Karlsruhe Institute of Technology

Formal Verification of Voting Schemes

Diploma Thesis by

Michael Kirsten

Department of Informatics
Institute of Theoretical Informatics (ITI)
Application-oriented Formal Verification

Supervisor:	Prof. Dr. rer. nat. Bernhard Beckert
Advisors:	Dipl.-Inform. Thorsten Bormer
	Prof. Dr. Rajeev Goré

December 21, 2014

Abstract

Fundamental trust and credibility in democratic systems is commonly established through the existence and execution of democratic elections. The vote-counting of an election, usually formalised by a *voting scheme*, essentially boils down to a mechanism that aggregates individual preferences of the voters to reach a decision. For this matter, there are various differing voting schemes in use throughout the world, commonly based on high expectations and means to ensure a sensible democratic process. However, incidents such as the ruling by the German federal constitutional court which led to a change of the German legislation in 2013 manifest that it is difficult for a voting scheme to meet these legitimate expectations [BGB13]. In fact, there is no general notion of correctness for a voting scheme and thus no universal mechanism as shown in Kenneth J. Arrow's *Impossibility Theorem* in 1951 [Arr51]. As a consequence, designing a real-world voting scheme without flaws, which still gives significant democratic guarantees, is a difficult task as a trade-off between desirable properties is non-trivial and error-prone.

The approach in this thesis is based on the idea to tackle this issue by proposing an incremental and iterative development process for voting schemes based on automated formal reasoning methods using program verification. We analyse two different forms of verification considering their role in this development process in order to achieve formal correctness of voting schemes. We perform a comprehensive set of case studies by applying "medium-weight" and "light-weight" verification techniques. The "medium-weight" approach uses the annotation-based deductive verification tool VCC based on an *auto-active* methodology and the "light-weight" technique is performed with the bounded model checking tool LLBMC. Our analysis covers a set of well-known voting schemes combined with a set of prominent voting scheme criteria. In addition to giving precise formalisations for these criteria adapted to the specific voting schemes and tools used, we advance the efficiency of the "light-weight" approach by exploiting fundamental symmetric properties. Furthermore, we investigate on encountered challenges posed by the *auto-active* verification methodology, which lies in-between automatic and interactive verification methodologies, with respect to specific characteristics in voting schemes and also explore the potential of bounded verification techniques to produce precise counterexamples in order to enhance the capability of our envisioned development process to give early feedback. This thesis gives fundamental insights in general challenges and the potential of automated formal reasoning with the goal of correct voting schemes.

Deutsche Zusammenfassung

Ein großer Teil des Vertrauens und der Glaubwürdigkeit in demokratischen Systemen wird gemeinhin durch das Abhalten und den Ablauf von demokratischen Wahlen geschaffen. Die Entscheidungsfindung einer Wahl, gewöhnlich durch ein *Wahlverfahren* formalisiert, ist im Kern ein Mechanismus zur Aggregation individueller Wählerpräferenzen auf ein Wahlergebnis. Für diese Aufgabe wird eine Vielzahl unterschiedlicher Wahlverfahren in vielen Ländern der Erde angewandt, die üblicherweise auf hohen gesellschaftlichen Ansprüchen basieren um einen angemessenen demokratischen Prozess sicherzustellen. Jedoch zeigen Vorfälle wie das Urteil des Bundesverfassungsgerichts der Bundesrepublik Deutschland, das 2013 zu einer Änderung der deutschen Verfassung führte, dass es schwierig ist, ein Wahlverfahren zu finden, dass diese berechtigten Erwartungen erfüllt [BGB13]. In der Tat existiert keine allgemein gültige Auffassung von Korrektheit für ein Wahlverfahren und in der Konsequenz auch keine universelle Prozedur, wie schon Kenneth J. Arrow 1951 in seinem *Allgemeinen Unmöglichkeitstheorem* zeigte [Arr51]. Daraus ergibt sich, dass der Entwurf eines realistischen fehlerfreien Wahlverfahrens, das dennoch angemessene demokratische Zusicherungen gibt, eine schwierige Aufgabe ist, da ein Ausgleich zwischen wünschenswerten Eingenschaften fehleranfällig und nicht trivial ist.

Das Vorgehen dieser Arbeit um diesen Missstand anzugehen basiert auf der Idee eines stufenweisen und iterativen Entwicklungsprozesses für Wahlsysteme basierend auf automatischen formal-logischen Beweisverfahren. Es werden zwei verschiedene Beweistechniken bezüglich ihrer Rolle in diesem Entwicklungsprozess analysiert, um formale Korrektheit für Wahlverfahren zu erreichen. Dabei wird eine umfassende Fallstudie durchgeführt, bei der verschiedene Verifikationstechniken, die wir als "leichtgewichtig" und "mittel-gewichtig" klassifizieren, zur Anwendung kommen. Der "mittelschwere" Ansatz wird durch das annotationsbasierte deduktive Verifikationswerkzeug VCC unterstützt, das auf einer *auto-aktiven* Methodik basiert und die "leichtgewichtige" Technik nutzt das Werkzeug LLBMC zur beschränkten Modellprüfung (*bounded model checking*). Die Analyse beinhaltet bekannte Wahlverfahren kombiniert mit allgemeinen Korrektheitskriterien für Wahlverfahren. Über die Aufstellung präziser Formalisierungen dieser Kriterien angepasst auf die spezifischen Wahlverfahren hinaus wird die Effizienz des "leichtgewichtigen" Ansatzes durch die Ausnutzung fundamentaler Symmetrieeigenschaften vorangebracht. Außerdem untersucht diese Arbeit Herausforderungen, die sich aus der *auto-aktiven* Methodik bezüglich bestimmter Charakteristika von Wahlverfahren ergeben, und erkundet das Potential von Verifikationstechniken innerhalb bestimmter Grenzen bezüglich der Eingabegrößen, präzise Gegenbeispiele zu generieren, um den langfristig angestrebten Entwicklungsprozess um die Möglichkeit zu erweitern, frühzeitige Rückmeldungen bezüglich der Korrektheit zu geben. Diese Arbeit erarbeitet fundamentale Einsichten in allgemeine

Herausforderungen und das Potential automatisierter formal-logischer Beweisverfahren bezüglich korrekter Wahlverfahren.

Acknowledgements

I would like to thank Prof. Dr. Bernhard Beckert and Prof. Dr. Rajeev Goré for giving me the opportunity to be part of this exciting project and the experiences gained during my visit at the Australian National University (ANU) in Canberra, Australia. Great thanks go especially to Prof. Dr. Bernhard Beckert for supporting my work during my visit and beyond, including lots of productive and motivating discussions. A special thanks goes to my advisor Thorsten Bormer for taking the time for me, having lots of patience, motivating me and many lively debates supporting my work. Last but not least, I want to thank my parents for all the support for me and my studies over the years.

Statement of Authorship

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, December 21, 2014

. . . .

(Michael Kirsten)

Contents

List of Definitions

List of Figures

List of Tables

List of Algorithms

List of Listings

1. Introduction

Nowadays, almost one-half of the world's countries, inhabited by one-half of the world's population, are ruled by democratic systems and are as such commonly denoted as free and modern societies [Uni13]. This is legitimated by enabling approval and participation of the people leading to a lawful, balanced, self-determined and sovereign government by the people through representation. Approval and participation of the people are mainly exercised through free, secret and universal elections in order to determine appropriate representatives of the people to rule the state. Fundamental trust and credibility in the system as a whole are established through the existence and procedure of these elections, and a vast majority of people agrees on the need for democratic elections. Participating in such an election is known as the process of voting and the procedure varies greatly throughout the world concerning the specific modalities. This fact already indicates that there is no commonly agreed and preferred way how to realise such a voting process. Voting and its effect, usually formalised by a *voting scheme*, boils down to a mechanism that aggregates individual preferences of the people, given in the form of ballots, to decide an election. Generally, the participating people, denoted as voters, need to be confident and convinced that the election is a credible and trustworthy process in order to establish an acceptably credible and trustworthy government.

When looking more closely at this fundamental procedure, we identify the two major elements vote-casting and vote-counting, which both play significant roles in the establishment of people's trust in and consequently reliance on the election procedure and hence the legitimacy of the resulting government. With the rise and development of technology, there are many advances in doing both these tasks using computers. This raises concerns about the correctness of the vote-casting process, but promises reducing the effects of human errors on the vote-counting process and making it more efficient, still leaving us with the general decision for a trustworthy *voting scheme*. There is a significant amount of research on how to answer the concerns with electronic vote-casting, but only little research exists on the development of trustworthy or correct *voting schemes* using computers. One development largely promoted by growing technology support is that more complex voting schemes are applied in real elections as the error-prone and labour-intensive counting by hand is more and more done by computers. The downside of this development is that with more complexity of a voting scheme, the probability of undesired side-effects increases.

An example has been seen in the German federal elections, for which the federal constitutional court ruled that the basic nature of the parliamentary elections in establishing a proportional representation was bypassed, which subsequently resulted in a change of the legislation (BWG §6) in 2013 [BTD12][BGB13]. This ruling manifests that there are high expectations and means to ensure a correct voting scheme. In this case, correctness means that the voting scheme establishes a proportional representation, but this is not the only notion of correctness for voting schemes specified in national legislations. Furthermore, even these legislations may have flaws or inconsistencies in themselves as they commonly contain multiple interpretations of notions of correctness. Moreover in 1951, Kenneth J. Arrow proved that already three specific fundamental properties for voting schemes are inconsistent with each other [Arr51]. We conclude that there has to be a trade-off between different notions of correctness, which appears to be non-trivial and error-prone as findings of undesired side-effects such as the German court ruling indicate.

1.1. Research Objectives

We propose an incremental and iterative process in order to develop correct voting schemes with appropriate and understandable correctness criteria based on automated formal methods of reasoning [Bec+14a]. For this matter, we start with a simple implementation and specification and gradually add complexity as we iron out errors in the implementation and specification, and gain insights into the practicality of the desired theoretical desiderata.
Furthermore, we propose multiple forms of verification to enable this development process, more precisely a range from "light-weight" and fully automatic methods using *software bounded model checking* (SBMC) to "medium-weight" full functional software verification using annotation-based program verification tools or "heavy-weight" interactive higher-order theorem provers. We argue for a development process enhanced by various different forms of verification and examine the suitability within this work. As a promising analysis of the suitability of "heavy-weight" verification methods using HOL4 has been done by Meumann in [Meu14], this work focuses on analysing "medium-weight" methods using VCC (Section 3.2.1) and "light-weight" methods using LLBMC (Section 3.2.2). Therein, we analyse formal reasoning for a selected set of well-known voting schemes (Section 2.3) with respect to well-known voting scheme properties (Section 2.4), declaratively formalised in first-order predicate logic.

1.2. Correctness of Voting Schemes

When talking about correctness of voting schemes, we need to clarify when we consider a voting scheme to be correct. Other than a mathematical equation, which is based on fundamental and generally accepted mathematical axioms, voting schemes do not have a generally accepted notion of correctness. Hence, we consider a given voting scheme correct or incorrect with respect to a given specification. The vagueness when talking about the correctness of voting schemes thus arises about which specification we are referencing. The field of *social choice theory* has developed and advanced a respectable variety of different correctness criteria for voting schemes along with profound examinations on how they relate to various voting schemes. Throughout this work, we will use first-order predicate logic in order to specify correct behaviour of a voting scheme. First-order

predicate logic is expressive enough to formalise appropriate properties for voting schemes, but also simple enough to produce understandable declarative properties.
Another notion of correctness considers more experimental criteria and focuses on the probability that certain criteria are met for specific voting schemes as most precise criteria are generally too strong for complete correctness [CS12]. Moreover, a soft notion of correctness was defined by the *Royal Commission on the Electoral System* in 1986 as an evaluation of possible new voting schemes for New Zealand, which addresses criteria such as fairness between political parties, effective minority representation, political integration, effective representation of constituents, effective voter participation, effective government, effective parties, as well as legitimacy [Lij87]. Notably, the report stated that the voting scheme should be understandable to the people, which seems to be a prevailing expectation. Furthermore, a so-called *distance rationalisability* has been proposed for comparing different voting schemes to each other by algorithmic analysis with rather general properties such as neutrality and consistency [EFS10]. Much more focussing on a concrete implementation, there has also been work on verifying concrete voting systems, which consider a specific voting software [DYJ08].
In this work however, we will examine abstract voting schemes with respect to concrete correctness properties specified in first-order predicate logic.

1.3. Outline

The rest of this work is structured as follows.

- Chapter 2 presents the fundamental terms, concepts and notions used in this thesis. This includes a general definition of voting schemes, fundamental insights in the difficulty of specifying and implementing correct voting schemes, as well as the four voting schemes and three correctness criteria, which we examine within this work.

- Chapter 3 introduces the two methodologies of *auto-active* deductive (which we classify as “medium-weight”) and fully automatic bounded (which we classify as “light-weight”) verification together with the two tools VCC and LLBMC, which we use in this thesis to apply the respective techniques.

- Chapter 4 presents a case study using “medium-weight” verification methods with the *auto-active* deductive verification tool VCC to prove the correctness of a simple *first-past-the-post* plurality voting scheme together with the encountered challenges and our responses to them.

- Chapter 5 then illustrates an extensive case study using “light-weight” bounded verification methods with the fully automatic verification tool LLBMC to give correctness guarantees or generate counterexamples (for incorrect voting schemes) for a selected set of voting schemes and appropriate correctness criteria.

- Chapter 6 further advances on results from Chapter 5 by using fundamental insights in the symmetric structure of the problem, leading to performance increases with

LLBMC through the reduction of the set of possible input parameters (for the matter of this thesis, these are ballot boxes).

- Chapter 7 concludes by discussing related work as well as the results of this work, and giving an outlook on further ideas and future work on the formal verification of voting schemes.

2. Voting Schemes and Correctness Properties

In the following, we familiarise the reader with some fundamental findings in the general analysis of voting schemes, which set the main motivation for our examinations for verifying correctness of voting schemes as outlined in Section 1.2. These findings justify low expectations regarding the development of one universal voting scheme with a universal notion of correctness and instead strengthen the motivation for research on a variety of voting schemes with respect to different notions of correctness.
In order to set the path for our analysis, this chapter gives a small set of necessary definitions in Section 2.1 and illustrates implications of the *Impossibility Theorem* by the economist, writer, and political theorist Kenneth J. Arrow in Section 2.2. The focus of this work is on the four well-known voting schemes plurality voting, approval voting, instant-runoff voting (IRV) and single transferable vote (STV). We illustrate those in Section 2.3 along with abstract definitions of the procedures. As our selection of voting schemes features great variations in complexity, we also reflect this in our choice of voting scheme properties, each representing a different notion of correctness. Section 2.4 is dedicated to presenting the three different voting scheme properties monotonicity criterion, Condorcet loser criterion as well as the Condorcet criterion, named after the philosopher, mathematician, and early political scientist Marquis de Condorcet. Both Arrow and Condorcet were great pioneers in modern "social choice theory" as they determined the fundamentals in this domain.
We also illustrate how these correctness properties for voting schemes relate to the voting schemes selected for our analysis. These relationships are well-known in the field of *social choice theory* [Fel12b] with the exception of correctness properties for the widely-used STV, which elects multiple candidates to achieve a proportional representation. For this matter, we investigate on adapting properties which are known to hold for IRV to the STV scheme, which is rather known for violating common notions of correctness and argued to have a *quasi-chaotic* behaviour [Nur96][Mil07][FB83].

2.1. A General Definition of Voting Schemes

Social Welfare Functions We start by defining social welfare functions as these are perhaps the most general form to aggregate preferences of individuals into collective preferences [BCE13]. Let us assume a finite set N containing at least two agents or individuals (or voters for *voting schemes*), i.e. $|N| \geq 2$, and a finite universe U containing at least two alternatives (or candidates for *voting schemes*), i.e. $|U| \geq 2$. Furthermore,

we assume every individual $i \in N$ to have preferences over the alternatives in U, which are represented by the *transitive* and *complete* preference relation $\succeq_i$. Completeness can be assumed for a rational social welfare function, since we also allow *indifference*, which represents an indecisiveness of an individual for the according two alternatives, and thus any rational decision can be expressed. However, we do not assign weights or other measurements to individual rankings (which then would be called *ratings*) as firstly the voting schemes examined within this work do not need ratings but only at most rankings, and secondly the following definitions would become considerably less intuitive. Transitivity is needed for reasons of consistency.
For any two alternatives a, b and an individual i, $a \succeq_i b$ denotes that individual i likes alternative a at least as much as alternative b. If we know that $a \succeq_i b$ holds, but $b \succeq_i a$ does not, we can express this by the strict preference relation $a \succ_i b$. Accordingly, $a \sim_i b$ is short for $a \succeq_i b$ and $b \succeq_i a$, which we call *indifference* relation. The set of all preference relations over the universe U is denoted as $\mathcal{R}(U)$. We will also be talking about the set of *preference profiles*, expressing all preferences for every individual $i \in N$, i.e. $\mathcal{R}(U)^n$ (for $n = |N|$). We can hence make the following definition [BCE13]:

Definition 1 (Social Welfare Function) *A* social welfare function *is a function*

$$f : \mathcal{R}(U)^n \to \mathcal{R}(U)$$

The resulting *social* preference relation can then be simply denoted by $\succeq$, as it does not depend on the individual.

Voting Schemes A *social welfare function* hence gives us an aggregated *preference profile.* However, a function used for voting needs to determine a set of elected alternatives. For this matter, we define the set of possible feasible sets $\mathcal{F}(U)$ as the set of all non-empty subsets of U (i.e. $\mathcal{F}(U) = \mathcal{P}(U) \setminus \emptyset$, where $\mathcal{P}$ denotes the power set). By using this set, we hence define the following:

Definition 2 (Social Choice Function) *a* social choice function *is a function*

$$f : \mathcal{R}(U)^n \to \mathcal{F}(U)$$

We will refer to the result of a *social choice function* by the name *choice set.* This is already a usable function for voting, but in order to be a reasonable voting scheme, we need f to additionally fulfil the two symmetry conditions denoted as *anonymity* and *neutrality.* The property *anonymity* states that the outcome of f remains unaffected when any individuals are renamed, i.e. when the individual relations are permuted within the preference profile. Similarly, *neutrality* means the outcome is invariant under the renaming or permutation of any alternatives.
Some voting schemes always yield a unique winner, i.e. $|f(\mathcal{R})| = 1$. These voting schemes are called *resolute* or *single-valued.* However, voting schemes cannot always be *resolute*, as this property generally contradicts *anonymity* and *neutrality.*

2.2. Arrow's Impossibility Theorem and its Implications

Kenneth J. Arrow's results from 1951 [Arr51] form the basic motivation for this work, as they introduce essential challenges and therefore have major implications for all areas of social choice theory. Arrow essentially showed that every "reasonable" voting scheme with at least three distinct alternatives is inconsistent. Additionally to the properties of our definition of a voting scheme, Arrow's definition of a "reasonable" voting scheme fulfils the following three conditions (defined for *social welfare functions*) [BCE13][Dut00][Arr51].

Definition 3 (Pareto-Optimality) *We call a* social welfare function *Pareto-optimal iff*

$$\forall a, b \in U : (\forall i \in N : a \succ_i b) \Rightarrow a \succ b$$

i.e. strict unanimous agreement is reflected in the social preference relation.

Definition 4 (Independence of Irrelevant Alternatives) *We call a* social welfare function *independent of irrelevant alternatives iff (the two preference profiles R and R' correspond to $\succeq$ and $\succeq'$ respectively)*

$$\forall R, R' \; \forall a, b \in U : (\forall i \in N : a \succeq_i b \Leftrightarrow a \succeq'_i b) \Rightarrow (a \succeq b \Leftrightarrow a \succeq' b)$$

i.e. the social preference between any pair of alternatives only depends on the individual preferences restricted to these two alternatives.

Definition 5 (Non-Dictatorship) *We call a* social welfare function *non-dictatorial iff*

$$\neg \exists i \in N : (\forall a, b \in U : a \succ_i b \Rightarrow a \succ b)$$

i.e. there is no individual who can dictate a strict ranking no matter which preferences the other individuals have.

All three conditions are very intuitive and seem clearly desirable for any voting scheme. However, Arrow has shown in his *impossibility theorem*, which he called the *general possibility theorem*, that there does not exist any voting scheme (as defined above) with at least three alternatives (i.e. $|U| \geq 3$) which satisfies *Pareto-optimality*, *independence of irrelevant alternatives* and *non-dictatorship* simultaneously [Arr51]. Thereupon, there has been much work on how to construct "reasonable" voting schemes by relaxing any of these conditions, but as indicated by Arrow's theorem, all of them provide clear insufficiencies (except for cases, where two alternatives suffice). We will now provide two examples as to demonstrate some inconsistencies regarding Arrow's conditions for some simple voting schemes [Sen70].

