

analyses the implementation of a voting rule by exhaustively looking for counterexamples in a finitely bounded search space. In order to use a software bounded model checker, the voting rule implementation together with the property to be shown are translated into a formula in propositional or first-order logic before processing. The checker then tries to construct either a proof or a counterexample, which can subsequently be used to understand why the voting rule violates the property.

SBMC techniques can be applied to prove that an implementation of a voting rule satisfies a property for all inputs of bounded size (e.g., up to a certain limit on the number of voters and alternatives) by exhaustively checking the space of all possible inputs. SBMC tools are fully automatic, fast, and easy to use, but their drawback is that they will not be able to identify counterexamples that lie outside these bounds. There are other more complete approaches to program verification that ensure that a property holds for all inputs, but these techniques require significantly more effort and are not covered in this chapter (see Section 14.5 for references to related work).

The role of program verification in social choice research is based on the power of the axiomatic approach, which in modern social science was largely initiated by Arrow's impossibility theorem (Arrow, 1963). His findings imply that the perfect voting rule does not exist. Therefore, developing a voting rule that satisfies a given set of axiomatic properties is cumbersome as the trade-off between any such properties is inherently difficult and error-prone.

The experiences presented in this chapter document that errors in voting rules are easy to make and formal program verification methods greatly enhance the chances of finding such errors. Furthermore, we document that there are formal program verification techniques which provide proof that a voting rule—and its algorithmic implementation—meets a given property. Moreover, we show that these techniques can also produce quantitative statements such as, e.g., changing how many votes can change the election result.

Concretely, we conduct three case studies, namely, (1) checking the properties of the voting rule known as single transferable vote (STV), designed in particular for the election of members of the board of the Conference on Automated Deduction (CADE), (2) using the power of SBMC techniques for verifying properties of voting rules (demonstrated on simple examples), and (3) computing election margins for the purpose of auditing election results.

The rest of this chapter is structured as follows: We start by giving insights into the logical models and formalisations used and also present formalisation techniques specially tailored to axiomatic properties of voting rules (Section 14.2). Then, we describe and compare tools and techniques used for performing formal program verification of voting rules (Section 14.3). In the main part of this chapter, we present experiences and case studies showing the reach and power of formal program verification (Section 14.4). We conclude with a summary of this chapter and a brief overview of related work (Section 14.5).

## 14.2 Logic-based Formalisation of Properties

In this section, after giving some basic definitions, we classify different types of properties for voting rules. These concepts provide the basis for logic-based formalisations.

### 14.2.1 Basic Definitions: Ballots, Profiles, Voting Rules

We consider voting rules where the individual preferences of voters are aggregated to produce an election result. Throughout this chapter, to simplify the presentation, we only consider scenarios where each voter casts exactly one ballot. Each ballot holds its voter's preferences in the form of an order over all (or a subset of all) alternatives (or candidates).

**Definition 14.1.** *Let $N = \{1, \ldots, n\}$ be a finite set of voters, let $A$ be a finite set of $m$ alternatives, and let $W$ be a set of possible election results. Then, a* ballot *is an order $\succ_i$ on a (not necessarily strict) subset of $A$; and a* profile *$\langle \succ_1, \ldots, \succ_i \rangle$ (with $i \leqslant n$) is a sequence containing one ballot for each voter. The set of all possible ballots is denoted with $\mathcal{B}$, and the set of all possible profiles is denoted with $\mathcal{B}^*$.*

*A* voting rule *is a total function $f \colon \mathcal{B}^* \to W$, assigning an election result to each profile. An individual pair $(p, w) \in (\mathcal{B}^* \times W)$ consisting of a profile and an election result is called an* evaluation. *The set of all evaluations is $\mathcal{E} = \mathcal{B}^* \times W$.*

The concrete structure of possible election results in $W$ depends on the voting rule that is being investigated. In the following, we assume that election results $\langle a_1, \ldots, a_s \rangle \in W$ are sequences of alternatives, denoting that alternatives $a_1, \ldots, a_s$ have been elected (the order of the elected alternatives may or may not be significant). The empty sequence $\langle\rangle$ can be used to denote that there is no "valid" result (e.g., in case of a tie). In case the result is a singleton, we write $a$ instead of $\langle a \rangle$.

### 14.2.2 Functional and Relational Properties

We distinguish between *functional* and *relational* properties. Functional properties, such as the majority criterion, refer to single election results while relational properties, such as anonymity, compare two (or more) results. In the literature, functional and relational properties are also called *intra-* and *inter-profile* properties, as defined by Fishburn (1973).

**Definition 14.2 (Functional property).** *Given a set $\mathcal{B}$ of possible ballots and a set $W$ of possible results, a* functional property *$F$ for voting rules is a set of evaluations, i.e., $F \subseteq \mathcal{E} = (\mathcal{B}^* \times W)$ is a relation between profiles and results.*

*A voting rule $f : \mathcal{B}^* \to W$* satisfies a functional property *$F$ iff $f \subseteq F$, i.e., all evaluations of $f$ 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 14.1 (Majority criterion, majority winner).** *Given a profile $p \in \mathcal{B}^*$, a majority winner for $p = \langle \succ_1, \ldots, \succ_n \rangle$ is an alternative $a \in A$ that is preferred over all other alternatives in more than half of the ballots:*

$$|\{\succ_i \in p \mid a \succ_i a' \text{ for all } a' \in A, a' \neq a\}| > \tfrac{n}{2} \ .$$

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

$$Maj \;=\; \{(p,a) \mid \textit{if } a' \textit{ is a majority winner for } p \textit{ then } a' = a\} \;.$$

**Definition 14.3 (Relational property).** *Given a set $\mathcal{B}$ of possible ballots and a set $W$ of of possible results, a* relational property $R$ *for voting rules is a set of pairs of evaluations, i.e.,*

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

*A voting rule $f\colon \mathcal{B}^* \to W$* satisfies a relational property $R$ *iff, for all evaluations $e \in f$ and $e' \in f$, 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 "coexist" in a voting rule.