First Example: Rank Order Let us begin with the so-called "rank order" voting scheme, where a certain number of marks are given to each alternative for being first in anyone's preference ordering, a smaller number for being second in someone's ordering, and so on. Then the total number of marks for each for each alternative is added up, and the one with the highest score wins. Let us take a set of three individuals $\{1, 2, 3\}$ and three alternatives $\{x, y, z\}$. Having a set of three alternatives, we use three marks for the first preference, two marks for the second preference, and one mark for the third preference. Firstly, as renaming does not affect the outcome, i.e. any alternative can win by having the highest score and every vote is given the same weight, it satisfies our definition of a voting scheme. Secondly, there is no one individual being able to dictate the social preference, hence it satisfies *non-dictatorship*. Thirdly, if any alternative receives a higher rank than another alternative by all voters, the aggregated rank will also be higher, hence *Pareto-optimality* is also satisfied. Fourthly, for evaluating the *independence of irrelevant alternatives*, we need a closer look.

Individual 1	Individual 2	Individual 3	Aggregation
x	z	z	x: 7
y	x	x	y: 4
z	y	y	z: 7

Table 2.1.: First Exemplary Ballot Box for Rank Order Example

Let us take a ballot box (Table 2.1), where individual 1 prefers x to y to z and both individuals 2 and 3 prefer z to x to y. By evaluating our voting scheme, x receives 7 marks, y receives 4 marks and z receives 7 marks. Thus, we have a tie between the candidates x and z.

Individual 1	Individual 2	Individual 3	Aggregation
x	z	z	x: 7
z	x	x	y: 3
y	y	y	z: 8

Table 2.2.: Second Exemplary Ballot Box for Rank Order Example

Now, let us take a second ballot box (Table 2.2), where the preferences between candidates x and z stay the same, but individual 1 ranks y on third position. The new scores evaluate to 7 marks for x, 3 marks for y, and 8 marks for z, making z win the election. For both ballot boxes the individual preferences between candidates x and z are the same, but the social preferences between them are different, as the irrelevant alternative y affects their social preference. Hence, the voting scheme violates the *independence of irrelevant alternatives.*

Second Example: Traditional Code In our second example, the social preference is determined by an entirely specified traditional code (i.e. the outcome is predetermined, e.g. by a written constitution, and independent of any individual preference), which determines a given ordering of all alternatives. Even though this does not appear to be a voting scheme, it satisfies *anonymity* and *neutrality* from our definition for voting schemes, since the outcome is invariant under renaming. Furthermore, the conditions *independence of irrelevant alternatives* and *non-dictatorship* are trivially satisfied as the individual preferences simply have no effect on the social preference and consequently no individual is a dictator. However, this voting scheme violates *Pareto-optimality*. Let us for example assume the traditional code determines for the two alternatives x and y the social preference $x \succ y$. This outcome remains, even when every individual strictly prefers y to x. Therefore, we see the importance of *Pareto-optimality* as this voting scheme does also not satisfy common democratic concepts.

2.3. Selected Voting Schemes

A voting scheme, or more precisely a *vote-counting scheme*, is essentially a mechanism that aggregates individual preferences to decide an election. We gave a concise general definition of voting schemes through precise mathematical descriptions in Section 2.1 using a minimal set of common notions. In its essence, vote-counting is a very straightforward task and we can easily think of very simple algorithms which satisfy this definition. However, the definition greatly under-specifies any possible solution as the decision of an election can be done in multiple ways. As such, there exists a tremendously great variety of voting schemes, with each having different justifications and qualities. in Section 2.2, we learned that there is no universal set of qualities, to which researchers generally agree and every voting scheme should fulfil. The major consequence of this finding is that there exist multiple different and often contradicting understandings of "reasonable" voting schemes. In general, the underlying motive is to balance objectives such as establishing legitimacy, encouraging participation, and discouraging factionalisation [LN95]. It is argued that there is not any one right answer as to how these objectives should be traded off against each other, but also that there are certainly many wrong answers. An essential difficulty arises if the voting population lacks a consensus, which constitutes the fundamentally conflicting case among different voting schemes.
Moreover, there are also differences in how a voter can express his will in the election. For this matter, we can distinguish the three general types of *one-vote*, *ranked*, and *rated* voting schemes. In *one-vote* voting schemes, a voter can only vote for one candidate and nothing more. Sometimes, multiple *one-vote* voting schemes are done sequentially one after another, reducing the number of available candidates, usually based on the results in the previous election, for every new election. This is called *runoff* election and the common case is to do a maximum of two consecutive elections, whereby in the second one, only the two candidates with the best results in the first election are available, and in case one candidate gets an absolute majority in the first election, no second one is held. Alternatively in *ranked* or *preferential* voting schemes, each voter ranks the candidates in order of preference. Unranked candidates are usually taken as sharing the last place and variations of this type also allow voters to assign one rank position to more than one candidate. Finally, another type is the *rated* voting scheme, wherein voters give a score to each candidate, allowing even more flexibility than *ranked* voting schemes, as differences in rank can be quantified within this category.
Often, voting schemes are used to elect multiple candidates in one election, e.g. in order to constitute a parliament, and are in the following also called *multi-seat* voting schemes

in distinction to *resolute* single-winner voting-schemes, which elect exactly one candidate. In this case, it becomes not only important which exact candidates get elected, but also how the overall composition of all elected candidates is. This has a particular impact on multi-party systems, wherein members of a particular party usually pursue very similar goals. The simplest version with respect to the overall composition is a *majoritarian* voting scheme, which simply elects a set of candidates, which is supported by a majority of voters, pursuing the *majority* rule. It is usually done by lists, each provided by a different party, and the elected candidates are simply the ones on the list, which is supported by a majority of voters. Another variation of *majoritarion* voting is to let each voter directly vote for any of the available candidates and then elect a number (usually determined by the size of the parliament) of candidates with the best results. Alternatively, many elections do some sort of *proportional* representation, wherein a candidate or party with x percent of the total votes also receives (roughly) x percent of all seats in parliament. Finally, there are also so-called *semi-proportional* voting schemes which combine proportional representation with majority rule, pursuing to find a trade-off between both objectives.
Besides these categories with respect to the expressiveness of the voter's will and how the plurality of voters is represented in a parliament, also the level of complexity of a voting scheme plays a role. This factor impacts transparency of the election process to the voters, but also and more directly the necessary effort of actually performing the election, especially the vote-counting. Some argue that complex voting schemes make it harder for voters to vote strategically, i.e. vote dishonestly in order to better achieve their honest will in the election outcome [LN95]. The complexity of a voting scheme is a significant obstacle when manual vote-counting is involved and many people have to get paid for counting the votes by hand. However, we believe this becomes and is already less of an issue as more and more elections are done electronically and machines do the actual computation of the election outcome. We also argue that the establishment of trust in an election and its outcome becomes harder if the voting scheme is less comprehensible by the voters, which usually holds for more complex voting schemes.
In the following, we illustrate the four different voting schemes plurality voting, approval voting, instant-runoff voting (IRV) and single transferable vote (STV). Plurality voting (Section 2.3.1) is a widely-used *one-vote* voting scheme, approval voting (Section 2.3.2) a restricted *rated* voting scheme, and IRV (Section 2.3.3) and STV (Section 2.3.4) are *ranked* voting schemes. From these selected voting schemes, STV is the only *multi-seat* voting scheme, more specifically a *proportional* one, with the others all being *resolute* single-winner voting schemes. We use some common constants for all voting schemes, whereof V always denotes the number of voters, C the number of candidates, and S the number of seats in case of a *multi-seat* voting scheme.

2.3.1. Plurality Voting

Plurality voting is a simple and perhaps the most fundamental voting scheme. More specifically, we refer to *first-past-the-post* (FPTP) plurality voting, where each voter can vote for exactly one candidate and only one candidate gets elected. Another variation of *plurality voting* is *block voting*, where a certain number n of candidates gets elected and each voter can cast n votes. In the case of *first-past-the-post plurality voting* (from now on simply referred to as *plurality voting*), each voter has one vote. These votes are then tallied and the candidate, who gets more votes than any other candidate, is elected. This is not to be confused with *majority voting* or *majority rule*, where the elect needs an absolute majority of all votes. Contrarily, *plurality voting* can elect a candidate with less than 50% of the votes, if the candidate reaches a relative majority among all votes. As *plurality voting* still leaves room for ties, real-world voting systems vary in the way in which they deal therewith. For the matter of this thesis, we do not elect any candidate in the event of a tie. The precise scheme used within this work can be seen in Algorithm 2.1. Therein, the notion $\#\{\dots\}$ specifies the cardinality of the set described between the two braces, i.e. the number of elements contained in this set.

Algorithm 2.1 Plurality Voting

```
Input: $v$: list of $|V|$ votes with $v_i \in C$, $C$: set of candidates, $V$: set of voters
Output: $elect$: the elected candidate ($elect \in C$) or a tie ($elect = \perp$)
 1: function VOTING($v$, $C$, $V$)
 2:     $max \leftarrow 0$, $voteCount \leftarrow 0$, $elect \leftarrow \perp$
 3:     for all $k \in C$ do
 4:         $voteCount \leftarrow \#\{i \in V \mid v_i = k\}$
 5:         if $max < voteCount$ then
 6:             $max \leftarrow voteCount$
 7:             $elect \leftarrow k$
 8:         else if $max = voteCount$ then
 9:             $elect \leftarrow \perp$
10:         end if
11:     end for
12:     return $elect$
13: end function
```

One noteworthy attribute of plurality voting is an observation expressed in *Duverger's law* [Dre72], according to which plurality voting within single-member districts tends to favour two-party systems. Single-member district voting contrasts proportional representation and refers to an election where either only a single candidate gets elected or bloc voting is used, i.e. one of multiple predefined candidate groups gets elected. Sometimes parliaments are constructed via multiple of these single-member elections, each within a predefined district, and thus denoted as single-member district voting. This scheme makes it harder for smaller parties to become successful, making the outcome less surprising and more stable. The idea behind this hypothesis is that voters do not want their vote to be insignificant and hence tend to vote for parties or candidates who were successful in past elections.

Referring to Arrow's *impossibility theorem*, the plurality voting scheme complies with *non-dictatorship* and *Pareto-optimality*, but violates the *independence of irrelevant alternatives.*

Example Let us assume `A`, `B` and `C` be three candidates and {`B`, `C`, `A`, `B`, `A`, `C`, `B`, `B`, `B`, `A`} a set of votes.
We then tally the votes and count 3 votes for candidate `A`, 4 votes for candidate `B` and 3 votes for candidate `C`. Hence, candidate `B` wins the election. Since each voter can only submit a vote for only one candidate, there can still be another candidate who is preferred over candidate `B` by a majority of voters, but which is not considered by plurality voting.

3 Votes	4 Votes	3 Votes
A	B	C
C	C	B
B	A	A

Table 2.3.: Exemplary Aggregated Ballot Box for Plurality Voting

If we assume for example (Table 2.3) that all the voters voting for candidate `A` have candidate `C` as their second choice and candidate `B` only as their last choice, there would be a majority who prefers candidate `C` over candidate `B` and this opinion of the majority would not be captured by this election result.

2.3.2. Approval Voting

The vote counting process of *approval voting* is very similar to the one for *plurality voting*. However, it differs in the amount of votes each voter is allowed to cast. *Approval voting* allows each voter as many votes as there are candidates running for election. The voter can vote for as many candidates as he or she likes. This is usually done by completing a ballot paper listing all candidates and one box for each of them. Completing the ballot paper then consists in ticking all the boxes for candidates the voter approves of. Hence, the voters can express their will more accurately than for *plurality voting*. Subsequently – just as for *plurality voting* – all votes are tallied and the candidate who gets more votes than any other candidate is elected. Here again, we do not deal with ties, but instead do not elect any candidate. The precise scheme used within this work can be seen in Algorithm 2.2.
Referring to Arrow's *impossibility theorem*, this voting scheme complies with *non-dictatorship* and *Pareto-optimality*, as well as with the *independence of irrelevant alternatives*.

Example Let us assume `A`, `B` and `C` be three candidates and there are ten voters. Four voters approve of candidates `A` and `B`, three voters approve of candidates `B` and `C`, two voters approve only of candidate `A` and one voter approves only of candidate `C` (Table 2.4).

4 Votes	3 Votes	2 Votes	1 Vote
A, B	B, C	A	C

Table 2.4.: Exemplary Aggregated Ballot Box for Approval Voting

Algorithm 2.2 Approval Voting

```
Input: $v$: list of $|V|$ vote lists of length $|C|$ with $v_{i,j} \in \{0,1\}$ (1 for approval), $C$: set of
    candidates, $V$: set of voters
Output: $elect$: the elected candidate ($elect \in C$) or a tie ($elect = \perp$)
 1: function VOTING($v$, $C$, $V$)
 2:     $max \leftarrow 0$, $voteCount \leftarrow 0$, $elect \leftarrow \perp$
 3:     for all $k \in C$ do
 4:         $voteCount \leftarrow \#\{i \in V \mid v_{i,k} = 1\}$
 5:         if $max < voteCount$ then
 6:             $max \leftarrow voteCount$
 7:             $elect \leftarrow k$
 8:         else if $max = voteCount$ then
 9:             $elect \leftarrow \perp$
10:         end if
11:     end for
12:     return $elect$
13: end function
```

We then tally all votes and count 7 votes for candidate `B`, 6 votes for candidate `A` and 5 votes for candidate `C`. Hence, candidate `B` wins the election. In contrast to plurality voting, in approval voting one voter can vote for more than one candidate. If for example, candidates `A` and `B` represent similar views, this does not diminish the result of either of them as voters can simply vote for both instead of having to choose between them (*independence of irrelevant alternatives*).

2.3.3. Instant-Runoff Voting

Whereas *plurality voting* and *approval voting* each consist of one round of voting, *instant-runoff voting* can be perceived as performing multiple voting rounds. Although there is only one vote casting process, multiple voting rounds are simulated by the voting scheme. This is done by casting preferential votes, i.e. each voter can cast a ranked vote expressed through an ordered list of candidates. For this matter, an empty ballot paper may look much like one for *approval voting*, but instead of ticking or not ticking the boxes, a voter expresses his or her will by writing numbers from 1 to C (where C denotes the amount of eligible candidates) in the boxes, where 1 denotes the voter's first preference, 2 the voter's second preference, and so on up to the voter's least preferred candidate.
Subsequently, this results in a complex vote counting process. The complete scheme used within this thesis can be seen in Algorithm 2.4. We start by tallying only the first preferences. If there is already one candidate with an absolute majority, this candidate is elected (Line 6). If this is not the case, the candidate with the least amount of first preferences is eliminated, the according votes are distributed to the remaining candidates according to their second preferences, and their preferences are recalculated (Lines 10 to 13). We iteratively repeat this process until a winner is found. For the given ballot

box used in the voting scheme, we assume there are no empty votes, all preferences of a vote either contain a valid candidate or are empty, in any preference list each candidate can appear only once, and if any preference rank in a vote is empty, then all subsequent ranks in this vote are also empty, and hence there are no gaps in preference lists.

Breaking Ties (Two Variations) For *instant-runoff voting*, ties can occur at various points in the vote counting process, which for our general version of *instant-runoff voting* is abstracted by the function *chooseCandidate*, which is in charge of picking one candidate out of the set of candidates W (Algorithm 2.4). Directly before the call of this function, we compute the set of candidates, who are weakest at the current state in the algorithm, i.e. currently have the least amount of first preferences compared to all other candidates still in the race. Hence, a tie occurs when the set W has more than one element, which means that there is no unique weakest candidate, but a set of equally weakest candidates. In this work, we abstract the tie-breaking mechanism by the function *chooseCandidate*, but in real-world examples, there are different approaches to deal with ties. Some consider the previous results of tied candidates and eliminate the one with a previously inferior result. Others consider their second or last preferences, whereas some even choose randomly. Within this work, we examine two variations. Firstly, we only eliminate one of the tied candidates, but pursue all possible choices simultaneously (from now on denoted as *exhaustive instant-runoff voting*). The precise scheme for this version can be seen in Algorithm 2.4, using the delete subroutine seen in Algorithm 2.3. Technically, we only pursue one randomly chosen path in this description when choosing a candidate for deletion. The pursuit of all possible paths is expressed by the non-concreteness of the function *chooseCandidate* (i.e. the choice of a candidate from the set W is under-specified), thereby creating multiple instances of the voting procedure every time we have multiple equally weakest candidates at this stage. For the real world scenario, this is the variation being used, but only one choice is pursued. As this choice is, however, generally arbitrary, we want to abstract away from the particular choice made by taking each generally possible path.
Secondly, we consider a version which eliminates all tied candidates with the least amount of first preferences at the same time (from now on denoted as *deterministic instant-runoff voting*). In this case, the function *chooseCandidate* is not needed and instead we replace Line 11 to Line 13 in Algorithm 2.4 by the three lines of code seen in Algorithm 2.5 and also using the same subroutine as in Algorithm 2.3, i.e. we simply iterate over the set of equally weakest candidates and delete each of them from the ballot box. In the rare case where all candidates are eliminated (which can happen, if we eliminate multiple candidates at once), we simply do not elect any candidate.
Furthermore, we make some assumptions about the votes. Firstly, we assume every voter assigns a first preference. Secondly, if a voter does not assign a certain preference (which is not the first one), all successive preferences are unassigned as well. Thirdly and lastly, we assume every voter does only rank each candidate at most once in his or her preference list.

Algorithm 2.3 Delete Subroutine for Preferential Voting Schemes

Input: *votes*: list of $|V|$ vote lists of length $|C|$ with $votes_{i,j} \in C$ or $votes_{i,j} = \perp$ ($votes_{i,1}$ for the candidate which voter i prefers most, $\perp$ means there is no j^{th} preference for voter i), *candidate*: candidate to be deleted from *votes*, C: set of candidates, V: set of voters

Output: *votes*: list of $|V|$ vote lists of length $|C - 1|$ without votes for *candidate*, otherwise equal to the input parameter

```
function DELETE($votes$, $candidate$, $C$, $V$)
    for all $(i, k) \in V \times C$ do
        if $votes_{i,k} = candidate$ then
            for $p \leftarrow k$ to $|C| - 1$ do
                $votes_{i,p} \leftarrow votes_{i,p+1}$
            end for
            $votes_{i,|C|} \leftarrow \perp$
        end if
    end for
    return $votes$
end function
```

Algorithm 2.4 Instant-Runoff Voting

Input: v: list of $|V|$ vote lists of length $|C|$ with $v_{i,j} \in C$ or $v_{i,j} = \perp$ ($v_{i,1}$ for the candidate which voter i prefers most, $\perp$ means there is no j^{th} preference for voter i), C: set of candidates, V: set of voters

Output: *elect*: number denoting the elected candidate ($elect \in C$)

```
function VOTING($v$, $C$, $V$)
    $quota \leftarrow \lfloor \frac{V}{2} \rfloor$, $CC \leftarrow C$, $elect \leftarrow \perp$
    while $elect \notin C \wedge CC \neq \emptyset$ do
        for all $k \in C$ do                                        }
            if $quota < \#\{i \in V \mid v_{i,1} = k\}$ then        }  Elect candidate
                $elect \leftarrow k$                                }  with more votes
            end if                                                  }  than the quota.
        end for                                                     }
        if $elect \notin C$ then                                                              }  Eliminate cand.
            $W \leftarrow \{k \in C | \#\{i \in V \mid v_{i,1} = k\} = \min_{c \in C} \#\{i \in V \mid v_{i,1} = c\}\}$   }  with fewest 1st
            $weakest \leftarrow$ CHOOSECANDIDATE($W$)                                         }  preferences,
            $v \leftarrow$ DELETE($v$, $weakest$, $CC$, $V$)                                  }  redistribute
            $CC \leftarrow CC \setminus weakest$                                              }  according
        end if                                                                                }  to 2nd pref.
    end while
    return $elect$
end function
```

Algorithm 2.5 Differences for Deterministic Instant-Runoff Voting

```
11: for all $w \in W$ do
12:     $v \leftarrow$ DELETE$(v, w)$
13:     $CC \leftarrow CC \setminus w$
14: end for
```

Referring to Arrow's *impossibility theorem*, instant-runoff voting – just as *plurality voting* – complies with *non-dictatorship* and *Pareto-optimality*, but violates the *independence of irrelevant alternatives*.