**Example 14.2 (Monotonicity criterion).** *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 $c \in A$ is given a higher rank.*

*The relational property of monotonicity is*

$$Mono \;=\; (\mathcal{E} \times \mathcal{E}) \setminus \{((p,w),(p',w')) \mid \textit{there is an alternative with } a \in w,\ a \notin w',\ \textit{and } p' \textit{ results from } p \textit{ by replacing a single ballot } b \in p \textit{ by a ballot } b' \in b^{\uparrow a}\}$$

*That is, $Mono$ contains* all *pairs of evaluations* except *those where a winning alternative is given higher preference in one of the ballots (denoted by $b'$) which results in the alternative $a$ no longer being elected.*

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, a relational property is a relation between two evaluations of the voting rule. Satisfaction is hence judged by considering each of its evaluations in the context of the 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.

The classes of functional and relational properties do not cover all interesting properties of voting rules, but only those that can be checked by looking at one (functional) or two (relational) evaluations at a time. However, there are properties that require a comparison of three or more evaluations.

**Example 14.3 (Consistency criterion).** *A voting rule satisfies the consistency criterion if, for any three profiles $p, p_1, p_2$ such that $p$ is the concatenation of $p_1$ and $p_2$: if $f(p_1) = f(p_2)$ then $f(p) = f(p_1) = f(p_2)$.*

Properties such as consistency[1], which can (only) be defined by comparing three evaluations are called 3-properties. This concept can be extended to generalised $k$-properties for $k \in \mathbb{N}$, which does—however—still not cover all properties.

[1] Sometimes also referred to as *reinforcement* or *convexity*.

For example, the surjectivity[2] property, which requires that for each possible election result there is a profile leading to that result, is a rather simple property that is not a $k$-property for any $k$. Surjectivity is an *existential* property, requiring the existence of (combinations of) certain evaluations, while $k$-properties are universal in nature, requiring all $k$-tuples of evaluations are "good" in some sense.

### 14.2.3 Formalisation in First-order Logic with Theories

To formalise properties of voting rules, we use first-order logic over the theories of natural numbers and arrays. Using these theories, on the one hand, allows to easily represent profiles and election results and, on the other hand, is supported by most SMT solvers and program verification tools (Section 14.3.1).

Arrays and numbers allow to encode profiles and election results as follows:

- A profile $p = \langle \succ_1, \ldots, \succ_n \rangle$ is represented as a two-dimensional array $P$, where $P[i,j] \in \{1, \ldots, m\}$ is the alternative that is ranked by ballot $\succ_i$ in the $j$th place, i.e., $P[i,1] \succ_i P[i,2] \succ_i \cdots \succ_i P[i,m]$.
- An election result $w = \langle a_1, \ldots, a_m \rangle$ is represented as a one-dimensional array $W$ of size $s$, where $W[i] = a_i$. If less than $s$ alternatives are elected, then $W[e] = 0$ for every empty place $e \leqslant s$.

A functional property (Definition 14.2), which is a set of evaluations $(p, w)$, can be characterised by a formula $\phi(P, W)$ that has exactly two free variables $P, W$ of type array. The property $F_\phi$, characterised by $\phi(P, W)$ consists of all $(p, w)$ such that $F_\phi$ evaluates to true when assigning the values $p$ to $P$ and $w$ to $W$ and interpreting the formula in the canonical model where the functions and predicates related to theories have their canonical meaning ('$+$' is interpreted as addition on natural numbers, '$<$' is the less-than relation, etc.).

**Example 14.4.** *The majority criterion (Example 14.1) can be characterised using the following formula:*

$$\begin{aligned} \phi(P,W) \;=\; & \forall a(\exists i(\forall k \forall k'((1 \leqslant k \wedge k < k' \wedge k' \leqslant \lfloor \tfrac{n}{2} \rfloor + 1) \rightarrow \\ & \qquad (i[k] \neq i[k'] \wedge P[i[k]] = a)) \\ & ) \rightarrow a = W[1]) \end{aligned}$$

*This formula expresses that, for all alternatives $a$, if there is an array $i$ of size $\lfloor \frac{n}{2} \rfloor + 1$ (which is the required majority) such that (1) the elements of $i$ are pairwise distinct indices $i[k]$ and, for all these indices, (2) $P[i[k], 1] = a$], i.e., alternative $a$ is the most preferred alternative in the $i[k]$th ballot in the profile, then $a = W[1]$, i.e., $a$ is elected.*

*Note, that $P, W$ are the only free variables but there can be any number of additional quantified variables in $\phi$. Moreover, the number $n$ of voters is not a variable, but either a concrete number or an uninterpreted constant.*

Correspondingly, a relational property (Definition 14.3), which is a set of pairs of evaluations, can be characterised by a formula $\phi(P_1, W_1, P_2, W_2)$ that has four free variables.

[2] Sometimes also referred to as (strict) non-imposition property.

## 14.3 Program Verification Methods

In the following, we apply the insights for classifying and formalising axiomatic properties from Section 14.2 in the field of computer-aided automated verification of voting rules, where these rules are validated against 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.