Example Let us assume, as depicted in Table 2.5, A, B and C be three candidates and there are ten voters. Four voters rank candidate C first, candidate A second and candidate B third. Three voters rank candidate B first, candidate A second and candidate C third. Two voters rank candidate A first, candidate B second and candidate C third. And one voter ranks candidate A first, candidate C second and candidate B third.

4 Votes	3 Votes	2 Votes	1 Votes	Aggregation 1	Aggregation 2
C	B	A	A	B, C: 5	A, C: 6
A	A	B	C	C, B: 5	C, A: 4
B	C	C	B		

Table 2.5.: Exemplary Aggregated Ballot Box for Instant-Runoff Voting

In instant-runoff voting, we start by looking only at all first choices. There, we count 4 votes for candidate C, 3 votes for candidate A and 3 votes for candidate B. Hence, there is no candidate with an absolute majority in all first preferences and we must delete one candidate. However, as there are two candidates who both have the lowest amount of first preferences, we would need some kind of tie-breaker in practice. Within this work, we explore both possibilities in this case.
Therefore, we start by deleting all votes for candidate A, as this candidate has the lowest amount of first preference votes. This results in the following modified votes (Aggregation 1 in Table 2.5). Five voters rank candidate C first and candidate B second. Five voters rank candidate B first and candidate C second. Again, we get two equally low ranked candidates and explore both possibilities. As after deleting any candidate there is only one candidate left, either candidate B or candidate C wins.
We now explore the case, in which we deleted all votes for candidate B in the first place. This results in the following modified votes (Aggregation 2 in Table 2.5). Four voters rank candidate C first and candidate A second. Six voters rank candidate A second and candidate C third. In this case, candidate A has the absolute majority of all remaining first preference votes and hence wins the election.
Hence, for this set of votes, there is an instance of instance-runoff voting for any candidate to win the election. Even though, deleting all votes for candidate B in the first round seems to lead to the clearest result and in practice many used tie-breaking rules suggest

this, we could also argue for candidate `C` to win the election in this case as this candidate also has a "clear" majority.

2.3.4. Single Transferable Vote

Single transferable vote (STV) is similar to *instant-runoff voting*, but instead elects a number of candidates, e.g. fills a number of seats in parliament. The precise scheme used within this work can be seen in Algorithm 2.6, again using the delete subroutine seen in Algorithm 2.3. This is basically done by repeating *instant-runoff voting* until all seats are filled ($e < |S|$) or there are as many seats left as remaining candidates, in which case the remaining seats are filled up automatically. After one candidate gets elected, he or she gets eliminated and only the surplus votes (i.e. the ones exceeding the quota) get transferred to the next preferences. This is a means to reuse votes, which an elected candidate did not need to get elected, and thus make every vote count and avoid strategic voter behaviour. However, there exist various different versions of STV. They differ, e.g., in the quota needed to get elected (which can be lower than 50%), the way ties are dealt with or the possible resurrection of already eliminated candidates when they receive transferred votes in future rounds. There are even further variations, but these are argued to not be part of the STV family anymore [BGS13].
This voting scheme also complies with *non-dictatorship* and *Pareto-optimality*, but violates the *independence of irrelevant alternatives*.

2.4. Selected Voting Scheme Properties

We argued that even though there is no universal notion of correctness for voting schemes, having a justified and formal concept of correctness is indispensable in order to establish trust and credibility in the election process as a whole and the vote-counting in particular. There is not any one right answer for a sensible trade-off for this matter, but finding a good, unambiguous and comprehensible one is still crucial for the election and its result to be lawfully justified and constitutional. As any voting scheme has its flaws, these should be well-understood in order to avoid surprising effects and have a suitable election procedure. This becomes an especially essential necessity for modern voting schemes, which grow in variety as well as complexity. in order for increasingly complex voting schemes to still be graspable enough to be trusted by all participants, we believe in the need for more declarative and traceable specifications of voting schemes such as precise and formal properties. They have the potential to enhance comparability, verifiability and justification, and thereby minimise any surprising behaviour and outcomes.
Research in the area of *social choice theory* has produced an overview of a large variety of voting scheme properties and the voting schemes which satisfy or violate those [Fel12b]. However, it becomes evident for the large number of violated properties for each voting scheme that we either need some kind of measure in order to determine how badly it is violated, or adapted tailor-made properties in order to instead formalise relaxed properties which are satisfied. The former has been investigated on and shown to be a difficult

Algorithm 2.6 Single Transferable Vote

Input: v: list of $|V|$ vote lists of length $|C|$ with $v_{i,j} \in C$ or $v_{i,j} = \perp$ ($v_{i,1}$ for the candidate which voter i prefers most, $\perp$ means there is no j^{th} preference for voter i), C: set of candidates, V: set of voters, S: set of seats

Output: *elect*: list of length $|S|$ with the elected candidates or $\perp$ (in case $|C| < |S|$)

```
function VOTING($v$, $C$, $V$, $S$)
    $quota \leftarrow \lfloor \frac{V}{2} \rfloor$, $CC \leftarrow C$, $e \leftarrow 0$, $res \leftarrow \perp$, $elect \leftarrow \{\perp, \ldots, \perp\}$
    while $CC \neq \emptyset \wedge e < |S| \wedge (|S| - e) < |CC|$ do
        for all $k \in C$ do
            if $quota < \#\{i \in V \mid v_{i,1} = k\}$ then        Elect candidate
                $res \leftarrow k$                                  with more votes
            end if                                                  than the *quota*.
        end for
        if $res \in C$ then
            $elect_e \leftarrow res$, $e \leftarrow e + 1$, $T \leftarrow \emptyset$
            for all $i \in V$: $v_{i,1} = res \wedge |T| < quota$ do      Elected candidate
                for all $k \in C$ do                                      gets eliminated
                    $v_{i,k} \leftarrow \perp$                            and surplus
                end for                                                   votes ($|T|$) get
                $T \leftarrow T \cup i$                                   transferred to
            end for                                                       next preferences.
            $v \leftarrow$ DELETE($v$, $res$)
            $CC \leftarrow CC \setminus res$, $res \leftarrow \perp$
        else                                                                              Eliminate cand.
            $W \leftarrow \{k \in C \mid \#\{i \in V \mid v_{i,1} = k\} = \min_{c \in C} \#\{i \in V \mid v_{i,1} = c\}\}$   with fewest $1^{st}$
            $weakest \leftarrow$ CHOOSECANDIDATE($W$)                                     preferences,
            $v \leftarrow$ DELETE($v$, $weakest$)                                         redistribute
            $CC \leftarrow CC \setminus weakest$                                          according
        end if                                                                            to $2^{nd}$ pref.
    end while
    if $e < |S|$ then
        for all $i \in S \setminus elect$: $CC \neq \emptyset$ do
            $res \leftarrow \perp$
            for all $k, j$: $k \in C \wedge j \in V \wedge res \notin C$ do     Fill remaining
                if $v_{j,1} = k$ then                                          seats with
                    $res \leftarrow k$                                         remaining
                end if                                                         candidates who
            end for                                                            received at
            $elect_i \leftarrow res$, $e \leftarrow e + 1$                      least one vote.
            $v \leftarrow$ DELETE($v$, $res$)
            $CC \leftarrow CC \setminus res$
        end for
    end if
    return $elect$
end function
```

approach [Nur12]. We argue for the latter one and propose a development cycle for voting schemes and suitable properties based on automatic reasoning with formal methods. In the course of this work, we examine the feasibility of our proposition on the basis of three well-known voting scheme properties; the monotonicity criterion (Section 2.4.1), the Condorcet criterion and the Condorcet loser criterion (both in Section 2.4.2). In the following, we illustrate these three criteria and give examples with respect to our selection of voting schemes.

2.4.1. Monotonicity Criterion

Monotonicity is perhaps one of the most natural properties one would suspect a voting scheme to have. Intuitively, voting for a preferred candidate x should not be worse for x than not voting for x. The criterion was originally posited as a desirable property of social welfare functions as follows [Arr50]:

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

Applied to voting schemes, this translates to the *mono-raise* criterion [Woo97]:

> "A candidate x should not be harmed if x is raised on some ballots without changing the orders of the other candidates."

In order to analyse voting schemes for the monotonicity criterion, we hence need to talk about two comparable runs of the same voting scheme. As for voting schemes where votes only consist in the indication of one candidate, raising candidate x means a vote for a candidate other than x is changed to a vote for candidate x instead. For preferential voting schemes with lists of ranked votes, raising candidate x means to switch the vote for candidate x with a higher ranked one of the same voter. In case the voter did not rank candidate x, we choose any one of the ranked candidates and proceed as for single vote voting schemes.
The criterion then states that the outcome for the "raised" candidate x in the second run is not worse than x's outcome in the first run. This hence means that if the raised candidate x has won the first run, x also wins the second run. If we argue inductively over the sets of voters and candidates, we get an equivalent statement with only one changed vote. By also defining the preference relation between different sets of individuals and thus also different ballot boxes (expressed by indexing the candidates with the according ballot box) coherently, we then can express monotonicity in the following manner:

Definition 6 (Monotonicity Criterion) *For a set of candidates C, a set of voters V and a set of ballot boxes B, we call a* voting scheme *monotone iff*

$$\forall c \in C, v \in V, b, b' \in B : (c_{b'} \succ_v c_b \wedge \forall i \in V : i \neq v \Rightarrow \forall a \in C : a_{b'} \sim_i a_b) \Rightarrow c_{b'} \succ c_b$$

The voting schemes *plurality voting* and *approval voting* comply with the *monotonicity criterion*, whereas *instant-runoff voting* and *single transferable vote* violate the condition.

Example: Instant-Runoff Voting Let us illustrate a voting scheme with non-monotone behaviour by means of an exemplary election with the instant-runoff voting scheme (Section 2.3.3), having 17 voters and the three candidates `A`, `B` and `C` [Gal13]. Hence, a candidate needs a quota of 9 votes in order to win the election. There are six voters, who rank candidate `A` first, candidate `B` second and candidate `C` third. Another six voters rank candidate `B` first, candidate `C` second and candidate `A` third. The remaining five voters rank candidate `C` first, candidate `A` second and candidate `B` last.

6 Votes	6 Votes	5 Votes
A	B	C
B	C	A
C	A	B

Table 2.6.: First Exemplary Aggregated Ballot Box for IRV

With this ballot box as shown in Table 2.6 in an aggregated manner, the amounts of first preferences are 6, 6 and 5 for candidates `A`, `B` and `C` respectively. Therefore, no candidate reaches the necessary majority of 9 votes in the first round and the weakest candidate gets eliminated, who is candidate `C`. The resulting ballot box with only candidates `A` and `B` has then 11 votes for candidate `A` and 6 votes for candidate `B`, making candidate `A` the winner of the election.
Now, let us consider two voters had changed their mind (e.g. as they visited a convincing speech by candidate `A` one day before the election). They switch to candidate `A` as their first preference, leaving the order of the remaining two preferences as in the first ballot box. More precisely, now two ballots in the first ballot box of the form `[B, C, A]` are changed to `[A, B, C]`, leaving us with an aggregated ballot box as shown in Table 2.7.

8 Votes	5 Votes	4 Votes
A	C	B
B	A	C
C	B	A

Table 2.7.: Second Exemplary Aggregated Ballot Box for IRV

Hence, the first preferences of the ballot box after the change compute to 8, 5 and 4 for candidates `A`, `C` and `B` respectively. Also in this election, no candidate reaches the necessary majority of 9 votes in the first round, but this time candidate `B` is the weakest candidate and gets eliminated. As the voters with candidate `B` as their first preference, ranked candidate `C` second, the resulting ballot box with candidates `A` and `C` contains 9 votes for candidate `C` and 8 votes for candidate `A`. The new winner is therefore candidate `C`, although his or her ranks are worse than in the first ballot box in Table 2.6, especially compared to candidate `A`, who achieved an increased voter support, but a worse election result, which thus constitutes a non-monotone behaviour.

2.4.2. Condorcet Winner and Loser Criteria

The two criteria illustrated in the following are both based on findings of the philosopher, mathematician, and early political scientist Marquis de Condorcet [Con85]. Therein he argues the candidate with the greatest amount of head-to-head competitions or pairwise majority comparisons, called the *Condorcet winner*, to be a sensible winner of an election. He also illustrates situations where such a candidate does not exist due to a preference cycle, which is nowadays commonly known as the *Condorcet paradox.* His investigations are based on *ranked* or *preferential* ballots.

Condorcet Winner Criterion The *Condorcet winner* criterion is based on pairwise simple majority comparisons. This means we compare for each candidate the (ranked) lists of votes of each voter. Therein, we compare all voted candidates mutually by determining the candidate who is ranked higher. When we do this for all lists of votes, we determine the candidate who wins all of these comparisons, i.e. defeats every other candidate. This candidate is then called the *Condorcet winner.* However, a *Condorcet winner* does not always exist as there can occur a cycle of these mutual comparisons, also called *Condorcet's paradox.* In this case, the ranking of pairwise simple majority comparisons provides no clear winner as the collective ranking forms a cycle. An example for the three candidates A, B and C with three voters can be seen in Table 2.8. Therein, every candidate is preferred over every other candidate by a margin of two to one and a Condorcet winner cannot be determined.

Voter 1	Voter 2	Voter 3
A	B	C
B	C	A
C	A	B

Table 2.8.: Example for Condorcet's Paradox

A voting scheme complying with the *Condorcet winner criterion*, also simply called *Condorcet criterion*, always elects the *Condorcet winner* when one exists. Hence, we can express this criterion in the following manner:

Definition 7 (Condorcet Winner Criterion) *For a set of candidates C and a set of voters V, we say a* voting scheme *satisfies the Condorcet criterion iff*

$$\begin{aligned}\forall c \in C : (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid k \succ_v c\} < \#\{v \in V \mid c \succ_v k\}) \\ \Rightarrow \forall c' \in C : c' = c \vee c \succ c'\end{aligned}$$

The voting schemes *plurality voting*, *approval voting*, *instant-runoff voting* and *single transferable vote* all violate the *Condorcet criterion.*

Condorcet Loser Criterion Conversely to the notion of the *Condorcet winner*, there is for of the *Condorcet loser*, which is also based on pairwise simple majority comparisons.

In this case, we determine the candidate who is ranked lower instead of higher against every other candidate. We then call this candidate the *Condorcet loser*. Similarly to the *Condorcet winner*, a *Condorcet loser* does not always exist.
A voting scheme complying with the *Condorcet loser criterion* thus never allows a *Condorcet loser* to get elected. Hence, we can express this criterion in the following manner:

Definition 8 (Condorcet Loser Criterion) *For a set of candidates C and a set of voters V, we say a* voting scheme *satisfies the Condorcet loser criterion iff*

$$\forall c \in C : (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid c \succ_v k\} < \#\{v \in V \mid k \succ_v c\}) \\ \Rightarrow \neg\exists c' \in C : c \succ c'$$

The voting schemes *instant-runoff voting* and *single transferable vote* comply with the *Condorcet loser criterion*, whereas *plurality voting* and *approval voting* violate the condition.

First Example: Plurality Voting In the case of plurality voting, each voter can only vote for exactly one candidate, which already indicates that the voting scheme does not capture any individual pairwise relations except for the ones between the chosen candidate and any other candidates. However, a voter who cast a vote for candidate `C` can also have more distinctive opinions as e.g. preferring candidate `A` to candidate `B` or even being undecided between candidates `A` and `C`. Let us assume an exemplary order of preferences for 7 voters and the three candidates `A`, `B` and `C` as shown in Table 2.9, where three voters prefer candidate `A` to candidate `C`, who they prefer to candidate `B`. Two voters prefer candidate `B` to candidate `C`, who they prefer to candidate `A`, and the remaining two candidates prefer candidate `C` to candidate `B`, who they prefer to candidate `A`.

3 Votes	2 Votes	2 Votes
A	B	C
C	C	B
B	A	A

Table 2.9.: Exemplary Preference Order

In an election with plurality voting (Section 2.3.1), if we assume honest voting behaviour, we get three votes for candidate `A`, two votes for candidate `B` and another two votes for candidate `C`, consequently making candidate `A` the winner of the election. However, four voters prefer both candidates `B` and `C` to candidate `A`, compared to only three candidates preferring candidate `A`. When we look at all pairwise head-to-head competitions, candidate `B` defeats candidate `A` with four versus three wins, candidate `C` defeats candidate `A` also with four versus three wins, and candidate `C` beats candidate `B` with five versus two wins. This means, the Condorcet winner is candidate `C` and `A` is even the Condorcet loser for

this preference order. As a consequence, plurality voting violates both the Condorcet winner and the Condorcet loser criterion.
Alternatively, we could revoke our assumption of an honest voting behaviour and suspect supporters of candidates, for who they suspect to not stand a chance for winning the election, to instead vote for a candidate with higher chances. However, even then the given preference order can create problematic outcomes with plurality voting. As both the supporters of candidate `B` and the supporters of candidate `C` are equally averse to support candidate `A`, at least two alternative voting behaviours seem likely.
If candidates `B` and `C` both share similar views, their supporters might all vote for candidate `B`. This would result in three votes for candidate `A` and four votes for candidate `B`, making candidate `B` win the election. In this case, we did not end up electing the Condorcet loser, but the Condorcet winner was not elected either. The other case, where supporters of candidates `B` and `C` unitedly vote for candidate `C` is just as likely as the latter scenario. But then, candidate `C` would win the election and the result would be compatible with the Condorcet winner criterion. Nonetheless, with the assumption of strategic voting, the exact voting behaviour is firstly hard to estimate with still a high chance of not electing the Condorcet winner or even making the Condorcet loser win the election, and we secondly argue strategic voting to be an undesirable behaviour in a democracy.

Second Example: Approval Voting For an election with approval voting (Section 2.3.2), each voter can express his or her approval or disapproval for every candidate separately and no order between the approved candidates is given. As a consequence, the outcome is a more consensual one than for plurality voting. Let us assume the same preference order as for the previous example in Table 2.9 with 9 voters and the three candidates `A`, `B` and `C`. In the case of approval voting, how to deduce a ballot box from a given preference order is not obvious as approval voting rather relies on *rated* ballots than on *ranked* ones. If we simply assume that every voter approves of two candidates, accordingly only disapproving of one candidate, we get get three votes for candidates `A` and `C`, and four votes for candidates `B` and `C`, making candidate `C` the winner of the election. This outcome is compatible with both the Condorcet loser and the Condorcet winner criterion as candidate `C` is the Condorcet winner. Indeed empirical data justifies that the approval voting scheme is very likely to elect the Condorcet winner in practice [BF88][RG98].
There are however at least theoretical counterexamples, where approval voting does not comply with the Condorcet criterion. When we revoke our assumption that every voter approves of exactly two candidates, the situation changes. The given preference order (Table 2.9) always lists candidate `A` as first or third, but never as second choice. Therefore, candidate `A` may hold views strongly opposing a large set of `A`'s or `B`'s views. In this case it is unlikely that a voter favouring candidate `A` additionally approves of either candidate `B` or `C`. Further pursuing this assumption, we could get a ballot box with three votes for candidate `A`, and four votes for candidate `B` and `C`. This is a tie situation between candidates `B` and `C` and thus only complying with the Condorcet loser criterion as candidate `A` does not get elected. Any change in the threshold for a supporter of either

candidate `B` or candidate `C` would determine the election result with either candidate `B` or candidate `C`, the Condorcet winner, as the winner.
A situation identical to the example for plurality voting with the same preference order arises, when every voter only elects one candidate. In this – arguably unlikely – case, approval voting violates both the Condorcet loser and the Condorcet winner criterion. We can conclude our example with the insight that approval voting can violate both criteria, but is in practice unlikely to violate the Condorcet loser criterion.

Third Example: Instant-Runoff Voting In contrast to plurality voting and approval voting, instant-runoff voting requires preferential ballots and is hence easier for examining both the Condorcet loser and the Condorcet winner criterion. As mentioned above, IRV is well-known to comply with the Condorcet loser criterion, but violates the Condorcet winner criterion [Fel12b]. Let us for this matter consider an exemplary ballot box very similar to Table 2.9, but with two more voters in order to avoid ties. We hence have a ballot box with 9 voters, the three candidates `A`, `B` and `C` as shown in Table 2.10, and therefore a candidate needs a majority of five votes to get elected.

4 Votes	3 Votes	2 Votes
A	B	C
C	C	B
B	A	A

Table 2.10.: Exemplary Preference Order for IRV

When determining the Condorcet winner, we compare all head-to-head competitions and observe that candidate `B` defeats candidate `A` with five versus four wins, candidate `C` also defeats candidate `A` with five versus four wins, and candidate `C` beats candidate `B` with six versus three wins. The Condorcet winner accordingly is candidate `C` and candidate `A` the Condorcet loser. In the event of an IRV election, the aggregated first preferences are four votes for candidate `A`, three votes for candidate `B`, and two votes for candidate `C`. Consequently, no candidate reaches the quota of five first preference votes and candidate `C` as the weakest candidate gets eliminated in the first round. After the elimination, we have five votes for candidate `B` and four votes for candidate `A`, effectively making candidate `B` the winner under an IRV election. Candidate `B` is neither the Condorcet loser nor the Condorcet winner and as such this outcome violates the Condorcet criterion.

3. Techniques and Tools for Verification

Throughout the previous chapter, we presented various different voting schemes as well as three different criteria to specify different aspects of correctness for a voting scheme. Voting schemes are usually designed with at least some notion of correctness in mind, sometimes the designers even attempt to comply with multiple correctness criteria. Whereas it is desirable to cope with multiple notions of correctness by satisfying various correctness criteria, we have demonstrated in Section 2.2 that this is impossible to achieve for general correctness properties. This finding makes the development of voting schemes with a consistent notion of correctness, where the selected correctness criteria are compatible, a difficult undertaking and provides several pitfalls for designers of voting schemes. Hence, we believe it is important to have a good understanding of the properties a voting scheme has, as these can lead to unexpected and often undesirable results.
In order to gain reliability on this issue, we perform an analysis of voting schemes with respect to formal notions of correctness, providing precise results with the goal of a non-regressive and thus fail-safe design process for voting schemes. For this matter, this chapter illustrates two major formal verification techniques as well as two formal software tools, of which each pursues one of these techniques. Thereby, we have a design process in mind, wherein each technique plays a different role and complements the respective other technique. The "light-weight" verification technique *software bounded model checking* (SBMC) gives early feedback for small inputs and thus leads to a rapid prototype for a desired voting scheme, where invalid voting schemes or properties can usually be disproved by producing concrete counterexamples. In the further process, the "medium-weight" verification technique *auto-active deductive program verification* can produce a more reliable verification for a larger or even unlimited set of inputs, but cannot be called rapid as it requires significantly more user interaction.

3.1. Selected Verification Techniques

Within this work, we apply two different verification techniques, which we argue to be *light-weight* and *medium-weight* techniques. Using this notion, we distinguish *light-weight* and *medium-weight* from *heavy-weight* techniques. This distinction is mainly made with respect to the necessary amount of specification and the necessary verification work using this specification. In terms of the verification work, we understand *light-weight* and *medium-weight* techniques to be mostly automatic and or at least *auto-active*. *Auto-active*

verification has to be considered in-between fully automatic analysis and interactive proofs [LM10]. It targets a high degree of automation and supports programmer-friendly user interaction through source code annotations with additional proof hints, usually relying on abstract interpretation. The fully automatic technique of *software bounded model checking* (SBMC) however requires only a small amount of user interaction, but does only provide verification for a small scope of inputs. SBMC unrolls the control flow graph of the program for a fixed number of steps (usually instructions) and checks whether an error location (leading to a counterexample) can be reached within this number of steps [Bie+99]. A major justification for the significance of this technique comes from the small scope hypothesis, which argues that a high proportion of bugs can be found for inputs within some small scope [ADK03]. SBMC is usually being used to find low level bugs which concern the technical functionality of the program, but can also be enhanced with simple source code annotations in order to specify more global properties, and bounded proof obligations are discharged automatically.
Both techniques SBMC and *auto-active* verification involve adding the properties to be checked as pre- and postcondition annotations to the actual program code. However, fully interactive verification, which we classify as *heavy-weight* technique, involves encoding the implementation as well as the specification into a higher order logic, commonly of an LCF-style theorem prover [Mil72]. Then, we use the theorem prover to prove that the implementation adheres to the specification, which is usually done interactively. The LCF-style theorem prover ensures that theorems are only derived from the inference rules by the abstract type and rely as such completely on a small core of rules. As a consequence, the *heavy-weight* approach produces proofs with a high degree of trust. The main disadvantages of this approach are that the user has to be proficient in logic and formal proof theory, and that a successful verification demands a high degree of interaction, i.e. user support, making this a labour-intensive technique. Additionally, this technique is not apt for generating counterexamples in case a property is not satisfied.
We believe that a proof using the *heavy-weight* approach is clearly desirable and necessary to ensure a high amount of trust and credibility in a voting scheme. Yet, we also argue that a successful verification process of voting schemes needs an early feedback in order to develop meaningful voting schemes and correctness criteria in the first place. Thus, this work analyses and employs the *light-weight* verification techniques SBMC and *auto-active* annotation-based program verification and examines their applicability on the formal verification of voting schemes. In the following, we amplify our motivation for these two techniques.

3.1.1. Auto-Active Deductive Program Verification

In general, we want program verification to be easy to use and the effort invested in doing the verification should be outweighed by the additional reliability, which can be gained in the event of a successful verification [LM10]. This task is tackled by the technique of *auto-active* deductive annotation-based verification with a significant effort to reduce the potential verification work for the programmer writing the program to be verified. The technique still provides universally reliable guarantees when a program can be successfully proven with respect to its specification. For this matter, the technique aims for a high portion of automation combined with early feedback to enhance the necessary interaction in order to accomplish the proof. More precisely, *auto-active* deductive program verification is a modular verification technique, wherein program artefacts are annotated with a method contract, consisting of a pre- and a postcondition and potentially more specification elements such as frame conditions, object invariants and auxiliary annotations specifying inner parts of the program artefact. When this contract can be successfully verified, the established guarantees can be used in order to verify other program artefacts, which invoke the already proven artefact. The same applies to specified parts inside the program artefact, as e.g. loop invariants for modularising loop iterations.

This approach attempts to provide a rather intuitive usability for programmers as its specification language aims to be very close to the language of the actual program code. In order to achieve this intuitive usability, the verifier needs an extensive set of usage rules in order to handle program language specific operations and types. As a consequence, the program artefact with its specification needs to be translated to a verifiable logical encoding. Thus, a set of logical *verification conditions* (VCs) is generated and then processed by a reasoning engine. Thereby, we can separate intricacies of the programming language and its modelling from the underlying logic and proof procedures.

At this point, we need to clarify, where we draw the boundary between *interactive* and *auto-active* verification, as within this work, we stick to *auto-active* verification. The boundary can be illustrated by defining exactly at which stage the interaction or user input is typically supplied. For most *interactive* proof assistants, the user interacts with the verifier after the VC generation. In the case of *auto-active* verification (lying between automatic and interactive verification) however, the *interaction* is closer to the program code and happens before generating the VCs. This programmer-friendly approach is enabled by powerful automatic satisfiability-modulo-theories (SMT) solvers [DB11].

Furthermore, proofs often closely relate to programs in the amount and structure of their case distinctions, e.g. `if`-statements usually result in performing a case split in the proof. The use of powerful SMT solvers enables us to receive early feedback also in case of failed proof attempts, in the form of precise counterexamples and the location in the program code or its specification, where the counterexample was generated. Moreover, the combination of early feedback and a good intuition of which part of the program causes the proof to fail enables the user to keep track of failed proof attempts and the program elements which possibly need to be re-proven after a change in the program. These means justify *auto-active* deductive program verification to bridge the gap between the user and the verification process, while still being apt to provide great reliability.

There is however a drawback concerning the established reliability. First off, the *auto-active* verification approach usually involves a complex verification tool with an interface to an SMT solver. This fact itself already lowers the gained reliability in comparison to *heavy-weight* verification, which provides intrinsic reliability based on its architecture with only little dependencies. Furthermore, the usually short verification time both for successful verification and verification failure also comes at a price [LM10]. This concerns the case when the tool fails to automatically discharge the required proof obligations, as usually little can be done in this event. However, the strengths of this technique indicate its important role in enhancing the formal verification process of voting schemes.

3.1.2. Software Bounded Model Checking

As indicated in the justification for the *auto-active* deductive verification technique, powerful SMT solvers can extensively enhance program verification. However, it can occur for *auto-active* verification that no proof obligation can be discharged automatically and we are stuck in the verification process. We justify in this section that the *software bounded model checking* (SBMC) technique can bridge this gap in order to gain at least partial reliability or generate smaller proof obligations in order to still produce a counterexample where the *auto-active* technique fails completely.
The SBMC technique does in contrast to *auto-active* or *interactive* deductive verification not aim to establish universal correctness guarantees or full reliability for all possible input parameters. Instead, it only considers a finite search or state space by cutting off program execution paths at a certain length. As such, this approach is comparable to systematic exhaustive testing up to a certain boundary of input size. However, SBMC provides means of symbolic representation for a state space and thus generally outperforms testing by large as truly exhaustive testing is rarely possible. Whereas for testing concrete inputs need to be provided and corresponding concrete outputs observed, SBMC tackles the potential *state space explosion* induced by simple mathematical combinatorics of all possible concrete input parameters and program paths by symbolic representation. While being more expressive than testing, we can usually generate simpler proof obligations than with deductive verification techniques and the emission of bounded guarantees is more likely.
More precisely, symbolic model checking uses boolean decision procedures and avoids the high space blow up, which occurs in approaches which use a compressed representation of all relations with a variable ordering [Bie+99]. SBMC uses modern SAT procedures in order to generate a propositional formula that is satisfiable if and only if a counterexample of a particular length (specified by the user) exists. The bound is thus given by the maximal length of a counterexample and we reduce the verification problem to propositional satisfiability. Furthermore, SBMC tools include additional decision procedures that support features of common complex programming languages such as complex memory models or further data types in order to check a wider range of correctness properties beyond the scope of traditional model checking without these decision procedures. This technique is usually argued to have the finding of small explicit counterexamples, which are easy to understand by the user, as its most important feature. Even though the potential for counterexamples within a larger scope still exists, we argue that SBMC is suitable for both finding counterexamples and the establishment of positive reliability guarantees. Bounded verification results provide great significance to full or unbounded reliability, based on the *small scope hypothesis*.

Small Scope Hypothesis The idea behind the *small scope hypothesis* is to artificially truncate the state space by checking only within some finite bounds [JD96]. It is an informal intuition for a “downward scalability” based on findings such as the *small model property*, which states that for some logical formulas with a possibly infinite variable domain, satisfiability checking for only a finite range of variables is sufficient [Pnu+02]. As such, most faults can be exposed by some “short” failing trace. The size of this failing trace may be influenced by various factors, e.g. the formula length, the number of operations or the complexity of input structures. One general danger of relying on the *small scope hypothesis* is the violation of resource bounds, which might not occur for the considered search space at all. However, this is not critical for the verification of abstract voting schemes within this work, as we apply the small scope hypothesis to the range of possible candidates, seats or voters, which also defines the actual resource bounds as e.g. the sizes of used data structures for vote counting in an according range. Evaluations of the *small scope hypothesis* have shown that exploiting the hypothesis and performing tests within a small scope can achieve complete coverage even when checking intricate methods [ADK03]. On the downside, we need to note that for some inputs, even the generation of proof obligations for a very small scope can be intractable and finding a more compact representation of the input might be a more promising approach for this case.

3.2. Selected Verification Tools

In the previous section, we have gained theoretical insights in the verification techniques employed within this work. Hereinafter, we illustrate the concrete software verification tools, which we use in our examinations of the following chapters in order to apply the chosen verification techniques.

3.2.1. Deductive Verification with VCC

The Verifying C Compiler (VCC), developed by the *European Microsoft Innovation Center* and the *RiSE* group at *Microsoft Research*, aims for full modular and deductive verification of concurrent C programs for every possible program execution [Coh+09]. It pursues an *auto-active* approach as described in Section 3.1.1 and uses the *Design by Contract* approach relying on contracts with function pre- and postconditions, state assertions, ghost code, loop and type invariants as well as frame conditions (i.e. describing which memory locations can be changed). These annotations are written in first order logic and other legal C statements. VCC translates the annotated C code into the intermediate verification language Boogie [DL05], which is then passed as an input to the verification condition generator Boogie [Bar+06]. The generated verification conditions are then fed into the Z3 SMT solver [MB08] and finally the proof result, in its minimal form simply the verification time together with either a success message or a partial counterexample, is presented to the user by VCC. As this work is not concerned with the verification of concurrent programs, we limit the following description of VCC’s usage to the verification of sequential programs.

Function Contracts VCC heavily relies on abstraction and modularisation and as such performs a static modular analysis for each function in isolation. This task is done with the help of the contracts of functions that the examined function calls and invariants of types used in the function's code. A function contract can be given for every function in C code and consists of a precondition with the keyword `requires`, a postcondition with the keyword `ensures`, and frame condition with the keyword `writes`. If parts of the pre- and the postcondition overlap, the keyword `maintains` can be used for combining both (overlapping) aspects into one predicate. A precondition declares under which condition a function may be called, a postcondition under which condition the function may return (when called in a state satisfying the precondition), and the frame condition describes the part of the whole program state that the function is allowed to modify. Whereas other functions have to guarantee the precondition of the function they are calling, the called function itself has to guarantee that its postcondition holds after the call and at most the specified locations in the frame condition have been changed afterwards. This also shows the duality of function contracts, as preconditions need to be weak enough to be fulfilled by calling functions, but strong enough for the function itself to be proven. Similarly, frame conditions need to be permissive enough to prove the postcondition, but strict enough to not impede verification of calling functions, and postconditions need to be weak enough to be verified to hold for a function contract, but strong enough to enable the other functions' proofs. Additionally to function contracts, VCC uses type invariants specifying data with either one- or two-state predicates. These type invariants can then be used at various points in the program.

Ghost Code A further concept, which is crucial for specification and verification with VCC, is *ghost code* only operating on *ghost states*, in contrast to regular operational (with respect to actual program states) code. It is only seen by the static verifier, but not the regular compiler, and comprises *ghost* type definitions, *ghost* fields, static or automatic *ghost* variables, *ghost* input and output parameters, as well as *ghost* state updates. In order to realise this division into two different kinds of code, a *ghost* memory state is being kept separate from the regular memory state by VCC and any data flow from the *ghost* state to the operational state of the software is forbidden. The types used in *ghost code* can either be regular C types, or special verification purpose types such as maps and other VCC specific types. One common application of *ghost code* is to keep *shadow copies* of implementation data, usually introducing abstractions or allowing for atomic *shadow* updates in order to enforce the usage of two-state invariants, specifying state transformations, on the overall system when this is not possible with the operational code.

Ownership Model and Type Invariants Speaking of states and type invariants, the exact notion of invariants and when they must hold is an important issue in a specification language targeted at an imperative programming language. Obviously, there are type invariants that cannot always hold as they depend on the states of other objects and a modification of any of these objects potentially violates the invariant. In a system with reusable components, the methodology must allow such behaviour and ensure that the invariants of the other objects are reestablished before it relies on them again. For this matter, VCC implements a Spec#-style *ownership model* organising the objects of the heap into a collection of tree structures and thereby also dealing with the important issue of aliasing, which arises in object-oriented programs [BLS05]. Edges in the ownership tree indicate ownership or an aggregate/sub-object relationship and a type or object invariant depends only on states in its sub-tree with the invariant being at the root. Methods can only call objects downwards in the ownership tree, but not upwards as to prevent method calls on inconsistent objects. Consequently, every object has an *ownership domain* containing all the objects directly or transitively owned by this object. Based on this model, VCC uses a range of specification elements in order to track an object's status in its meta-state (especially useful to deal with concurrent modification). We do not discuss these specification elements in detail at this point as verification within this work manages without them and instead makes heavy use of *ghost code*, which describes *ghost states* not prone to concurrent modification. However, we note that frame conditions strongly relate to the *ownership model* as granting write access to the root of an *ownership domain* automatically enables writing to the entire *ownership domain.*

Z3 SMT Solver The reasoning back-end of VCC is the Z3 SMT solver which consists of a simplifier, a compiler, a congruence closure core with solvers for equalities and uninterpreted functions, theory solvers for linear arithmetic, bit-vectors, arrays and tuples, an *E-matching* engine, and a Davis-Putnam-Logemann-Loveland(DPLL)-based SAT solver [MB08]. When Z3 receives the verification conditions from Boogie, it first applies an incomplete but efficient simplification in order to reduce the formula before this abstract syntax tree representation is converted into a set of clauses and congruence-closure nodes by the compiler.
Then the complex constraint solving commences with the congruence closure core which propagates equalities, assignments, atoms and literal assignments between the SAT solver, the theory solvers and the *E-matching* engine. With the truth assignments and links to atoms from the SAT solver, the congruence closure core propagates asserted equalities with the help of a data structure called an *E-graph following.* Nodes in this E-graph may point to one or more theory solvers and when nodes are merged, the merge is propagated as an equality to the theory solvers referenced in the intersection of these nodes. Vice versa, also the effects of the theory solvers as inferred equalities and atoms are assigned to the SAT solver. As verification conditions may also contain quantifiers, Z3 also needs a mechanism in order to instantiate these quantifiers appropriately and efficiently.
This is done by the *E-matching* engine, which is an abstract machine also using the mentioned E-graph with a well-known approach for quantifier reasoning. Generally speaking, algorithms are used to identify matches on E-graphs incrementally and efficiently. Z3 uses heuristics to select instances, which are "relevant" to the conjecture [DB07]. The key idea of *E-matching* is to consider those possible instances as relevant or a *match*, of which enough terms are represented in the current E-graph. As such, non ground terms from the quantified formula are selected as *patterns* and an instance of the formula

is considered a *match* whenever the any *pattern* with the same instantiation is in the E-graph. Additionally, Z3 also supports the more restrictive concept of *multi-patterns*, which operate similar to *patterns*, but use multiple non ground terms for one instantiation. Only when all of them are contained in the E-graph, the instantiation is considered a *match*. In addition, Z3 has the ability to produce partial models assigning values to the constants in the input and subsequently generate partial function graphs for predicate and function symbols.

Quantifier Instantiation Using E-Matching and Triggers In the previous paragraph, we have seen an overview of Z3's architecture including the *E-matching* engine which attempts to find good instantiations for quantifiers. Yet, however efficient this instantiation mechanism is, the issue of solving quantified formulas is in general undecidable. For this matter, VCC has an automatically integrated mechanism to enhance Z3's *E-matching* mechanism in deciding which instantiated instances are useful and which are not. Moreover, VCC also provides primitives such that the programmer or verification engineer can enhance this task with present program knowledge.
This mechanism is called *triggers* and VCC infers appropriate *triggers* automatically by default or the user can specify some *explicit triggers* manually [Coh+11]. These *triggers* are then passed as *patterns* to Z3 in order to guide the matching mechanism and consequently which instances of quantified formulas are generated. In general, VCC's mechanism to generate appropriate *triggers* is reliable and avoids any instantiation loops. However, complex specification elements may require the use of *explicit triggers* specified by the user. Usually, this is to reduce the set of generally admissible *triggers* or specify "bigger" *multi-triggers* in order to generate more restrictive *patterns* or *multi-patterns* respectively and cause the formula to be instantiated less often, generally leading to a better proof performance, but also possibly preventing the proof altogether. Sometimes there is the opposite situation, where we want to specify a less restrictive *trigger* as the proof gets stuck as a formula cannot be instantiated at all. For this matter, VCC provides some trivial *triggers*, as e.g. the functions `\match_long()` and `\match_ulong()`, matching on any instantiation for a given type. Additionally, very similar primitives are so-called *hints*, which intuitively indicate that they might have something to do with proving the following formula. They are usually used so that certain terms are not "missed" in the proof.

VCC Tool Suite VCC's tool chain includes Microsoft Research's Common Compiler Infrastructure (CCI) libraries for all usual compiler tasks such as name resolution and type and error check [Coh+09]. Subsequently, the program undergoes a range of source-to-source transformations in order to perform simplification, add proof obligations stemming from the methodology, and generate the Boogie source. Therein, Boogie encodes the input program according to a given C formalisation and adds minimal imperative control flow, procedural and functional abstractions, as well as types on top of first order predicate logic. The generated verification conditions are then fed into Z3, which uses fast decision

procedures for linear arithmetic. In case the proof succeeds, this is the whole work flow with VCC.
Yet, there may be genuine errors in the code or the annotations as well as a lack of computer memory, time or patience. Especially as this verification is in general undecidable, VCC provides a larger framework that includes tools for monitoring proof attempts, tracking and examining failed proofs as well as generating partial counterexample traces. The *VCC Model Viewer* is part of this framework and is able to translate partial counterexamples generated by Z3 into a representation for inspecting the sequence of program states that led to the failure. It also includes the values of local and global variables as well as the heap state.
Another possible reason to further examine the situation is when the prover takes an excessive amount of time to come up with either a proof or a refutation for a verification condition. For this matter, the *Z3 Inspector* allows to monitor the progress of Z3 linked to the code annotations for the user to pinpoint the responsible verification conditions. This can be either caused by a valid verification condition for which the prover requires a long time, or by an invalid verification condition for which the search for a counterexample takes a very long time. It is argued that identifying the problematic assertion for the latter case is a quick task with this tool [Coh+09]. In the former case, the *Z3 Axiom Profiler* is proposed for a closer inspection of the quantifier instantiation pattern with the objective to determine inefficiencies in the underlying axiomatisation of C or the program annotations and for example triggering cycles can be detected by this means. The developers further propose the usage of *Visual Studio IDE* (also used within this work) for a direct access of the described framework and tool chain and options for verifying only individual functions.