### 14.3.1 SMT Solvers and Software Bounded Model Checking

SAT solvers are programs that decide the satisfiability of a given set of formulas in propositional logic. SMT solvers go beyond SAT in that they can handle formulas in first-order logic with quantifiers and theories. They provide domain-specific and highly optimised solvers for arithmetics, arrays, uninterpreted functions, and so on. SMT solvers have evolved into powerful reasoning tools that are successfully used in model checking and software verification. As will be reported during the course of this chapter, SMT solvers can be used to check a voting rule with respect to a formalised property for particular input/output pairs, i.e., for testing the rule, without the need to implement a checker for the particular property.

SMT solvers also form the basis of modern software bounded model checking (SBMC), which goes beyond mere testing and allows to verify voting rules for all inputs of a particular size. SBMC statically analyses programs up to a predefined number of loop iterations and recursions. Programs are symbolically executed and checked for errors up to the given bound. Beyond the bound, no formal correctness guarantee can be obtained. Nevertheless, the restriction to a finite scope is justified because (1) it allows for fully automated proof search, and (2) typical faults manifest themselves already in small instances (small-scope hypothesis (Jackson, 2006)).

### 14.3.2 Relational Verification

Relational properties (Definition 14.3) relate the behaviour of a voting rule for two independent inputs (profiles). For verification, two runs of the same program $\alpha$, implementing the voting rule, need to be analysed and their results compared. A common technique, called *self-composition* (Darvas et al., 2005; Barthe et al., 2011), for proving a relational property for a program $\alpha$ is to show a functional property for the concatenation "$\alpha_1 \,;\, \alpha_2$", combining the behaviour of two variants $\alpha_1$ and $\alpha_2$ of $\alpha$ that are identical up to variable names, hence operating on disjoint variable sets, and storing the outputs in disjoint variable sets as well. Based on Hoare logic (Hoare, 1969), we then verify a relational property $R$ by running "$\alpha_1 \,;\, \alpha_2$" with (symbolic) inputs $p_1, p_2$ with the results $w_1, w_2$, and proving $((p_1, w_1), (p_2, w_2)) \in R$.

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 the exploration space 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 properties.

Another way to handle relational verification, which improves on self-composition, is to weave the two variants into a single combined program. Since $\alpha_1$ and $\alpha_2$ have disjoint variable sets, reordering statements cannot have an effect on the result as long as the execution order of statements is preserved. Details about the possibilities of flexibly weaving programs can be found in the work by Felsing et al. (2014) and Barthe et al. (2011). 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

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

This weaved program does not require separate loop invariants for the loops in $\alpha_1$ and $\alpha_2$ 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 (functional) 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.

### 14.3.3 Symmetry Breaking

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

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. Exploiting symmetries is an important technique to make testing or formal verification more feasible.

The idea is to only prove that a voting rule $f$ satisfies a functional property $F$ for a small subset $X \subseteq \mathcal{B}^*$ of the possible profiles, i.e., $(x, f(x)) \in F$ for all $x \in X$, and to then make use of the symmetry property to conclude that the same holds for all profiles $p \in \mathcal{B}^*$, i.e., $f$ 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 $f$ is symmetric with respect to a symmetry relation $S$ for which $X$ are the representatives—or if we can assume an existing proof because the symmetry is an interesting property in its own right (anonymity, neutrality, monotonicity, etc.). In the specification used for verification, the restriction to the set $X$ is achieved by a first-order logic predicate $\psi$, called a *symmetry-breaking predicate* (SBP), a term originating from the field of constraint satisfaction (Crawford et al., 1996). The formula $\psi(P)$ has a free variable $P$, and $X_\psi \subseteq \mathcal{B}$ is the set of profiles that satisfy $\psi(P)$.

In addition to establishing $(x, f(x)) \in F$ for all $x \in X$, we also have to establish (1) that $f$ has symmetry property $S$, i.e., it produces symmetric outputs for symmetric inputs, (2) all elements in $\mathcal{B}^*$ are represented by (i.e., are symmetric to) at

least one element in $\mathcal{B}^*$ which satisfies $\psi$, and (3) for any evaluation $(p, w)$ satisfying property $F$, all evaluations $(p', w')$ symmetric to $(p, w)$ also satisfy $F$. Note, that only (1) needs to be proven for the specific voting rule *f*, while (2) and (3) only depend on $F$, $S$, and $X$. Propositions (2) and (3) can hence 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). The proof for $(x, f(x)) \in F$ can be done using program verification techniques, using $\psi$ as an assumption in the proof. More details on this technique may be found in Beckert et al. (2016a).

## 14.4 Experiences

In this section, we report on experiences applying program verification to analyse voting rules w.r.t. axiomatic properties. We also present an application for aiding in auditing processes by automatically computing election margins.

### 14.4.1 Checking Properties of Single Transferable Vote

Seemingly innocuous revisions to a voting rule can have serious implications on its properties. In this case study, we show that undesired implications can be uncovered using an SMT solver to check the rule's properties (Beckert et al., 2013, 2014b). The application target is a particularly interesting variant of the single transferable vote (STV) algorithm (CADE-STV) used to elect the board of trustees of the International Conference on Automated Deduction (CADE).

The property to be checked is a tailor-made criterion that is intended to capture the essence of *proportional representation*, stating that "there must be enough votes for each elected alternative". This criterion only considers the number of votes for an alternative and ignores the order of preferences within ballots. More specifically, this property requires that the input profile can be partitioned into (disjoint[3]) groups of ballots such that each elected alternative is supported by one voter group of sufficient size:

**Definition 14.4 (Criterion: Enough votes for each elected alternative).** *Let q be the quota and s the number of alternatives to be elected (e.g., the number of seats in a parliament). The property* $Enough_{q,s}$ *consists of all evaluations* $(p, \langle a_1, \ldots, a_{s'} \rangle)$ *for which there are multisets* $m_1, \ldots, m_{s'}, m_{rest}$ *(*$s' \leqslant s$*) that form a partition of the ballots in the profile* $p$*, i.e.,*

$$\{b \mid b \text{ is a ballot in } p\} = m_1 \dot\cup \ldots \dot\cup m_{s'} \dot\cup m_{rest}$$

*such that, for* $0 < i \leqslant s'$ *(i.e., not taking* $m_{rest}$ *into account), the following holds:*

1. $|m_i| = q$ *(there are exactly q voters in each class that support an elected alternative),*
2. *for all* $b \in m_i$*, there is a preference position* $j$ *such that* $b_j = a_i$ *(each vote* $b$ *in the class* $m_i$ *supports alternative* $a_i$*, i.e., the alternative occurs at some position* $j$ *among the preferences in* $b$*).*

[3]Hence, we will use the operator $\dot\cup$ for disjoint unions.

**Example 14.5.** *Assume there are four alternatives $a, b, c, d$ for two vacant seats, the profile $p$ consists of five ballots $[a, b, d]$, $[a, b, d]$, $[a, b, d]$, $[d, c]$ and $[c, d]$, and the quota is $q = 2$. The evaluation $(p, \langle a, d\rangle)$ satisfies property $Enough_{2,2}$ using the partition consisting of $\{[a, b, d], [a, b, d]\}$, $\{[c, d], [d, c]\}$ and $\{[a, b, d]\}$, where the first group supports alternative $a$ and the second one supports alternative $d$.*

**Formalisation.** We formalise property $Enough_{q,s}$ by a formula $\phi(P, W)$, which uses an existentially quantified variable *part* of type array that represents the partition and the assignment of classes in the partition to elected alternatives as follows:

$$part[i] = \begin{cases} k & \text{if the } i\text{th vote supports the } k\text{th elected alternative } W[k] \\ 0 & \text{if the } i\text{th vote does not support any elected alternative} \end{cases}$$

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

$$\forall i\big(0 < i \leqslant n \rightarrow 0 \leqslant part[i] \leqslant s\big) \qquad (\phi_1)$$

$$\forall i\big(0 < i \leqslant n \rightarrow (part[i] \neq 0 \rightarrow W[part[i]] \neq 0)\big) \qquad (\phi_2)$$

$$\forall i\big((0 < i \leqslant n \wedge part[i] \neq 0) \rightarrow \exists j(0 < j \leqslant m \wedge P[i, j] = W[part[i]])\big) \qquad (\phi_3)$$

$$\begin{aligned} &\forall k\big((0 < k \leqslant s \wedge W[k] \neq 0) \rightarrow \qquad (\phi_4)\\ &\quad \exists\, count\,(\, count[0] = 0 \wedge \\ &\qquad\qquad \forall i(0 < i \leqslant n \rightarrow (part[i] = k \rightarrow count[i] = count[i-1] + 1) \wedge \\ &\qquad\qquad\qquad\qquad (part[i] \neq k \rightarrow count[i] = count[i-1])) \wedge \\ &\qquad\qquad count[n] = q)\big) \end{aligned}$$

Formulas $\phi_1$ and $\phi_2$ express well-formedness of the partition. Formula $\phi_3$ expresses that for every vote supporting an alternative, that alternative must be ranked somewhere in that vote. Formula $\phi_4$ expresses that each class supporting a particular elected alternative has exactly $q$ elements. To formalise this, we use an array *count* such that $count[i]$ is the number of supporters among votes $1, \ldots, i$ that support the $k$th elected alternative.

**Checking the Property Using an SMT Solver.** To check the property, we generated input profiles (a) randomly and (b) exhaustively starting from small profiles. The corresponding election results were computed using an implementation of CADE-STV in Python. The formula $\phi(P, W)$ was then evaluated for pairs of profiles and results using the SMT solver Z3 (De Moura and Bjørner, 2008). Note, that the evaluation is not trivial because of the existential quantifier in $\phi$. Using an SMT solver in this way amounts to testing.

CADE-STV differs from standard STV in that, after an alternative is elected, all ballots are still in play. In standard STV, however, ballots that have been used to elect an alternative are not available for the next round. Moreover, CADE-STV uses a quota different from standard STV, namely the absolute majority of votes. Because of its non-standard behaviour, CADE-STV does not satisfy property $Enough$ from Definition 14.4.

**Example 14.6.** *Let us run CADE-STV on Example 14.5. First, we compute the majority quota $q = 3$. In the first iteration, alternative $a$ has three first preferences, so that $a$ is the majority winner and is seated. Since CADE-STV uses restart*[4]*, $a$'s votes are not deleted but are redistributed at the end of the first iteration. Now the profile contains $[b, d], [b, d], [b, d], [d, c], [c, d]$. Following the algorithm, we observe that now alternative $b$ has the majority 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). Obviously, there is no partition of the votes to support both alternatives $a$ and $b$ with 3 ballots each, i.e., property $Enough$ does not hold.*

Indeed, our checker based on Z3 easily finds smaller counterexamples than the one shown in Example 14.6, but these are not as illustrative. Run-times of Z3 for checking the property for a single evaluation (i.e., performing a single test) are in the range of a few seconds for profile sizes of up to about 30 ballots and 30 alternatives. Thus, random testing for small profiles is easily possible. Exhaustive testing quickly becomes infeasible because of the exponential number of possible profiles. In Beckert et al. (2013), we report on more details using an SMT solver for checking this and other properties of STV, as well as possible solutions to the undesired effects in CADE-STV.