3.2.2. Bounded Verification with LLBMC

The Low-Level Bounded Model Checker (LLBMC), actively developed by the research group *Verification meets Algorithm Engineering* at *Karlsruhe Institute of Technology* (KIT), employs bounded model checking in order to verify properties in sequential C/C++ programs with the main purpose of finding bugs and run-time errors [MFS12]. This is a fully automatic approach also known by the name SBMC as described in Section 3.1.2, converting the originally generally undecidable verification task into a decidable, yet incomplete, endeavour through only considering a finite part of it. In order to achieve this, LLBMC restricts the number of considered loop iterations and nested function calls by loop unrolling and function inlining up to specified bounds and essentially analyses one large function with only finite runs, on which an efficient symbolic exhaustive search is performed.

Built-In Checks Concerning the verification capabilities, LLBMC provides a wide range of built-in checks for commonly occurring bugs in C programs. These include arithmetic overflow and underflow, logic or arithmetic shifts exceeding the bit-width, memory access at invalid addresses, invalid memory allocations and de-allocations, overlapping memory regions in `memcpy`, memory leaks, divisions by zero, as well as user defined and SBMC specific (i.e. manually given) assumptions and assertions using the keywords `assume` and `assert` (also usable by the C compiler) or their specification-only versions prefixed with `__llbmc_` (only usable by LLBMC). Additionally, LLBMC provides a measure to check, whether the specified bound for loop iterations (by using the option "`max-loop-iterations=`n") is sufficient. Each of these checks can be enabled or disabled independently with most of them enabled by default. For commonly used `for`-loops, LLBMC automatically identifies an appropriate upper bound for the number of loop iterations immediately, but for some more complex loops, an upper bound for the number of loop iterations cannot be easily determined. Custom specifications can be given with any legal C/C++-expression of type boolean and thus do not directly support expressions in full first order logic. Our examination within this work mainly targets these custom assertions in order to analyse more complex properties specific to voting schemes.

Encoding and Representation Verification with LLBMC is not done directly on C source code, but operates on the intermediate compiler representation LLVM-IR generated by the LLVM compiler framework with an according front-end such as `clang` or `llvm-gcc` [LA04]. LLVM-IR is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers consisting of type definitions, global variable declarations, and a representation of the program itself. The program is represented as multiple functions, each realised by a graph of instruction lists or so-called basic blocks. As such, this static program representation is in *static single assignment* (SSA) form, wherein each variable is assigned exactly once and assignments can hence be treated as logical equivalences.
LLBMC subsequently converts the LLVM-IR program into its internal logical representation ILR in the logic of bit-vectors and arrays with some extensions in order to handle the special semantics of memory allocation instructions, simplifies it by the usage of rewrite rules, and annotates it with LLBMC's built-in checks. Therein, LLBMC uses a bit-precise and untyped memory model, where arbitrary data can be read from any valid address, and memory is just an array of bytes with accomplishing stores and loads by a sequence of reads and writes on the logic level [SFM10]. By these means, common constructs in C code, such as casting blocks of memory containing a structure to a byte-array, are supported with dynamic memory and casts. Furthermore, LLBMC accomplishes function inlining and loop unrolling by using code from the LLVM libraries and the basic block graph becomes a directed acyclic graph (DAG) afterwards.
Finally, LLBMC passes the ILR formula, which is satisfiable if and only if the program violates the given conditions, to an SMT solver, which is STP [GD07] with either a configurable version of MiniSat or CryptoMiniSat [ES03]. Small and bit-precise counterexamples are generated if any of the conditions does not hold.

4. Deductive Verification of Plurality Voting with VCC

In this chapter, we examine annotation-based modular deductive verification with the tool VCC for the first-past-the-post (FPTP) voting scheme plurality voting from Section 2.3.1 with the monotonicity criterion from Section 2.4.1. We have seen in Section 3.1.1 that this "medium-weight" approach makes use of the modular structure of programs as it uses method or block contracts and loop invariants instead of inlining methods, blocks or loop statements. The idea of this modular approach is to reason about an abstract heap or set of program elements, which does not need to reflect the actual whole program state, but only an abstract representation of it. This modular approach usually needs a lot of manual specification, but has the benefit of a more concise and understandable specification and also speeds up the automatic verification process (once the specification is complete) and gives a universally valid (i.e. unbounded) proof.
In the following, we examine the feasibility of this approach for our verification target and furthermore explore VCC's capability to give bounded guarantees in case we have no loop invariant, but do loop unrolling, which is aimed to be done automatically without any user interaction. Loop unrolling means that, as we know how many loop iterations there are, we translate a loop into nested `if`-statements with a given nesting depth, which is determined by the number of possible loop iterations. On the one hand, this means we lose universal validity, but instead we do not need to specify a loop invariant, which is in general hard to determine.

4.1. FPTP Implementation and Specification of Monotonicity Criterion

This section describes the verification of the monotonicity criterion for FPTP plurality voting using VCC. As in Section 2.3, we use some common constants for all voting schemes. V always denotes the number of voters and C the number of candidates. For the implementation, we do not use two method calls which would enable performing modular verification for calling the voting scheme, but instead decide to inline both calls. As we aim at reducing the necessary user interaction in the verification process, the inlining is done in order to be able to selectively replace parts of the implementation by a more declarative description in terms of VCC annotations. Furthermore, our implementation uses only ghost code, i.e. ghost parameters, ghost variables and ghost statements. This is done as to firstly simplify reasoning and abstract away from C specific issues as e.g. aliasing between different arrays which is not possible for *ghost* variables, and secondly for convenience as to be able to use lambda-expressions describing contents of a ballot box, as well as to let a method return more than one value by using ghost out parameters instead of integer pointers. However, the fact that we only use ghost code also means that the code is not read by a C compiler and hence cannot be executed as a C program, but only serves for our verification purposes.

Implementation of Plurality Voting In the following, we present and describe the implementation of plurality voting as used for the experiments in this chapter. The implementation is as shown in Listing 4.1, having the three parameters `votes`, `res` and `elect`. The array `votes` is the input and represents the ballot box, which contains V votes, each an integer between 1 and C, meaning the voter has cast a vote for the according candidate. The two parameters `res` and `elect` are the output. Therein `elect` stands for the election outcome, which is denoted by either an integer between 1 and C for the according elected candidate, or 0 in case there is a tie and no candidate gets elected. The array `res` is used to count the given votes for each candidate i in the according entry `res[`i`]`. It is actually not a necessary output of the election, but rather an intermediate value. However, we specify it as an output parameter as to be able to use it in the VCC annotation for the whole method as an enhancement for the verification process. Additionally to these three variables, our implementation uses the two helper variables `i` and `max`. The variable `i` is used for iterating through loops and `max` to determine the maximum amount of votes for a candidate.

```
 1 void voting(_(ghost int votes[int])
 2             _(ghost int res[int])
 3             _(ghost int elect)) {
 4     _(ghost int i = 0;)
 5     _(ghost int max = -1;)
 6     _(ghost elect = 0;)
 7     _(ghost int res[int] = \lambda int i; 0;)
 8     _(ghost for (i = 0; i < V; i++) res[votes[i]]++;)
 9     _(ghost for (i = 1; i <= C; i++) {
10         if (max < res[i]) {
11             max = res[i];
12             elect = i;
13         } else if (max == res[i]) elect = 0;
14     })
15 }
```

Listing 4.1: Plurality Voting

We start by initialising all these variables to zero respectively or a negative value for `max` as to distinguish the preset value from the number of votes for candidates without any votes. In Line 7, we benefit from the usage of ghost code by using a lambda-expression as an easy means to initialise a whole array. The actual vote tallying is done in Line 8 by reading the candidate, which voter `i` has voted for and incrementing the according entry in `res`. Subsequently starting in Line 9, we iterate through all candidates and compare the numbers of votes each candidate received, which are stored in the array `res`. Thereby, we determine the maximum amount of votes for one candidate and the according candidate, and store these values in `max` and `elect` respectively. In case there is more than one candidate, who received the maximum amount of votes, we set `elect` to zero. As the method `voting` does not return any value visible to the C compiler and there are only ghost variables, there is no return statement or anything similar.

Specification of Monotonicity Criterion The correctness property we want to prove for this voting scheme is monotonicity (as in Definition 6 on page 19). For voting schemes as plurality voting, where each voter can only vote for one candidate, we can simplify this definition as follows:

Definition 9 (Monotonicity for Plurality Voting) *For a set of candidates C, a set of voters V and a set of ballot boxes B, we call a* plurality voting scheme *s monotone iff*

$$\begin{aligned}&\forall m \in C, n \in V, b, b' \in B :\\ &\qquad (b_n \neq m \wedge b'_n = m \wedge \forall i \in V : i \neq n \Rightarrow b_i = b'_i)\\ &\qquad\qquad \Rightarrow (s(b) = m \Rightarrow s(b') = m).\end{aligned}$$

Therein we reason about two different ballot box inputs b and b' for plurality voting, herein denoted by the function s. Hence, in the C implementation, we need to call the `voting`-method (Listing 4.1) with the actual voting scheme twice, each time with slightly different parameters `v1` and `v2` (corresponding to the abstract ballot boxes b and b'). However, we actually inline both method calls in order to enable reasoning about all parameters of the `voting`-method in the contract for the monotonicity criterion, as specified in Listing 4.2 as a VCC contract for the method `correctness`:

```
 1 #define valid_cand(a) ((0 < a) && (a <= C))
 2 #define valid_voter(a) ((0 <= a) && (a < V))
 3
 4 void correctness(_(ghost int v1[int]) _(ghost int v2[int])
 5                  _(out int elect1) _(out int elect2)
 6                  int m, int n)
 7     _(requires 0 < V && V < INT_MAX - 1)
 8     _(requires 0 < C && C < INT_MAX - 1)
 9     _(requires \forall int i; {v1[i]}
10                     (valid_voter(i) ==> valid_cand(v1[i]))
11                  && (valid_voter(i) ==> valid_cand(v2[i])))
12     _(requires v1[n] != v2[n])
13     _(requires valid_cand(m) && valid_voter(n))
14     _(requires v2 == \lambda int i; i == n ? m : v1[i])
15     _(ensures (m == elect1) ==> (elect1 == elect2))
16 { ... }
```

Listing 4.2: Monotonicity Specification for Plurality Voting in VCC

In addition to the two ballot boxes `v1` and `v2` and the two respectively elected candidates (or zero if no candidate gets elected) `elect1` and `elect2`, we use the two parameters `m` and `n` as to denote the raised candidate m and its raising vote n from the monotonicity criterion in Definition 9. The conjoined formulas of the precondition are denoted with the keyword ***requires***, whereas ***ensures*** denotes the postcondition. In order to ease readability, the definitions `valid_cand` and `valid_voter` in Lines 1 and 2 are abbreviations to express that a variable denotes a valid candidate or valid voter respectively. Another definition (Line 7), which we use for our specification, is `INT_MAX`, which denotes the maximum value a C integer variable can hold. Hence, our specification is technically bounded and does not deal with integers above this value. However, for the deductive methodology of VCC the precise number of `INT_MAX` does not matter and is only specified to avoid considering an integer overflow as the integer data type in C is bounded. It is thus justified to assume that a proof for this specification is also valid for greater numbers, when we neglect the potential occurrence of the C specific technicality of an integer overflow.

We start our contract by limiting the numbers of voters and candidates to an appropriate range, i.e. they are both strictly positive and less than `INT_MAX`. Furthermore (Line 13), we use `valid_cand` and `valid_voter` as to make `m` and `n` a valid candidate and a valid voter, respectively. The same well-formedness assumption is made for the votes in ballot boxes `v1` and `v2` in Line 10, i.e. every valid voter votes for a valid candidate. The term `{v1[i]}` after the quantified variable is a so-called *matching trigger*, which is a *pattern* specified in order to guide the SMT solver for instantiating the variable with appropriate terms. The usage of *triggers* is necessary as quantifier instantiation is a task, which is generally undecidable and thus provides a means to support this process through knowledge by the user. In the remaining lines, we specify the monotonicity criterion

with `v1` and `v2` differing precisely in the vote cast by voter n in Line 12. We use a lambda-expression in Line 14 to specify that the ballot box `v2` is identical to `v1` for all voters, except for voter `n` who votes for candidate `m`. Finally, we conclude the contract with the postcondition in Line 15 by asserting that both elected candidates `elect1` and `elect2` are identical if candidate `m` has won the first election.
In the the following we examine VCC's capabilities in performing a deductive proof that the monotonicity criterion as specified in Listing 4.2 is satisfied by our implementation of plurality voting as illustrated in Listing 4.1. For this analysis and all experiments throughout this chapter, we use VCC 2.3.10214.0, Boogie 2.1.40227.0 and Z3 3.2 on a dual-core Intel Core 2 Duo E7300 processor at 2.66 GHz with 1.5 GB of available memory and specify a timeout of 7 hours (the script which we used can be found in Listing A.1 in Appendix A).

4.2. Full Verification with Auxiliary Specifications

In this section, we explore a verification process using VCC with the goal to have a general and comprehensive proof of the monotonicity criterion for our implementation of plurality voting. Having given the implementation of the voting scheme and the property to be proven for it, we still need some auxiliary specifications for VCC's *auto-active* approach. In our case these are loop invariants to specify the two loop statements. Therein, we must find an appropriate trade-off between invariants that are strong and expressive enough to eventually deduce the postcondition of the monotonicity specification, but also simple enough to prove that they hold before entering the loop as well as for every loop iteration. Moreover as our specification comprises quantified formulas, we need to support the E-matching process in finding useful quantifier instantions, which is in general undecidable. This means that we need to guide quantifier instantiation using pattern matching by providing a useful combination of triggers and specification elements in order to advance the solver task. Another step towards a full verification consists in supplying a recursive specification function `count` (Listing 4.3) for the specification of the first loop. We denote a function without any side-effects by the keyword ***`pure`***, the return value by the keyword ***`returns`***, and variant, which decreases in each invocation, by the keyword ***`decreases`***. We could have used an abstract map at this point, but experiments have shown that the resulting term would have been to complex and is thus outperformed by using the following specification function:

```
_(ghost _(pure) \integer count(int votes[int],
                               int upto,
                               int cand)
    _(decreases upto)
    _(returns (upto <= 0) ?
            0 : count(votes, upto-1, cand)
              + (votes[upto-1] == cand ? 1 : 0))
{
    return (upto <= 0) ?
            0 : count(votes, upto-1, cand)
              + (votes[upto-1] == cand ? 1 : 0);
})
```

Listing 4.3: Auxiliary Recursive Specification Function

This function is used to ease specifying which votes in the ballot box are already counted. Therein the parameters are the ballot box `votes`, the variable `upto` to denote up to which index the votes are counted, and the variable `cand` to specify the candidate whose votes are being counted. We count all votes starting from the index specified by `upto` until the first entry and recursively sum them all up all. The function terminates when `upto` is equal to or smaller than zero. Hence, we can use this function in the loop invariant in Listing 4.4 for the first loop of the voting scheme implementation as in Listing 4.1 at Line 8, which tallies all votes and aggregates them by candidate in the array `res`.

```
1 _(ghost for (int i = 0; i < V; i++)
2     _(invariant 0 <= i && i <= V
3              && valid_cand(votes[i]))
4     _(invariant \forall int k;
5         (valid_cand(k) ==> 0 <= res[k] && res[k] <= i))
6     _(invariant \forall int k; {res[k], count(votes, i, k)}
7         (valid_cand(k) ==>
8             (res[k] == count(votes, i, k))))
9 { ... })
```

Listing 4.4: Loop Invariant for Tallying Votes

In Line 2, we specify the loop iteration variable to be in range and also the referenced vote in the ballot box to denote a valid candidate. We specify furthermore in Line 4 that the counted votes for each candidate in the array `res` are a positive number and not greater than the loop iteration variable, i.e. in each iteration at most one vote is counted. Finally in Line 7, all entries in the array `res` are specified by using the function `count` in Listing 4.3 to exactly denote the amount of votes up to the current entry for the according candidate. This quantification relating the array `res` with calls of the specification function `count` necessitates the specification of according triggers to guide the instantiation with correct values. Having specified the vote tallying process of one line of code in three lines of specification, the loop invariant of the loop determining the winner of the election in four lines of code necessitates an even more complex specification. For this specification in Listing 4.5, we define another abbreviation along the lines of the previously defined ones for a valid voter and a valid candidate in Listing 4.2. In this case we define the abbreviation `valid_candZero` in Line 1 to describe either a valid candidate or the value zero, as in the following the result is set to zero in case of a tie.

```
 1 #define valid_candZero(a) ((0 <= a) && (a <= C))
 2
 3 _(ghost for (int i = 1; i <= C; i++)
 4     _(invariant valid_candZero(i-1)
 5             && valid_candZero(elect))
 6     _(invariant elect != 0 ==> res[elect] == max)
 7     _(invariant elect != 0
 8         <==> (\exists int k; {res[k]}
 9             {\match_long(k)} {:hint \match_long(k)}
10                0 < k && k < i
11             && (\forall int j; 0 < j && j < i && j != k
12                 ==> res[j] < res[k])))
13     _(invariant 0 <= max
14         ==> \exists int k; 0 < k && k < i && res[k] == max)
15     _(invariant \forall int k; 0 < k && k < i
16             ==> res[k] <= max)
17 { ... })
```

Listing 4.5: Loop Invariant for Determining Candidate with Most Votes

Accordingly, we specify the iteration variable to be in range and the variable `elect` for the elected candidate in Line 4 of Listing 4.5 to either denote a valid candidate or the value zero. If furthermore (Line 6), the variable `elect` denotes a valid candidate, the vote count for this candidate is the value of the variable `max` with the current maximum. We then specify in Line 7 that if and only if `elect` contains a valid candidate, there exists a candidate among the ones already considered (i.e. smaller than the iteration variable), who received strictly more votes than all other candidates already considered, i.e. there really exists a unique maximum. This rather complex formula necessitates a trigger for instantiating the array index for `res` as well as a trigger and a hint in order to get the instantiation of the quantified variable `k` started. The function `\match_long()` is a trivial specification function, which is true for all signed integer values. Subsequently, we specify in Line 13 that if the value of the variable `max` (initialised with -1) has changed, then there exists a vote count entry in the array `res` which is equal to the value of `max`. It finally remains to specify in Line 15 that all vote counts in the array `res` are either smaller than or equal to the value of `max`.

First Verification Attempt We specified both loop invariants for our implementation of the plurality voting scheme in Listing 4.1 and also a contract for the monotonicity criterion in Listing 4.2. Subsequently, we inline both calls of the voting scheme and rename its variables `votes`, `res`, `elect` and `max` by suffixing "`1`" for the first and "`2`" for the second call and shorten `votes1` and `votes2` to `v1` and `v2` in order to reason about them in the method contract, which specifies the monotonicity criterion. The complete specification can be found in Listing A.2 in Appendix A. When attempting to verify the contract, we observe that the loop invariant for the determination of the first elected candidate (Listing 4.5) verifies, whereas the loop invariant for determining the winner of the second election cannot be successfully proven (both in Listing 4.5). We suspect this to be based on the limited available memory supply as the E-matching process for non-trivial and nested combinations of quantifiers in complex formulas is a memory intensive task. In order to prove the whole contract with all loop invariants, we make the following changes to the second loop invariant, which are shown in Listing 4.6:

```
1  _(ghost for (int i = 1; i <= C; i++)
2      _(invariant valid_candZero(i-1)
3          && 0 <= elect && elect < i)
4      _(invariant elect != 0 ==> res[elect] == max)
5      _(invariant elect == 0 && 1 < i ==>
6          \exists int j, k; 0 < j && j < i && 0 < k && k < i
7              && k != j && res[k] == res[j])
8      _(invariant 0 <= max
9          ==> \exists int k; 0 < k && k < i && res[k] == max)
10     _(invariant \forall int k; 0 < k && k < i
11         ==> res[k] <= max)
12     _(invariant (\forall int j; {res[j]}
13         0 < j && j < i && elect != j
14         ==> (\exists int k; 0 < k && k < i
15                 && k != j && res[j] <= res[k])))
16 { ... })
```

Listing 4.6: Second Loop Invariant for Determining Candidate with Most Votes

Therein, the complex formula from Line 7 of the first loop invariant in Listing 4.5 does not advance the verification, but impedes the verification process, and we drop it. Instead the following three changes in Lines 3, 5 and 12 bring the verification process forward. Firstly, we limit the value of the variable `elect` in Line 3 to be smaller than the iteration variable, i.e. it can only specify a candidate, who has already been considered in the iteration. Secondly in Line 5, we specify the case that `elect` is equal to zero and we have already done at least one iteration. In this case, there exist at least two candidates in the set of already considered candidates, who have received exactly the same amount of votes. This formula holds as `elect` is only equal to zero before the loop and if there are at least two tied candidates in the set of candidates already considered. Thirdly and lastly, we state in Line 12 that for all candidates already considered other than the one currently stored in the variable `elect`, there exists another candidate we already considered (possibly the candidate currently holding the maximum amount of votes) with at least the same number of votes, i.e. none of the already considered candidates who are not elected has the maximum amount votes. As this is a formula with nested quantifiers, making it comparatively complex, we need to specify a trigger for the instantiation of the outer quantified variable. We do this with the knowledge that it is an index of the array `res`.

Results For this implementation with the shown auxiliary specifications (the complete specification can be found in Listing A.2 in Appendix A), VCC is neither able to give a universal proof nor a counterexample, but the memory consumption exceeded the limit of 1.5 GB. However, when specifying a bound for the number of voters, VCC is able to show that plurality voting satisfies the monotonicity criterion for up to 5 voters and any number of candidates. The run-times of the VCC depending on the number of voters are shown in Fig. 4.1. The graph does not indicate a strong effect of the number of voters on VCC's run-time, but instead VCC fails to give results for a higher number of voters. By using the "Z3 Inspector" (Section 3.2.1), we observe that the postcondition "`(m == elect1) ==> (elect1 == elect2))`" causes the generation of a great number of instantiation samples, which we suspect to be the main cause for the unsuccessful universal verification. However, VCC does not allow us to guide the sample generation in this case as no quantifier is used in the formula.

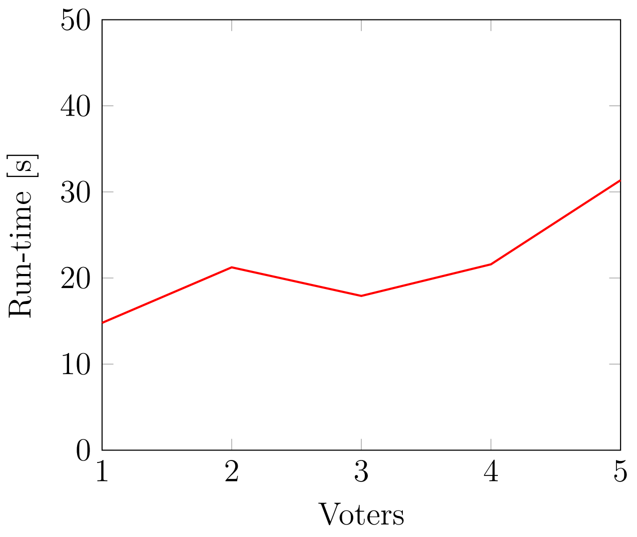

Figure 4.1.: VCC Verification Performance with Monotonicity Criterion for Plurality Voting Fully Annotated

Using an Intermediate Lemma to Simplify the Verification In order to further investigate on the capabilities of VCC for our means, we replace the postcondition of the monotonicity criterion by the lemma given as assertions with the keyword ***assert***

in Listing 4.7. This is done in order to help the verification mechanism as we found the sample generation for the postcondition for the monotonicity criterion to be the crucial part in the verification process. Firstly, we assert that the precondition still holds in Lines 1 to 5 as well as that the two ballot boxes `v1` and `v2` can be distinguished. Furthermore, we assert in Lines 6 and 8 that the counting in the arrays `res1` and `res2` is done correctly, i.e. as specified by the specification function `count` from Listing 4.3. In Line 10, the first elected candidate `elect1` is asserted to have the unique maximum amount of votes as counted and stored in the array `res1`. Then we assert in Line 13 that for all candidates other than the candidate `elect2` elected in the second election, there exists another candidate with at least the same amount of votes as stored in the array `res2`. Finally, we provide formulas in Lines 17 and 18 in order to provide easy triggers for the instantiation of nested array operations as `res1[v1[j]]` and `res2[v2[j]]`. Essentially, this is an aggregation of all loop invariants in our program combined with the repetition of the precondition.

```
1  _(assert 0 < V && V < INT_MAX - 1)
2  _(assert 0 < C && C < INT_MAX - 1)
3  _(assert v1 != v2)
4  _(assert valid_cand(m) && valid_voter(n))
5  _(assert v2 == \lambda int i; i == n ? m : v1[i])
6  _(assert \forall int k;
7      (valid_candZero(k) ==> (res1[k] == count(v1, V, k))))
8  _(assert \forall int k;
9      (valid_candZero(k) ==> (res2[k] == count(v2, V, k))))
10 _(assert elect1 != 0
11     ==> (\forall int k; 0 < k && k <= C && k != elect1
12             ==> res1[k] < res1[elect1]))
13 _(assert (\forall int j; {res2[j]}
14     0 < j && j <= C && elect2 != j
15     ==> (\exists int k; 0 < k && k <= C && k != j
16             && res2[j] <= res2[k])))
17 _(assert \forall int j; \match_long(res1[v1[j]]))
18 _(assert \forall int j; valid_voter(j)
19                 ==> \match_long(res2[v2[j]]))
```

Listing 4.7: Specification of Lemma Which Implies Monotonicity for Plurality Voting

The lemma stated in Listing 4.7 (the complete specification can be found in Listing A.3 in Appendix A) can thus be successfully proven to hold our implementation of plurality voting by VCC without specifying any bounds other than those preventing an overflow as in the monotonicity specification in Listing 4.2. We subsequently attempt to prove the monotonicity criterion for a completely empty program with the same parameters as in Listing 4.2 and having the lemma from Listing 4.7 as its precondition and the postcondition of the monotonicity specification in Listing 4.2. During this task, we observed that changing the order of the specification may greatly impede the verification process. Again, we do not achieve a universal proof, but bounded guarantees with run-times as shown in Fig. 4.2:

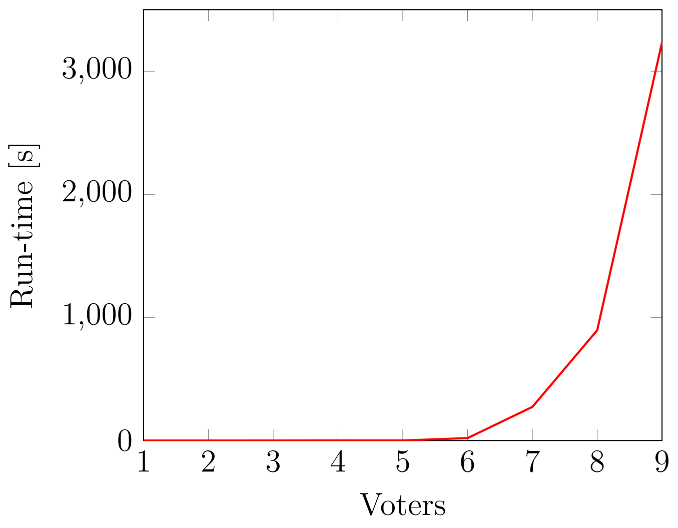

Figure 4.2.: VCC Verification Performance with Monotonicity Criterion for a Lemma Satisfied by Plurality Voting

Therein, we see that VCC proves the monotonicity criterion for up to 5 voters by using the specified lemma in Listing 4.7 almost instantaneously, i.e. in less than a second. However, the run-times for the verification with 6, 7, 8 and 9 voters quickly increases its memory consumption and indicates exponentially increasing run-times. The reason for this behaviour only for the number of voters might be based on the fact that there are C^V possible ballot boxes and Z3 does an exhaustive search as it fails to give universal guarantees. At this point, we should also note that there are various different quantified formulas as well as a recursive specification function are involved, which all have to be combined in order to deduce the postcondition in a non-linear manner. Our experiments foster the suspicion that this complex combination is highly impractical for VCC.

Verification Challenges As stated above in this section and shown in Listing 4.6, the successful verification of all loop invariants needed some asymmetrical changes. This is surprising as the other parts verified successfully and the remaining loop invariant was identical modulo renaming to another one in the first election, which verified successfully. Here, the complex instantiation process of Z3 plays a large role, as we discovered in our analysis with the “Z3 Inspector” (Section 3.2.1). After the successful verification of the first three loop invariants (i.e. the whole first election and the vote tallying for the second election), Z3 created too many samples for the instantiations in order to prove the last loop invariant (i.e. the determination of the candidate with the most votes). Providing the right combination of matching hints and triggers is a difficult task, especially as our verification indicates that this is not completely modular. We observe that a specification for a loop leading to a successful proof of the loop invariant can be insufficient for verifying the same loop at a later point in the program. This makes the program specification very fragile, the verification partially unpredictable and consequently more error-prone. We suspect this behaviour to be at least partially based on the length and thereby the higher complexity of our program as VCC provides a great extent of means to enhance modularisation and abstract as its methodology heavily relies on these principles. In the following, we investigate on means in order to simplify the program specification.

4.3. Simplifying the Specification with Loop Unrolling

From the previous experiments in Section 4.2 with VCC in proving the monotonicity criterion for plurality voting, we have gained confidence that the complex specification is the main reason for the poor verification results. However, we achieved proofs, which are only bounded in the number of voters. On this basis, we examine whether VCC can produce bounded verification results for higher input boundaries. In this section we therefore apply the bounded verification technique of loop unrolling in order to minimise complex specifications and to analyse VCC's potential with this technique.

Loop Unrolling in VCC When we mention loop unrolling throughout the remaining part of this section, we refer to the following technique. We assume a program containing a ghost `for`-loop with an initial value α and a non-inclusive upper bound β for the iteration variable i as well as a loop body Φ which depends on the current value of i as follows (Listing 4.8):

```
_(ghost for (int i = α; i < β; i++) { Φ(i); })
```

Listing 4.8: Ghost For-Loop in VCC

Then we can transform this loop into an `if`-cascade if we know the value of β beforehand. Let us assume without loss of generality – as to simplify the demonstration – that β has the value 2. We hence complement the precondition with "`_(requires β == 2)`" and the unrolled loop is of the form as shown in Listing 4.9:

```
_(ghost int i = α;)
_(ghost if (i < β) {
    Φ(i);
    i++;
    if (i < β) {
        Φ(i);
        i++;
        _(assert i == β)
    }
})
```

Listing 4.9: Unrolled Ghost Loop in VCC for β Equals Two

This technique can be applied for any bound of β in a straightforward fashion by further nesting of the `if`-statement right after the last incremental of the iteration variable i and accordingly putting the assertion in the body of the innermost `if`-statement.

Results for Fully Unrolled Loops In the following experiment, we apply this technique to all loops in our implementation of FPTP plurality voting and attempt to prove the monotonicity criterion for bounded numbers of candidates and voters. For this implementation, VCC is able to show that plurality voting satisfies the monotonicity criterion for up to 3 candidates and 6 voters, 5 candidates and 5 voters or 7 candidates and 2 voters within the timeout of seven hours. The run-times of VCC depending on the number of voters are shown in Fig. 4.3 (separate curves for 1, 3, 5, and 7 candidates). The graph indicates an exponential time complexity in the number of voters. We learn from these measurements that VCC is not apt for our implementation where all loops are unrolled.

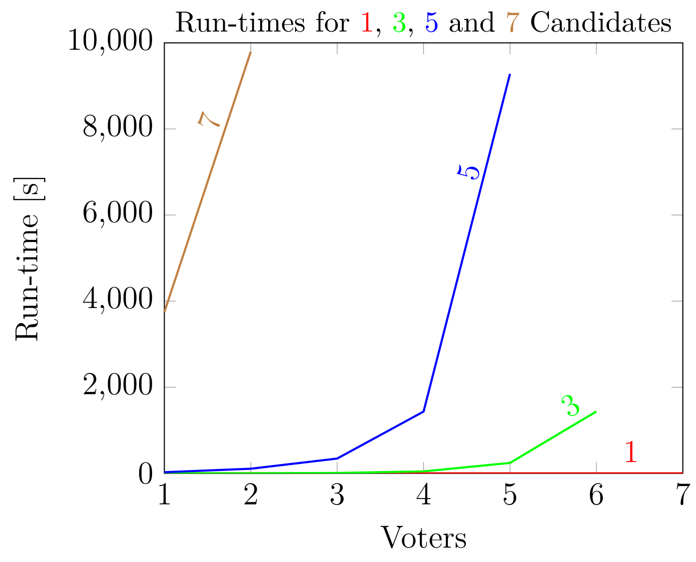

Figure 4.3.: VCC Verification Performance with Monotonicity Criterion for Plurality Voting with Manually Unrolled Loops

Results for Selective Loop Unrolling Hence, we examine whether only unrolling some loops advances VCC's verification process and thus apply the unrolling technique from Listing 4.9 selectively. In the following, the results for selective unrolling of either only the vote counting loop (in the following abbreviated as "count loop") or only the loop to determine the maximum and the elected candidate (in the following abbreviated as "max loop") are shown in Fig. 4.4, where the run-times are shown for a number of five candidates as this was the highest bound wherein meaningful results could be found.

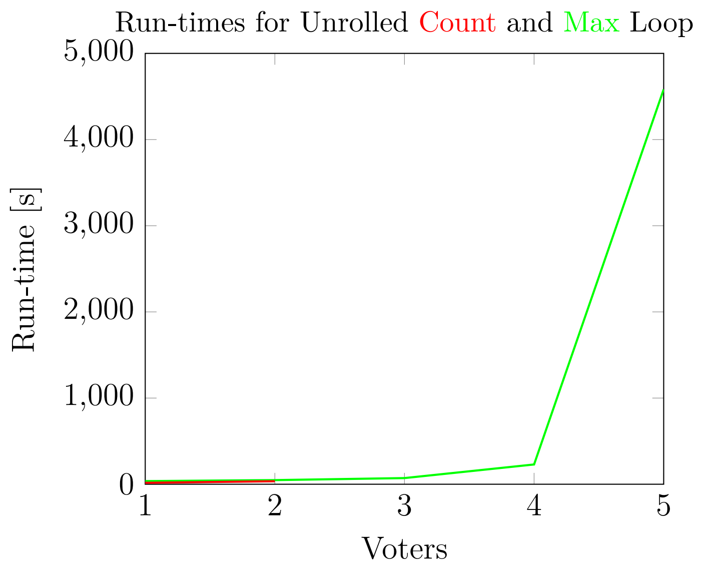

Figure 4.4.: VCC Verification Performance with Monotonicity Criterion for Plurality Voting with Partial Unrolling

The graph in Fig. 4.4 shows that unrolling the “count loop” only achieves proofs for up to 2 voters, which terminates after only 35 seconds, but exceeds the available memory of 1.5 GB for a higher number of voters. However, the unrolling of the “max loop” achieves proofs for up to 5 voters where run-times are growing exponentially in the number of voters. If we compare this experiment to the one in Fig. 4.3 where all votes are unrolled, we observe that the achieved number of voters is the same, but the performance for the unrolled “max loop” is considerably better, however still showing an exponential run-time development.
Finally, we can conclude that VCC appears impractical for proving that the monotonicity criterion holds for two sequential runs of a small voting scheme such as plurality voting. Our findings indicate that problems involving combinatorics, such as the potentially complex effects of a specific structure of a ballot box, provide challenges to the matching mechanism used by VCC, which are hard to specify and thus make the verification process fragile and difficult.

5. Bounded Verification with LLBMC and Counterexample Generation

In this chapter, we examine the verification technique *software bounded model checking* (SBMC) with the tool LLBMC for the voting schemes plurality voting, approval voting, instant-runoff voting and single transferable vote (STV) with respect to appropriate correctness criteria from Section 2.4. We have seen in Section 3.1.2 that contrary to deductive verification, this approach does not provide any general correctness guarantee, as it analyses up to a certain program length. As such, this technique relies on the small scope hypothesis, which states that a high proportion of bugs can be found for test inputs within some small scope [ADK03]. Using the SBMC approach, correctness guarantees within the specified bounds can be achieved fully automatically without user support. Furthermore, the SBMC approach can also generate counterexamples for specifications which are not satisfied by the program within the specified bounds.
Technically, the bound is always the length of the execution path, which can be bound by giving a maximum number of loop iterations and a maximum function call depth. For the matter of this case study, there are no recursive function calls and hence only the maximum number of loop iterations must be considered. Analysing voting schemes, the bounds always only depend on the number of voters, the number of candidates and for STV (Section 2.3.4) also the number of seats to be elected.

5.1. General Setup and Encoding

We analyse different voting schemes and correctness properties, but the setups for all experiments using LLBMC are very similar to each other. First off, all experiments are done with LLBMC 2013.1 and LLVM 3.3 with a timeout at 1500 seconds on a dual-core Intel Core 2 Duo E7300 processor at 2.66 GHz with 4 GB of memory. As in Section 2.3, we use some common constants for all voting schemes. V always denotes the number of voters, C the number of candidates, and S the number of seats in case the voting scheme is not *resolute* (in our case study only STV).
As LLBMC provides a whole program analysis, we start with a main entry point (exemplary shown in Listing 5.1 on the next page for one-vote voting schemes and a relational correctness criterion), where all the program variables and specification variables of the pre-state are initialised with symbolic values. These variables can be understood as implicitly universally quantified within the scope of the whole contract to be proven, making use of underspecification. Since only simple assumptions and assertions without quantifiers are allowed within LLBMC, we use this mechanism to

establish implicitly universally quantified variables in order to formalise correctness properties for arbitrary but fixed input values. In the example in Listing 5.1, all values in the array of votes are initialised as undefined positive (including zero) integer values with the LLBMC primitive "`__llbmc_nondef_unsigned_int()`". Instead of using this primitive in order to explicitly set the parameters to undefined or non-deterministically chosen values, we could have also used undetermined parameters and hence omitted the main entry point. As however the understanding of undetermined parameters can differ in general (e.g. some standards consider them to be initialised with zero values), we justify this to be the more comprehensive approach.
In the example in Listing 5.1, we initialise two sets of votes, which are needed for voting scheme properties that relate two runs of the voting scheme, as e.g. the monotonicity criterion (Section 2.4.1). However, for other criteria as e.g. the Condorcet loser criterion (Section 2.4.2), one set of votes suffices. Additionally to the input values for the actual voting scheme, we use the same primitive to initialise further variables, in order to reference specific votes and voters in the ballot box and use them in the contract for the correctness property. For the example in Listing 5.1, these are two values which represent a symbolic vote and a symbolic voter.

```
int main(int argc, char *argv[]) {
    uint v1[V], v2[V];
    for (uint i = 0; i < V; i++) {
        v1[i] = __llbmc_nondef_unsigned_int();
        v2[i] = __llbmc_nondef_unsigned_int();
    }
    uint vote  = __llbmc_nondef_unsigned_int();
    uint voter = __llbmc_nondef_unsigned_int();

    correctness(v1, v2, vote, voter);
    return 0;
}
```

Listing 5.1: Main Method for One-Vote Voting Schemes and Relational Properties

From the main entry point, we pass the initialised variables (in this case `v1`, `v2`, `vote` and `voter`) as parameters and call the `correctness`-method, where we specify the correctness property as well as some well-formedness assumptions concerning the input parameters and call the voting scheme. The specification for LLBMC is done using assumptions with the keyword `assume` for the precondition and assertions with the keyword `assert` for the postcondition. As these assumptions and assertions only allow legal C statements and not full first order logic, we need to translate the missing specification primitives, i.e. quantifiers, into C statements.
In the following, we demonstrate the translation process using a formula in the precondition with the keyword *`assume`*. For formulas in the postcondition, the translation is identical, except that the keyword *`assert`* is used instead of *`assume`*. We translate a

universally quantified formula in the precondition (keyword *assume*) as in listing 5.2a into a for-loop, which iterates over the quantified variable i with the inner formula in its body as in listing 5.2b. This translation does however not work for general quantification, but the analysis of voting schemes only uses bounded quantification. Moreover, nested quantification also necessitates an adaption of this technique, which is usually a straightforward task, but not shown here as to simplify the demonstration.

```
assume (∀i : α ≤ i < β → φ(i));
```

a: In First Order Logic

```
for (int i = α; i < β; i++)
    assume (φ(i));
```

b: Formula in C Code

Listing 5.2: Exemplary Translation of Universal Formula (a) into C Code (b) for LLBMC

An existentially quantified formula as in listing 5.3a then translates into a for-loop with a helper variable v as in listing 5.3b.

```
assume (∃i : α ≤ i < β ∧ φ(i));
```

a: In First Order Logic

```
int v = 0;
for (int i = α; i < β; i++)
    v = v || φ(i);
assume (v);
```

b: Formula in C Code

Listing 5.3: Exemplary Translation of Existential Formula (a) into C Code (b) for LLBMC

As described in Section 3.2.2, LLBMC can check for a great variety of low-level bugs, which are rather technical and mostly C specific. However, our focus is not the C implementation itself, but the correctness of the abstract voting scheme, where the major part of these checks is only of minor concern regarding the properties we want to prove. As these checks impede the performance of the overall verification process, we only check them for some small inputs and from then on assume our program to pass all these tests successfully. Thus, we use the LLBMC option "`only-custom-assertions`" in order to omit all these checks for our experiments. Furthermore, the two voting schemes instant-runoff voting (the script which we used can be found in Listing A.11 in Appendix A) and single transferable vote Listing A.16 in Appendix A) both include a rather complex `while`-loop, of which an upper bound for the number of loop iterations cannot be easily determined. We hence specify an upper bound n for these two voting schemes calculated by multiplying the number of candidates C by the number of seats S (for IRV, S is equal to one). This enables us to omit the according check with the option "`no-max-loop-iterations-checks`", which again speeds up the performance of our experiments.
Another measure in order to reduce verification time is the choice of the SMT solver as a back end. For most examined voting schemes, the default option STP with MiniSat outperforms the other solvers. In the case of plurality voting, experimental testing has shown that using STP with CryptoMiniSat (by using the option "`smt-solver=stp-cms`") and furthermore eagerly encoding array-read axioms in STP (with the option "`stp-eager-`

`read-axioms`") is the better option (the scripts which we used for plurality voting and approval voting can be found in Listing A.4 and Listing A.8 respectively in Appendix A).