### 14.4.2 Property Verification for Voting Rules

Below, we report on experiences with relational verification and symmetry breaking techniques (see Section 14.3). For our case study, we used the automated software model checker CBMC (Clarke et al., 2004), which takes C/C++ programs as input that are annotated with specifications in the form of assertions and assumptions. We performed our computations using CBMC's SAT back end based on MiniSat (Eén and Sörensson, 2003) in combination with an efficient bit-vector refinement procedure (Bryant et al., 2007).

**Relational Verification Using CBMC.** As explained in Section 14.3.2, relational verification with weaved programs and coupling invariants is often 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 simple first-past-the-post plurality voting with respect to the anonymity property. For plurality voting, anonymity can be written as follows:

$$\begin{aligned}\phi_{Anon}(P_1, W_1, P_2, W_2) \;=\; \forall b_1, b_2\big(\;\big(\;& \forall i(0 < i \leqslant n \rightarrow (0 < P_1[i] \leqslant m \wedge 0 < P_2[i] \leqslant m)) \wedge \\ & 0 < b_1 < b_2 \leqslant n \wedge \\ & P_1[b_1] = P_2[b_2] \wedge P_2[b_1] = P_1[b_2] \wedge \\ & \forall i((0 < i \leqslant n \wedge i \neq b_1 \wedge i \neq b_2) \rightarrow P_1[i] = P_2[i]) \\ \big) \rightarrow W_1 = W_2\big)\end{aligned}$$

[4] Deviating from standard STV, whenever a seat gets assigned by electing an alternative, the next seat is to be assigned based on the original profile (only having the so far elected alternatives removed), i.e., keeping ballots which already contributed, and resurrecting already eliminated alternatives.

```
1 void anonymity(int p1[N], int p2[N]) {
2     for (int i = 1; i <= N; i++) {
3         assume (0 < p1[i] ⩽ M ∧ 0 <p2[i] ⩽ M) }
4     int b1, int b2; assume (0 < b1 ⩽ N ∧ 0 < b2 ⩽ N ∧ b1 < b2);
5     assume (p1[b1] == p2[b2] ∧ p2[b1] == p1[b2]);
6     for (int i = 1; i <= N; i++) {
7         if (i != b1 && i != b2) assume (p1[i] == p2[i]); }
8     int w1 = plurality(p1);
9     int w2 = plurality(p2);
10    assert (w1 == w2);
11 }
```

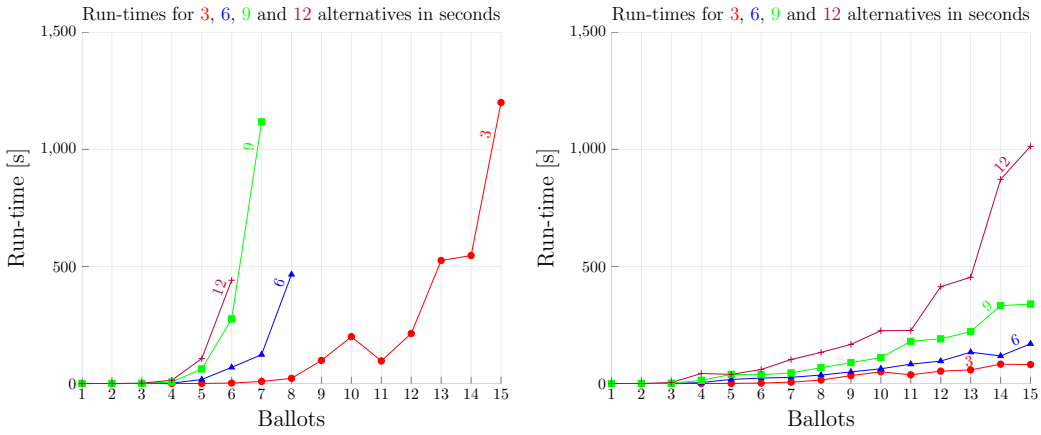
Listing 1: Anonymity property as a C program

Anonymity is a relational property (see Section 14.2.3). It is formalised as a first-order logic formula $\phi_{Anon}(P_1, W_1, P_2, W_2)$, using four free variables denoting two profiles and two election results, respectively. Since plurality voting is a single-choice voting rule (it is not preferential), we assume the profiles to be one-dimensional arrays, i.e., the $i$th ballot $p[i]$ of profile $p$ equals the $i$th voter's single choice and is not itself an array. Moreover, we assume that, in case of a tie, no alternative is elected and that this is indicated by the election result of $w = 0$.

Listing 1 shows the corresponding CBMC specification, expressing that the voting rule implemented in the C function `plurality` satisfies the anonymity property. Lines 2 and 3 express the assumption that the profiles are well-formed. The universal quantification of variable `i` is expressed using a `for`-loop. The variables $b_1, b_2$ introduced by the assumption in Line 4 are implicitly universally quantified as CBMC carries out the proof for all values satisfying the assumptions. The profiles $p_1, p_2$ are assumed to only differ in ballots of voters $b_1, b_2$ in that the ballots of these two voters are exchanged. This is expressed in Lines 5 to 7. The function `plurality` is invoked in Lines 8 and 9 to compute the election result for the two profiles $p_1, p_2$. Finally, Line 10 makes the assertion that the two election results are identical. CBMC will prove that this assertion holds for all inputs (a) whose size is within a given bound and that (b) satisfy the assumptions from Lines 2 to 7.