5.2. Plurality Voting

For plurality voting, the method for the main entry point is the one shown in Listing 5.1 on page 52. Each voter has exactly one vote for one candidate of his or her choice and thus for a ballot box b, a voter i and a candidate j, $b_i = j$ means that voter i has cast a vote for candidate j. The correctness property we want to prove for this voting scheme is monotonicity, as defined in Definition 9 on page 37 for voting schemes, where each voter can only vote for one candidate. Therein we reason about two different ballot box inputs b and b' for plurality voting, in the defiition denoted by the function s. Hence, in the C implementation, we need to call the `voting`-method (Listing 5.5 on the facing page) with the actual voting scheme twice, each time with slightly different parameters `v1` and `v2` (corresponding to the abstract ballot boxes b and b') as specified in the `correctness`-method (Listing 5.4).
The specification of the monotonicity criterion in Listing 5.4 consists of the following precondition. We firstly (Lines 3 to 7) make well-formedness assumptions such that the implicitly quantified variables `m` and `n` denote a valid candidate and a valid voter respectively and all votes being cast in `v1` and `v2` are assumed to be valid candidates as well. Secondly (Lines 8 to 11), we determine the relationship between `v1` and `v2` such that all votes by voters other than voter n are the same in both sets of votes (Line 8) and voter n does not vote for candidate m in `v1`, but does so in `v2` (Line 11). The postcondition finally states for the case that candidate m is elected in the election with votes `v1` that m gets also elected with the votes in `v2` (Line 13).

```
void correctness(uint v1[V], uint v2[V],
                 uint m, uint n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);
    for (uint i = 0; i < V; i++) {
        assume (0 < v1[i] && v1[i] <= C);
        assume (0 < v2[i] && v2[i] <= C);  } Precondition
        if (i != n)
            assume (v1[i] == v2[i]);
    }
    assume (v1[n] != m && v2[n] == m);
    uint elect1 = voting(v1), elect2 = voting(v2);
    if (m == elect1)
        assert (elect1 == elect2);  } Postcondition
}
```

Listing 5.4: Monotonicity Specification for Plurality Voting

The method for plurality voting itself in Listing 5.5 contains the declaration and initialisation of an auxiliary array `res` in order to store the amounts of votes for each candidate (Lines 2 and 3). The main part of the program then consists of two for-loops. In the first loop in Listing 5.5 (Line 4), all the votes are tallied for each candidate and stored in the auxiliary array `res`. The second loop (Line 6) finally determines the maximum amount of votes for one candidate and also the according candidate. If this maximum is unique, the result is the according candidate. Otherwise, the result is zero.

```
1  uint voting(uint votes[V]) {
2      uint res[C + 1];
3      for (uint i = 0; i <= C; i++) res[i] = 0;
4      for (uint i = 0; i < V; i++) res[votes[i]]++;
5      uint max = 0, elect = 0;
6      for (uint i = 1; i <= C; i++) {
7          if (max < res[i]) {
8              max = res[i];
9              elect = i;
10         } else if (max == res[i]) elect = 0;
11     }
12     return elect;
13 }
```

Listing 5.5: Plurality Voting

Results For this implementation (the complete specification can be found in Listing A.5 in Appendix A), LLBMC is able to show that plurality voting satisfies the monotonicity criterion for up to 15 candidates and 10 voters. The run-times of the tool depending on the number of voters and candidates are shown in Fig. 5.1 (separate curves for 3, 6, 9, 12, and 15 candidates) and Fig. 5.2 (separate curves for 2, 4, 6, 8, and 10 voters) respectively. The graphs indicate at least an exponential time complexity in the number of voters (Fig. 5.1) and an at most quadratic time complexity in the number of candidates (Fig. 5.2).
In general, the reasons for the results can be based on the length of the execution paths, the number of possible symbolic execution paths, the complexity of program operations or combinations of those. Considering the length of the execution path, plurality voting has linear complexity in the numbers of voters and candidates, and the invoking `correctness`-method has linear complexity in the number of voters as well as two calls of the actual voting scheme, which leads to a linear overall complexity. When we also analyse the complexity with respect to the possible number of execution paths, the potential number of different ballot box instances is C^V. We initialise two ballot boxes, which have a very close relation to each other and differ in only one vote, precisely determined by the parameters m and n. These two parameters are symbolic representations of a random candidate and a random voter, effectively multiplying the total complexity by the number of voters V and candidates C. This hence computes to a total number of execution paths of $C * V * C^V$.
When considering the complexity of program operations, we essentially find numerical comparisons, boolean operations and array read and write operations. We suspect the array write operations to have the biggest impact, which mainly depends on the number of voters. However, as LLBMC uses a bit-precise memory model, the number of potentially written bits per array write operation depends on the number of candidates. In conclusion, the analysis about the number of possible execution paths, which we justify to be asymptotically $V * C^{V+1}$, comes very close to the results of our measurements and also indicates a big impact of the number of voters.

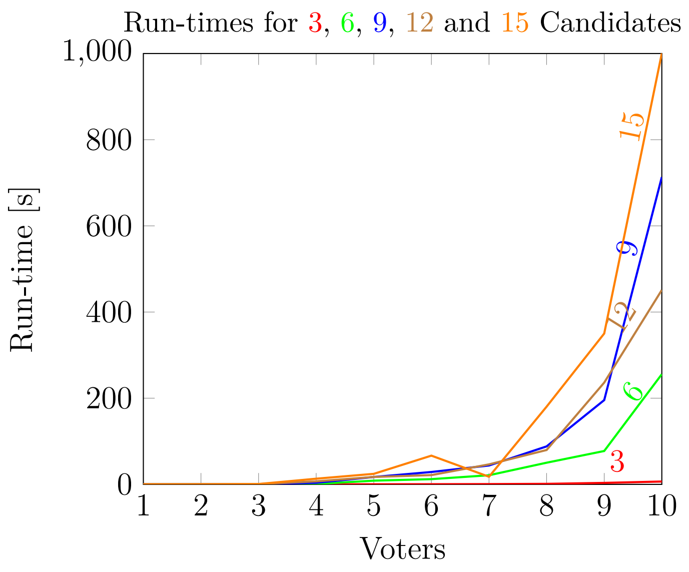

Figure 5.1.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting per Voter

Furthermore, our experiments show unusual deviations with respect to the number of candidates for the second graph in Fig. 5.2. Therein, we notice a significant oscillation of run-time for different numbers of candidates. Unfortunately, we did not get justifications from the experiments for this surprising behaviour yet and this result gives rise to further investigations on this issue for future work.

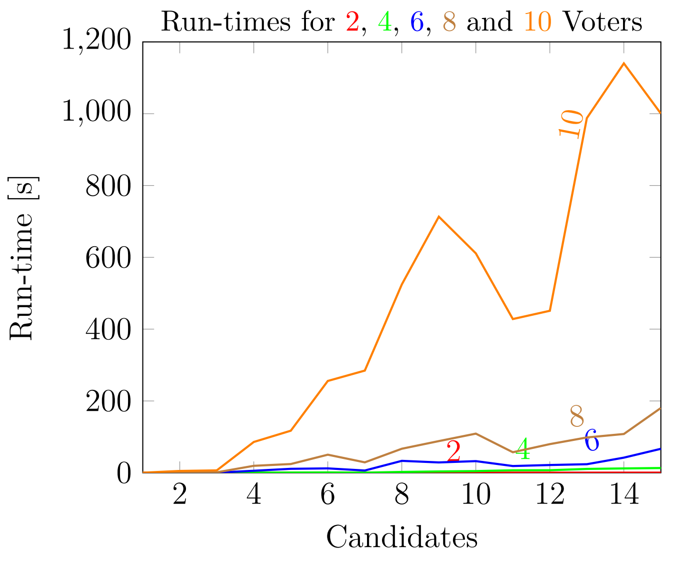

Figure 5.2.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting per Candidate

5.3. Approval Voting

As for approval voting, the method at the main entry point is slightly different to Listing 5.1, but only in the data type for the sets of votes `v1` and `v2`. In this case, these are two-dimensional arrays with the voters as the first dimension and the candidates as the second dimension. Each voter can either approve or disapprove of each candidate separately and thus for a ballot box b, a voter i and two candidates j and k, $b_{i,j} = 1$ means that voter i approves of candidate j and $b_{i,k} = 0$ that voter i disapproves of candidate k. The correctness property we want to prove for this voting scheme is again monotonicity. Particularly referring to approval voting, we can adapt the definition of monotonicity as follows (with 1 and 0 corresponding to *true* and *false* respectively):

Definition 10 (Monotonicity for Approval Voting) *For a set of candidates C, a set of voters V and a set of ballot boxes B, we call an* approval voting scheme *s monotone iff*

$$\begin{aligned}
&\forall c \in C, v \in V, b, b' \in B : \\
&\qquad (\neg b_{v,c} \wedge {b'}_{v,c} \wedge \forall i \in V, j \in C : (i \neq v \vee j \neq c) \Rightarrow b_{i,j} = {b'}_{i,j}) \\
&\qquad\qquad \Rightarrow (s(b) = c \Rightarrow s(b') = c).
\end{aligned}$$

Thereto we need to reason about two different ballot box inputs b and b' for approval voting, the voting function is denoted by the function s, and hence need to call the `voting`-method with the actual voting scheme twice, each time with slightly different parameters `v1` and `v2` as specified in the `correctness`-method (Listing 5.6).
The specification of the monotonicity criterion in Listing 5.6 goes along the lines of the specification for plurality voting, except for the fact that it deals with a different data type for the sets of votes. The well-formedness assumptions only differ with respect to the ones for plurality voting in the values and the dimensions for the ballot boxes `v1` and `v2`. In Lines 7 and 8 of Listing 5.6, we determine the values to be either zero or one, and iterate over both the numbers of voters and candidates.
Furthermore (Lines 9 to 13), we determine the relationship between `v1` and `v2` such that all votes by voters other than the (dis-)approval of voter n for candidate m are the same in both sets of votes (Line 9) and voter n does not approve of candidate m in `v1`, but does so in `v2` (Line 13). The following postcondition reads exactly as the one of plurality voting.

```
for (uint i = 0; i < V; i++) {
    for (uint j = 0; j < C; j++) {
        assume (0 <= v1[i][j] && v1[i][j] <= 1);
        assume (0 <= v2[i][j] && v2[i][j] <= 1);
        if ((i != n) || (j != (m-1)))
            assume (v1[i][j] == v2[i][j]);
    }
}
assume (v1[n][m-1] == 0 && v2[n][m-1] == 1);
```

Listing 5.6: Monotonicity Precondition for Approval Voting

Compared to plurality voting, the only difference in the procedure for the voting scheme itself is the vote tallying. Here, we have a nested loop, iterating over both the numbers of voters and candidates and summing up the votes for each candidate. The rest of the voting scheme is exactly the same as for plurality voting, as we are looking for the candidate with a unique maximum of approvals, and we omit the listing at this point.

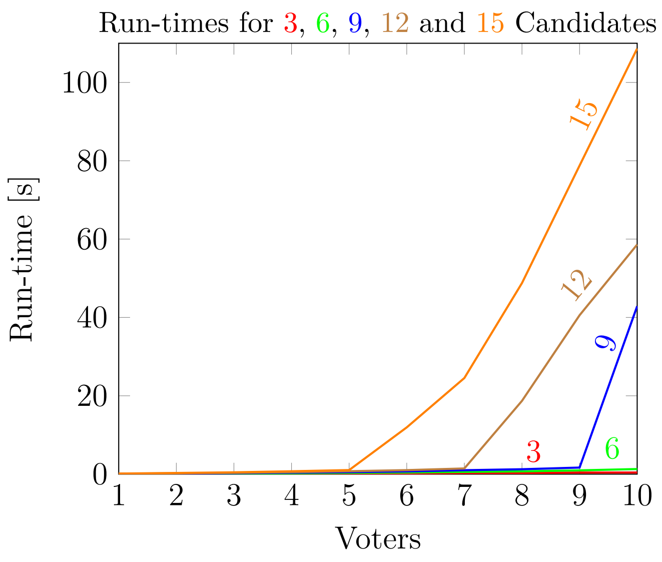

Figure 5.3.: LLBMC Verification Performance with Monotonicity Criterion for Approval Voting per Voter

Results For this implementation (the complete specification can be found in Listing A.9 in Appendix A), LLBMC is able to show that approval voting satisfies the monotonicity criterion for up to 15 candidates and 22 voters. The run-times of the tool depending on the number of voters and candidates are shown in Fig. 5.3 (separate curves for 3, 6, 9, 12, and 15 candidates) and Fig. 5.4 (separate curves for 2, 4, 6, 8, and 10 voters) respectively. The graphs indicate an exponential time complexity in the number of voters (Fig. 5.1) and also an exponential time complexity in the number of candidates (Fig. 5.2). Analysing the number of possible execution paths similarly to the analysis of plurality voting, we get a number of possible ballot boxes of 2^{C*V} and hence a total number of execution paths of $C * V * 2^{C*V}$. The execution path length hence computes to $C * V$. Consequently, this explains the exponential complexity in the numbers of candidates and voters with the number of possible execution paths.

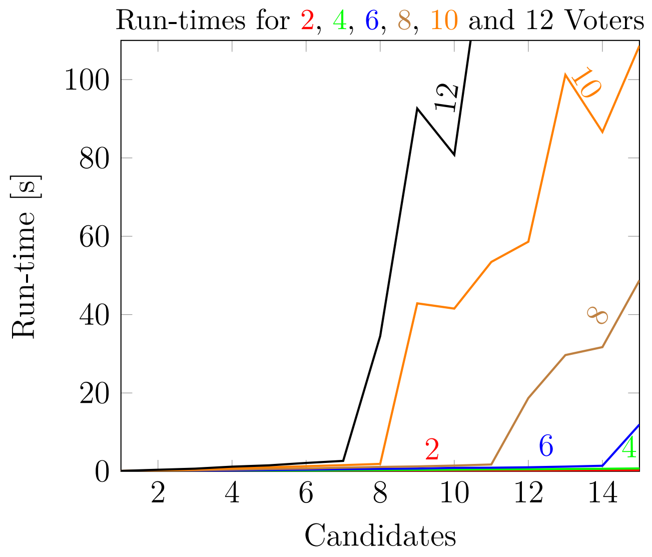

Figure 5.4.: LLBMC Verification Performance with Monotonicity Criterion for Approval Voting per Candidate

Similar to the experiments for plurality voting, these experiments also show some unusual deviations with respect to the number of candidates for the second graph in Fig. 5.4. This time, the run-time oscillation for different numbers of candidates is not as significant as for plurality voting. Again, this observation needs further examination as future work. Compared to plurality voting, the results for approval voting are better in performance even though the voting scheme is more complex. We suspect this to be based on the amount of possible values in the ballot boxes as LLBMC uses a bit-precise memory model [MFS12], which is more efficient for small numbers. Whereas the votes for plurality voting are numbers representing valid candidates, for approval voting we have one more dimension, but only two possible values expressing either approval or disapproval for the according candidate.

5.4. Instant-Runoff Voting

In the case of instant-runoff voting (IRV), the analysis grows in complexity compared to the previous voting schemes as the voting scheme itself is significantly more complex. One reason for the higher complexity is that we now have preferential votes, i.e. each voter can cast his or her vote as a ranked list of candidates. This means for a ballot box b, a voter v, a candidate c and a rank position p, $b_{v,p} = c$ means that voter v has ranked candidate c on his or her preferential vote on position p. We also allow a voter

to not cast a complete ranked list of all candidates, but to rank only a part of them. In this case, $b_{v,p} = 0$ means voter v has not ranked any candidate on position p. We assume a voter does not leave any gaps between ranked candidates, i.e. for any p with $b_{v,p} = 0$ that for any rank position p' with $p' \geq p$, $b_{v,p'} = 0$ holds as well. Furthermore, we assume that each voter can rank each candidate only once. This also has the effect that for all rank positions p, we have $1 \leq p \leq C$. We also do not allow empty votes, i.e. for all voters v, we assume $b_{v,1} \neq 0$. Another reason for the more complex voting scheme is that instant-runoff voting simulates a number of election rounds, thereby it deletes candidates in a specified manner from the ballot box if a winner cannot be determined in the current round.
As instant-runoff voting does not comply with the monotonicity criterion, instead we analyse it with respect to the Condorcet loser criterion (Definition 8 on page 22), which is well-known to hold for instant-runoff voting [Fel12b] and in the following defined for this voting scheme:

Definition 11 (Condorcet Loser Criterion for Preferential Voting) *For a set of candidates C, a set of voters V, a set of preferences P and a set of ballot boxes B, we say a* single-winner preferential voting scheme *s satisfies the Condorcet loser criterion iff*

$$\begin{aligned}
&\forall b \in B, c \in C : \\
&\quad (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid \exists p \in P : b_{v,p} = c \\
&\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p' \ \wedge\ b_{v,p'} = k) \ \vee\ \forall p' \in P : b_{v,p'} \neq k)\} \\
&\qquad\qquad < \#\{v \in V \mid \exists p \in P : b_{v,p} = k \\
&\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p' \ \wedge\ b_{v,p'} = c) \ \vee\ \forall p' \in P : b_{v,p'} \neq c)\}) \\
&\qquad\qquad\qquad\qquad \Rightarrow s(b) \neq c
\end{aligned}$$

In Definition 11, we use “$\#\{\dots\}$” to denote the cardinality of the set, which is described in the curly braces by set comprehension. For the Condorcet loser criterion, we only need to reason about one ballot box input b for the instant-runoff voting scheme s and hence, in the implementation, need to call the `voting`-method with the actual voting scheme only once. However, the definition of the Condorcet loser criterion in Definition 11 compares the cardinality of two different sets with each other and hence we need to translate this into C code in order to verify with LLBMC.
Additionally to the specification of this correctness property (Definition 11), we also need a number of well-formedness assumptions for ballot boxes in preferential voting schemes. Therein, we assume there are no empty votes, all preferences of a vote either contain a valid candidate or are empty, in any preference list each candidate can appear only once, and if any preference rank in a vote is empty, then all subsequent ranks in this vote are also empty, leaving no gap in a preference list. These are stated in Definition 12 and translated into C code as explained in the beginning of this chapter.