We used CBMC to verify the anonymity property for plurality voting using (a) simple composition of two variants without coupling invariants and (b) weaved programs with coupling invariants for the loops (the implementation of plurality voting has two loops, one counting the ballots for each alternative and one for finding the alternative with the maximum number of votes).

Figure 14.1 shows the run-times (in seconds) for between 1 and 12 alternatives and 1 to 15 ballots. For the missing data points, the run-times exceed our predefined time-out of 30 minutes. The results show that 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 alternatives and 25 ballots (in contrast to only 10 alternatives and 7 ballots without coupling invariants).




(a) Without coupling invariants

(b) Weaved program with coupling invariants

Figure 14.1: Verification of the anonymity property for plurality voting

**Symmetry Breaking.** We continue the case study, now with the goal to verify the majority criterion (Example 14.1) for plurality voting.

Using CBMC in a straightforward manner, verification is possible for small bounds on the numbers of voters and alternatives, but becomes infeasible for higher numbers. For example, a time-out of 30 minutes is reached with 5 alternatives and 45 voters resp. with 10 alternatives 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 10 alternatives are shown in the second column of Table 14.1 ('t/o' indicates time-out). The full data can be seen in Beckert et al. (2016a).

Using symmetry breaking, however, the efficiency of verification can be considerably increased—and, thus also, the reachable bounds. Assuming anonymity, which is a symmetry property, by applying the symmetry breaking predicate

$$\forall i(0 < i < n) \rightarrow P[i-1] \leqslant P[i],$$

the situation improves dramatically. This predicate requires the ballots to be sorted according to which alternatives they prefer. Intuitively, this is a valid assumption as anonymity allows to re-order the ballots.

The much lower run-times are shown in the right column of Table 14.1. Experiments show that handling more than 100 ballots for 10 alternatives becomes feasible, when adding predicates for further symmetry properties.

### 14.4.3 Margin Computation

A method to create confidence in the outcome of an election is to audit the election result against the physical evidence, i.e., the ballots. Some auditing methods require the computation of a margin (Stark and Teague, 2014). Below, we present

| Ballots | Without Symm. Red. | With Symm. Red. |
|---|---|---|
| 5 | 1.2 | 0.2 |
| 10 | 41.7 | 0.9 |
| 15 | 84.3 | 3.8 |
| 20 | t/o | 6.9 |
| 50 | t/o | 194.2 |
| 80 | t/o | 747.9 |
| 85 | t/o | 855.4 |
| 90 | t/o | 1,369.6 |
| 95 | t/o | t/o |

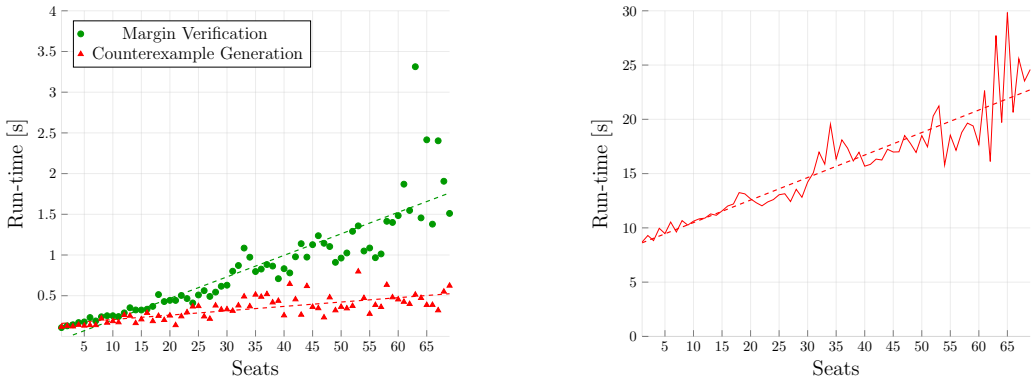
Table 14.1: Verification of the majority property for plurality voting for 10 alternatives (run-times in seconds)

a technique based on software bounded model checking for computing the margin of an election (consult the work in Beckert et al. (2016b) for more details). This application is different from verification as it provides information about particular elections instead of the voting rule in general.

The margin is the minimal number of votes that would need to be misfiled in order to change the election outcome. The margin is identical to the number of votes that would have had to be miscounted or tampered with during tabulation. If the election margin is large, only a small sample needs to be drawn and audited. The smaller the margin, the larger the sample. In the worst case, the audit will trigger a full manual recount. We assume that a voting rule is given (as an implementation in C) as well as a concrete input, which consists of a table aggregating the initial profile by the alternatives running for election. This table is the result of vote counting and tabulation. We model it as an integer array of size *m*, which is effectively the number of different stacks into which identical votes are accumulated during counting.

The idea of our approach is to use an SBMC tool to check an assertion claiming that, when the ballots are changed by putting at most a certain number $x$ of votes on other stacks than they were on, the outcome of the election is *not* changed. If that assertion is provable, we know that the actual election margin is greater than $x$. If the assertion is not provable, we know that the actual election margin is less than or equal to $x$. In the latter case, the SBMC tool generates a counterexample to the assertion demonstrating that the election outcome can be changed by changing $x$ votes. Having this proof obligation as a basis, we can use binary search to find a value for $x$ such that the assertion holds for $x-1$, but fails for $x$, i.e., $x$ is exactly the election margin. The main advantage of this method is that it can be uniformly and automatically applied to arbitrary rules without any adaptation.

In contrast to our work, there has been a lot of research on how to compute margins for *specific* voting rules, for which that problem is particularly hard (Bartholdi and Orlin, 1991; Cary, 2011; Sarwate et al., 2013; Magrino et al., 2011; Blom et al., 2016).




(a) Time for last step in computation.

(b) Accumulated time for whole computation.

Figure 14.2: Run-times of automatic margin computation for the Jefferson method with various values for the total number of seats to be allocated.

If the implementation of a voting rule is based on choosing or searching for some parameter, then margin computation can be made much more efficient by replacing the search for the parameter by a non-deterministic choice to be resolved by the SBMC tool. An example is the Jefferson method, similar to *largest-remainder* methods such as the Hare-Niemeyer method. Here, a quota is chosen, i.e., the number of votes needed for one seat, such that the resulting seats per alternative, when rounded down to the next natural number, sum up to the required total number of seats.

We demonstrated our approach on the 2005 Schleswig-Holstein state elections with various values for $s$, the total amount of seats to be allocated. The run-times are shown in Fig. 14.2a and Fig. 14.2b, with all computations well below $30$ seconds. The election margin for the original number of seats ($69$) is $634$ ballots. The computed margins range from only $42$ (for $s = 62$) to $177,863$ (for $s = 2$). Performing our method for various values of $s$ scales well on the Jefferson method, as there is no loop depending on the value of $s$ (in contrast to the "imperative" version known as D'Hondt method). However, further experiments also indicated a non-exponential dependency on the value for $m$. For example, an allocation of $69$ seats to $10$ alternatives takes about $55$ seconds, whereas for $20$ alternatives, the analysis takes about $300$ seconds. We also demonstrated the applicability of our approach to a further, more complex real-world election: the Danish parliamentary elections in 2015. The Danish elections use a two-tier system, allocating $135$ seats using the D'Hondt/Jefferson method for each of the lower-tier electoral districts (Elklit et al., 2011 (accessed August 23, 2016), and allocating the remaining $40$ seats using the Saint-Laguë method. We performed our analysis on the first tier for the first $135$ seats. Using the Jefferson method, we compute a margin of $10$ votes within $7,815$ seconds, i.e., around $2$ hours and $10$ minutes. The final verification (proving that a change in $9$ votes cannot change the election outcome) takes $53$ seconds and a counterexample for $10$ votes (i.e., an example profile that changes the election outcome) can be found within $27$ seconds.

## 14.5 Summary and Related Work

**Summary.** We have seen that SMT solvers and software bounded model checking can be effectively used for the verification of voting rules w.r.t. functional and relational properties as well as for finding violations of such properties. 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 regarding formal analysis. Moreover, verification techniques can be important parts in a process for the design and development of verified tailor-made voting rules with clear and trustworthy axiomatic characterisations, eventually leading to reliable electoral laws. For this purpose, semantic criteria need to be explicitly stated instead of a mere discussion of voting rules using anecdotal descriptions of individual scenarios.

Our experiences with bounded model checking demonstrated 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. Taking the small-scope hypothesis into account, 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).

Beyond a formal analysis of voting rules with respect to semantic criteria, we presented a further application of bounded model checking in the domain of social choice research: computing election margins fully automatically. It can be applied to arbitrary implementations of voting rules without understanding or even knowing how the election result is computed. Our approach scales for implementations of real-world voting rules in real elections if the number of loop iterations in the voting rule does not go beyond a few hundred.

**Related Work.** In this chapter, we have discussed the use of program verification technology based on first-order logic for the verification of voting rules. There are also other approaches using tactical theorem provers and higher-order logic. Dawson et al. (2015) specify a complex voting rule according to legal text in higher-order logic and verify its SML implementation against this specification. Moreover, Pattinson and Schürmann (2015) encode voting rules into axioms for a tactical theorem prover, which is then used to produce certificates for election results by their implementions. Examples which verify voting rules against axiomatic properties are proofs carried out by Goré and Meumann (2014) and Beckert et al. (2014a). Verification using tactical theorem provers may lead to very high confidence levels, but the task is inherently difficult and time-consuming, resulting in huge and laborious interactive proofs. In Beckert et al. (2016a), we have also used the semi-interactive deductive theorem-prover KeY (Ahrendt et al., 2014) for a case study proving axiomatic properties regardless of the input size using the technique of relational verification as covered in Section 14.3.2. Conducting proofs in KeY is less automatic than SBMC and requires further user interaction and additional specifications such as, e.g., loop invariants, but it enables full proofs for all inputs without any bounds.

Furthermore, there is research on the verification of concrete voting systems, i.e., considering concrete voting rules and software (Dennis et al., 2008; McGaley and Gibson, 2005; Kiniry et al., 2007; Cochran, 2012; Goré and Meumann, 2014; Dawson et al., 2015). With a focus on security, we have conducted an extensive case study on an electronic voting system by using a combination of different program verification techniques (Küsters et al., 2015). Finally, a multitude of theoretical work on proving and finding new incompatibilities of voting rule properties has been done using SAT solvers (Tang and Lin, 2009; Geist and Endriss, 2011; Brandt et al., 2016; Chatterjee and Sen, 2014; Brandt and Geist, 2016). They analyse only properties while encoding or generating abstract voting rules meeting some given assumptions, i.e., an encoding of a mapping from profiles to sets of alternatives is further constrained to form a manageable subset in order to mitigate state-space explosion. More on this topic can be found in Chapter 13.

## Acknowledgments

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. This publication was made possible by NPRP grant NPRP 7-988-1-178 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

## Bibliography

W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In D. Giannakopoulou and D. Kroening, editors, *Proceedings of the 6th International Conference on Verified Software: Theories, Tools and Experiments (VSTTE), Revised Selected Papers*, volume 8471 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2014.

K. J. Arrow. *Social Choice and Individual Values*, volume 12 of *Cowles Foundation Monographs*. Yale University Press, 2nd edition, 1963.

G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011.

J. J. Bartholdi and J. Orlin. Single transferable vote resists strategic voting. *Social Choice and Welfare*, 8, 1991.

B. Beckert, R. Goré, and C. Schürmann. Analysing vote counting algorithms via logic - and its application to the CADE election scheme. In M. P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*

*(CADE)*, volume 7898 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2013.

B. Beckert, T. Bormer, R. Goré, M. Kirsten, and T. Meumann. Reasoning about vote counting schemes using light-weight and heavy-weight methods. In *8th International Verification Workshop (VERIFY) in connection with the 7th International Joint Conference on Automated Reasoning (IJCAR)*, 2014a.

B. Beckert, R. Goré, C. Schürmann, T. Bormer, and J. Wang. Verifying voting schemes. *Journal of Information Security and Applications (JISA)*, 19(2), 2014b.

B. Beckert, T. Bormer, M. Kirsten, T. Neuber, and M. Ulbrich. Automated verification for functional and relational properties of voting rules. In *Sixth International Workshop on Computational Social Choice (COMSOC)*, 2016a.

B. Beckert, M. Kirsten, V. Klebanov, and C. Schürmann. Automatic margin computation for risk-limiting audits. In R. Krimmer, M. Volkamer, J. Barrat, J. Benaloh, N. J. Goodman, P. Y. A. Ryan, and V. Teague, editors, *Proceedings of the 1st International Joint Conference on Electronic Voting – formerly known as EVOTE and VoteID (E-Vote-ID)*, volume 10141 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2016b.

M. L. Blom, V. Teague, P. J. Stuckey, and R. Tidhar. Efficient computation of exact IRV margins. In G. A. Kaminka, M. Fox, P. Bouquet, yke Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI), Including Prestigious Applications of Artificial Intelligence (PAIS)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2016.

F. Brandt and C. Geist. Finding strategyproof social choice functions via SAT solving. *Journal of Artificial Intelligence Research (JAIR)*, 55, 2016.

F. Brandt, C. Geist, and D. Peters. Optimal bounds for the no-show paradox via SAT solving. In C. M. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, editors, *Proceedings of the 15th International Conference on Autonomous Agents & Multiagent Systems*. ACM, 2016.

R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

D. Cary. Estimating the margin of victory for instant-runoff voting. In H. Shacham and V. Teague, editors, *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE)*. USENIX Association, 2011.

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

E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2004.

D. Cochran. *Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms*. PhD thesis, IT University of Copenhagen, Copenhagen, Denmark, 2012.

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

Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings of the 2nd International Conference on Security in Pervasive Computing (SPC)*. Springer, 2005.

J. E. Dawson, R. Goré, and T. Meumann. Machine-checked reasoning about complex voting schemes using higher-order logic. In R. Haenni, R. E. Koenig, and D. Wikström, editors, *Proceedings of the 5th International Conference on E-Voting and Identity (Vote-ID)*, volume 9269 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2015.

L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 5295 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2003.

J. Elklit, A. B. Pade, and N. Nyholm Miller. The parliamentary electoral system in Denmark, 2011 (accessed August 23, 2016). URL `http://www.ft.dk/Dokumenter/Publikationer/Engelsk/The_Parliamentary_Electorial_System_Denmark.aspx`.

S. Falke, F. Merz, and C. Sinz. The bounded model checker LLBMC. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2013.

D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In I. Crnkovic, M. Chechik, and P. Grünbacher, editors, *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2014.

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

C. Geist and U. Endriss. Automated search for impossibility theorems in social choice theory: Ranking sets of objects. *Journal of Artificial Intelligence Research (JAIR)*, 40, 2011.

R. Goré and T. Meumann. Proving the monotonicity criterion for a plurality vote-counting program as a step towards verified vote-counting. In R. Krimmer and M. Volkamer, editors, *Proceedings of the 6th International Conference on Electronic Voting: Verifying the Vote (EVOTE)*. IEEE Computer Society, 2014.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

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

J. R. Kiniry, D. Cochran, and P. E. Tierney. Verification-centric realization of electronic vote counting. In R. Martinez and D. Wagner, editors, *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT)*. USENIX Association, 2007.

R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A hybrid approach for proving noninterference of Java programs. In C. Fournet and M. Hicks, editors, *28th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2015.

T. R. Magrino, R. L. Rivest, E. Shen, and D. Wagner. Computing the margin of victory in IRV elections. In *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE)*. USENIX Association, 2011.

M. A. McGaley and J. P. Gibson. Electronic voting: An analysis of the safety critical issues. In *National Symposium of The Irish Research Council for Science, Engineering and Technology*, 2005.

D. Pattinson and C. Schürmann. Vote counting as mathematical proof. In *Proceedings of the 28th Australasian Joint Conference on Advances in Artificial Intelligence (AI)*, volume 9457 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2015.

A. Sarwate, S. Checkoway, and H. Shacham. Risk-limiting audits and the margin of victory in nonplurality elections. *Statistics, Politics, and Policy*, 4(1), 2013.

P. B. Stark and V. Teague. Verifiable European elections: Risk-limiting audits for D'Hondt and its relatives. *USENIX Journal of Election Technology and Systems (JETS)*, 1(3), 2014.

P. Tang and F. Lin. Computer-aided proofs of Arrow's and other impossibility theorems. *Artificial Intelligence*, 173(11), 2009.