Definition 12 (Well-Formedness Conditions for Preferential Votes) *For a set of candidates C, a set of voters V and a set of preferences P, we call a preferential ballot box b* well-formed *iff*

$$\begin{aligned}\forall v \in V : b_{v,1} \in C \\ &\wedge \forall p \in P : (b_{v,p} \in C \vee b_{v,p} = \bot) \\ &\quad \wedge \forall p' \in P : (b_{v,p} \in C \wedge p \neq p' \Rightarrow b_{v,p} \neq b_{v,p'}) \\ &\qquad \wedge (b_{v,p} = \bot \wedge p \leq p' \Rightarrow b_{v,p'} = \bot)\end{aligned}$$

Furthermore for the Condorcet loser criterion (Definition 11), the actual correctness property we want to verify, we need a symbolic value for a valid candidate representing the Condorcet loser. We assume this value to represent a valid candidate and translate the definition of the criterion in Definition 11 to C code according to the described translation methods in Section 5.1. For this matter we need to apply a combination of the methods for universally quantified formulas in Listing 5.2 and for existentially quantified formulas in Listing 5.3 as well as a variation for the computation of the set cardinality. The set cardinality can be translated very similarly to a universally quantified formula with the only difference that instead of assuming or asserting the program corresponding to the formula to hold, we use a helping numerical variable, which is incremented each time the formula holds. The remaining translation is a composition of the methods in Section 5.1. Thereby, we precisely describe the Condorcet loser for a given ballot box `votes` with V valid votes and C valid candidates and also finalise the precondition of the Condorcet loser criterion. Now we simply call the actual voting scheme and assert as a postcondition that the Condorcet loser, represented by the symbolic value described in the precondition, is never elected by the voting scheme. In the following, we explain the implementation of the two variations *exhaustive instant-runoff voting* and *deterministic instant-runoff voting* concerning the tie-breaking procedure as described in Section 2.3.3.

```
min = quota;
weakest = 0;
for (i = 1; i <= C; i++) {
    if (count[i] < min && count[i] != 0) {
        min = count[i];
        weakest = 1;
    } else if (count[i] == min) weakest++;
}
choice = __llbmc_nondef_unsigned_int();
assume (0 < choice && choice <= weakest);
weakest = 0;
for (i = 1; i <= C; i++) {
    if (count[i] == min) weakest++;
    if (count[i] == min && weakest == choice) {
        // delete candidate i from array v
        // (as in the delete subroutine).
    }
}
```

Listing 5.7: Repetitive Step in Instant-Runoff Voting

Breaking Ties (Two Variations) The C implementation of the voting scheme itself is very close to the algorithm for instant-runoff voting given in Algorithm 2.4 with the subroutine for candidate deletion from Algorithm 2.3 being inlined where it is called. However, the implementation of the tie-breaking procedure needs some clarification. This procedure consists of the computation of the set of equally weakest candidates, the determination of the minimum number of first preferences a candidate currently has, and the abstract function `chooseCandidate` as in Algorithm 2.4. We compute the amounts of first preferences for every continuing (i.e. not yet eliminated) candidate in the array `count`, of the same size as there are valid candidates (for this array, we do not use the entry 0 as to avoid confusion), in order to compare them to the quota (not shown in Listing 5.7). Hence, we also have access to `count` when we need to compute the set of equally weakest candidates. We also assume the quota to be calculated beforehand and stored in the variable `quota`.
Having made these assumptions, for standard instant-runoff voting (i.e. *exhaustive instant-runoff voting*, partly shown in Listing 5.7) we start by setting the variable `min`, which contains the minimum number of first preferences a candidate currently has, to the value of `quota` and `weakest`, which contains the number of currently equally weakest candidates, to zero. We then iterate over the number of candidates and look for the minimum by comparing each positive value in `count` to the current value of `min`, which cannot be greater than the quota, because otherwise he or she would have already won the election. In the same iteration, we count the number of these equally weakest candidates in `weakest` by setting it to one when finding a new minimum and otherwise in case the

value in `count` for the currently observed candidate is equal to `min`, increase `weakest` by one. By performing this procedure, we find out the minimum amount of first preferences for a candidate as well as the number of these candidates. Subsequently, we introduce a numerical variable `choice` by initialising it randomly with the LLBMC primitive "`__llbmc_nondef_unsigned_int()`" and assume it to denote a valid candidate, but not a higher number than declared by `weakest`. At this point, there are as many options and hence possible paths for the variable `choice` as there are equally weakest candidates. Now, we still have to find out the actual equally weakest candidates, as until now we only have their amount. This can be done by resetting `weakest` to zero and iterating again over all candidates, while therein incrementing `weakest` by one and calling the delete procedure in case we reached the actually by `choice` chosen candidate. We subsequently simply delete this candidate from the ballot box and redo the subroutine `delete` for the chosen candidate.
Alternatively, the implementation of *deterministic instant-runoff voting* (the complete specification can be found in Listing A.12 and Listing A.14 in Appendix A) starts off similarly, i.e. it iterates over the whole set of candidates and searches for the set of weakest candidates. But instead of using a randomly initialised variable in order to choose one of them, we simply delete all of them from the ballot box. Hence, we follow only one execution branch and possibly delete more than one candidate in one round, applying the (inlined) subroutine `delete` to each of them. This version differs from standard instant-runoff voting and this presumably small change also affects the compliance with formal criteria for voting schemes. It notably does not comply with the Condorcet loser criterion anymore, which can be seen in the following example.

Counterexample for the Condorcet Loser Criterion with *Deterministic* IRV Let us consider an election with three candidates A, B, C, and seven voters with the following votes being cast:

2 Votes	2 Votes	2 Votes	1 Votes
A	B	C	C
B	A	A	B
C	C	B	A

Table 5.1.: Exemplary Aggregated Ballot Box for Instant-Runoff Voting

In order to determine the Condorcet loser, we need to count all head-to-head matches. Comparing candidates A and B, A wins 4 and B wins 3 matches. For A and C, A wins 4 and C wins 3 matches, and for B and C, B wins 4 and C wins 3 matches. Hence, candidate A wins all head-to-head matches and candidate C loses all head-to-head matches with other candidates. This makes A the Condorcet winner and candidate C the Condorcet loser.
Moreover, we examine an election with deterministic instant-runoff voting. As there are 7 voters, the quota is 4 in order to elect one candidate. Considering the first preferences, no candidate has enough votes to reach this quota. Therefore, we need to eliminate the

weakest candidates, which are the candidates A and B. Eliminating these two candidates thus means electing candidate C, who is the Condorcet loser, and this voting scheme hence violates the Condorcet loser criterion.

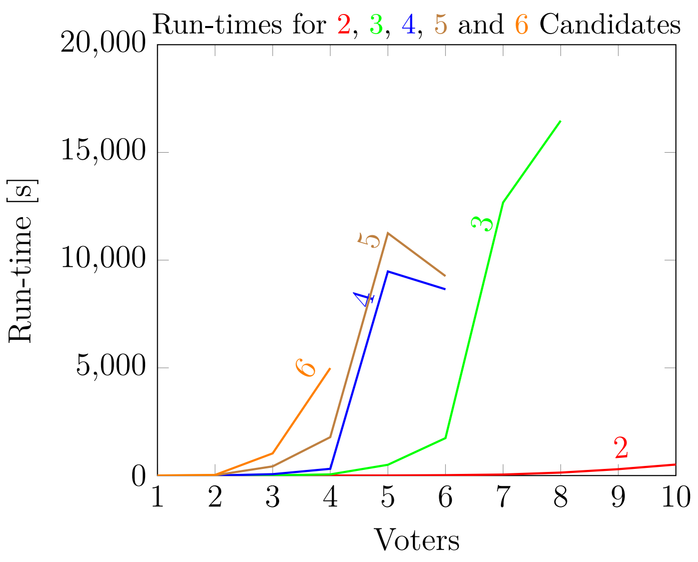

Figure 5.5.: LLBMC Verification Performance with Condorcet Loser Criterion for Instant-Runoff Voting per Voter

Results For the implementation of standard (or *exhaustive*) instant-runoff voting (the complete specification can be found in Listing A.13 and Listing A.14 in Appendix A), LLBMC is able to show that instant-runoff voting satisfies the Condorcet loser criterion for up to 3 candidates and 5 voters or 2 candidates and 10 voters before the timeout of 1500 seconds. The run-times of the tool depending on the number of voters and candidates are shown in Fig. 5.5 (separate curves for 2, 3, 4, 5, and 6 candidates) and Fig. 5.6 (separate curves for 2, 4, 6, and 8 voters) respectively, wherein also results up to 16472 seconds are shown for a better comparison, as the experiments within the original timeout of 1500 seconds do not lead to a sufficient amount of results. The graphs indicate an exponential time complexity in the number of voters (Fig. 5.5) and the same behaviour in the number of candidates (Fig. 5.6). As we have seen in the previous experiments, the verification performance appears to be largely dependant on the number of possible execution paths or ballot boxes in the case of voting schemes. For a preferential voting schemes as instant-runoff voting, this is very problematic as the number of different ballot boxes (and hence also the number of execution paths) asymptotically exceeds $2^V * C!^V$ whereas the length of the whole execution path is only $V * C^2$. Compared to the previous two voting schemes and the considered properties, plurality voting and approval voting with the monotonicity criterion, the verification performance for instant-runoff voting with the Condorcet loser criterion is much worse and the timeout is reached for even smaller numbers, hampering a comprehensive analysis.

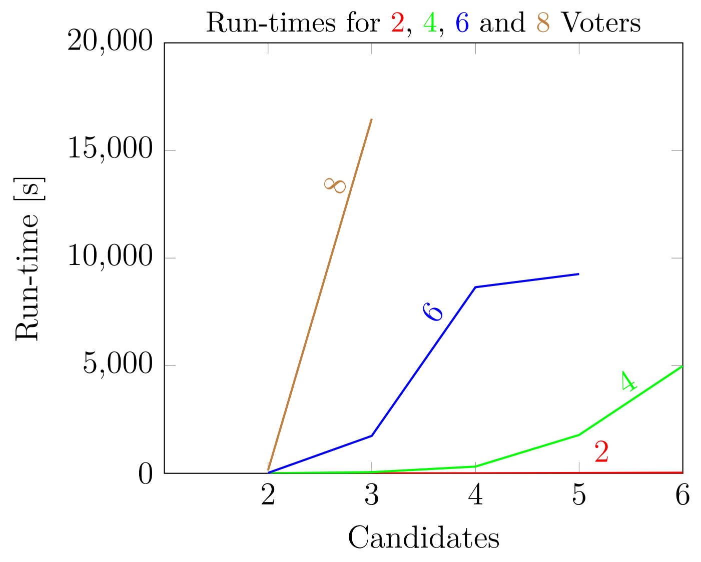

Figure 5.6.: LLBMC Verification Performance with Condorcet Loser Criterion for Instant-Runoff Voting per Candidate

5.5. Single Transferable Vote

The voting scheme *single transferable vote* is largely based on *instant-runoff voting* (Section 5.4 on page 60) and is equivalent if there is only one vacancy. If, however, there are more vacancies to fill, single transferable vote differs in the following aspects from instant-runoff voting:

The first difference is the distribution of surplus votes of a candidate who is elected (Listing 5.8). Therein, the variable `e` denotes a counting variable for the already elected candidates and `res` denotes the currently elected candidate to be stored in the output array `r`. For this matter, all votes exceeding the number of votes needed to fulfil the necessary quota for a candidate a to be elected are being redistributed to other candidates according to the second preferences. This can be done in various ways and in the real-world scenario, there are different implementations for the redistribution of surplus votes. To simplify matters, we simply take the first Q votes having the previously elected candidate a as their first preference. Naturally, this results in the voting scheme lacking *anonymity*, i.e. in this implementation the order of the votes may matter. There are also methods ensuring *anonymity* as e.g. transferring fractional votes. However, many existing versions of STV proceed by either randomly choosing the votes to be redistributed, by also taking either the first or last Q votes in a predefined order, or by combining some of these methods. This is usually based on the time-consuming process of manual vote counting once there are fractional votes involved. Taking into account that our experiments do not involve fixed sets of votes, but rather symbolic values with predetermined bounds and hence cover all possible orders of votes, not using fractional votes is justified as we do not lose any possible execution paths.

```
if (res != 0) {
    r[e++] = res;
    for (t = 0; t <= quota; t++) {
        for (i = 0; votes[i][1] != res; i++);
        for (j = 1; j <= C; j++) votes[i][j] = 0;
    }
    // delete candidate "res" from ballot box "votes",
    // set "res" to zero and decrement "cc" by one.
}
```

Listing 5.8: Vote Redistribution for Single Transferable Vote

The second difference to instant-runoff voting is the election of candidates in case the quota is too high to fill all available seats (5.9). In this case, we simply elect remaining candidates if there is a first preference for this candidate and delete all preferences for this candidate each time subsequently. We do this until all vacancies are filled or there are no more candidates who received votes.

```
if (e < S - 1) {
    for (i = e; i < S && 0 < cc; i++) {
        for (res = 0, k = 1; k <= C && res == 0; k++)
            for (j = 0; j < V && res == 0; j++)
                if (votes[j][1] == k) res = k;
        r[i] = res;
        // delete candidate "res" from ballot box "votes"
        // and decrement "cc" by one.
    }
}
```

Listing 5.9: Allocation of Remaining Seats for Single Transferable Vote

Having one more parameter and hence one more dimension compared to instant-runoff voting, we also need to adjust our correctness criterion for STV, i.e. we need to modify the Condorcet loser criterion for voting schemes with multiple seats. Originally, the Condorcet loser criterion is not defined for voting schemes wherein multiple candidates are elected. We leave the precondition as it was for instant-runoff voting, because there is only one Condorcet loser, whose definition does not change if we elect more candidates. However, we need to modify the postcondition and thus propose the one in Definition 13:

Definition 13 (Condorcet Loser Criterion for Multi-Seat Pref. Voting) *For a set of candidates C, a set of voters V and a set of ballot boxes B, we say a* multi-winner preferential voting scheme *s satisfies the Condorcet loser criterion iff*

$$\begin{aligned} &\forall b \in B, c \in C : \\ &\quad (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid (\exists p, p' \in P : p' < p \ \wedge \ b_{v,p} = c \ \wedge \ b_{v,p'} = k)\} \\ &\qquad\qquad < \#\{v \in V \mid (\exists p, p' \in P : p < p' \ \wedge \ b_{v,p} = c \ \wedge \ b_{v,p'} = k)\}) \\ &\qquad\qquad\qquad \Rightarrow (|s(b)| = |C| \vee c \notin s(b)) \end{aligned}$$

Translated to an assertion for LLBMC, we get the postcondition in Listing 5.10. Therein, `elect` denotes the array of elected candidates, which is returned by the voting scheme function. The variable `loser` denotes the Condorcet loser, which is computed in the beginning of the implementation.

```
uint *elect = voting(votes);
for (i = 0; i < S; i++)
    assert (C == S || elect[i] != loser);
```

Listing 5.10: Postcondition of Condorcet Loser Specification for STV

Therein, we assert every elected candidate to differ from the Condorcet loser, except if the number of candidates is equal to the number of seats. As we assumed in our precondition that there can be no more candidates than available seats, this condition is sufficient.

Results For this implementation (the complete specification can be found in Listing A.17 and Listing A.18 in Appendix A), LLBMC quickly needs more than 4 GB of memory and inputs of three candidates, two seats and five voters or more do not return any results.

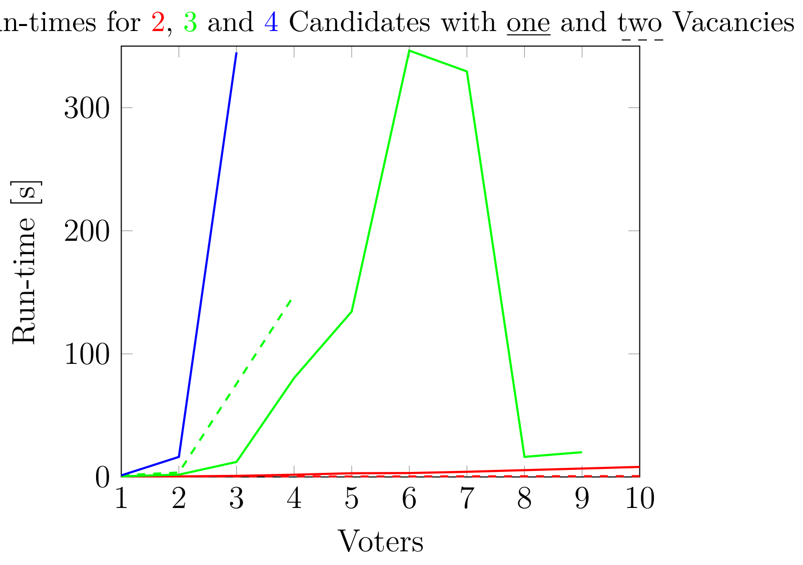

Figure 5.7.: LLBMC Verification Performance with Condorcet Loser Criterion for STV per Voter

The performance results for all successful proofs depending on the number of voters and candidates can be seen in Fig. 5.7 (separate curves for 2, 3, 4, 5, and 6 candidates) and Fig. 5.8 (separate curves for 2, 4, 6, and 8 voters) respectively. Furthermore, we observe that for some numbers with one vacancy and the same numbers for candidates and voters, the performance is better than for instant-runoff voting, which is exactly the same voting scheme when only one candidate gets elected. In our experiment, this appears for increasing numbers of voters, e.g. for 2 candidates and 8 voters the verification takes 138 seconds for instant-runoff voting and 5 seconds for STV, even though the generated formula is bigger (30398 expressions for IRV and 47614 expressions for STV). However, for some numbers the verification performance for instant-runoff voting is better than for STV, e.g. for 4 candidates and 3 voters this is 64 seconds and a formula with 66246 expressions for IRV and 344 seconds and a formula with 99189 expressions for STV. As we have only a small set of numbers for comparison, we do not have an explanation for the different performance for one vacancy. The reason that LLBMC much earlier exceeds the available memory can be explained by the size of the generated formula for STV as the number of possible execution paths is equal for both voting schemes when S is equal to one.

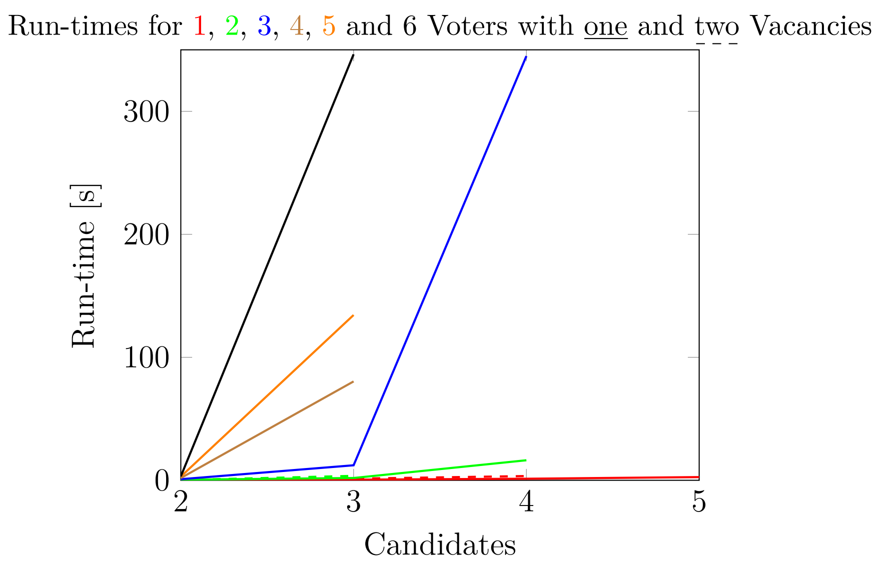

Figure 5.8.: LLBMC Verification Performance with Condorcet Loser Criterion for STV per Candidate

Counterexample for the Condorcet Loser Criterion with STV However, LLBMC is able to create the following counterexample for an input of three candidates, two seats and three voters in 6.4 seconds:
In order to rule out bugs in our implementation or our formalisation of the Condorcet loser criterion, we check that this is not a spurious counterexample. Firstly, we determine the Condorcet loser by comparing all head-to-head matches. Therein, candidate B wins against candidate A with two winning matches and one loss, candidate C wins against candidate A with two winning matches and one loss, and candidate B achieves a tie

Voter 1	Voter 2	Voter 3
B	C	A
C	B	
A		

Table 5.2.: Counterexample for Condorcet Loser Criterion with STV

of one to one with candidate C. This means there is no Condorcet winner, but the Condorcet loser is candidate A, since this candidate is defeated by every other candidate in head-to-head matches. Secondly, we examine possible outcomes of an election with single transferable vote considering this example as an input. As no candidate attains the necessary quota in the first preferences, we start by eliminating one candidate. However, all three candidates have the same number of first preferences and we consequently need to choose at random. Taking into account that we need to fill two seats, this shows already that any combination of two elected candidates is possible, since the voting scheme is designed in such a way that every seat is filled with a candidate. Thus, either the elimination of candidate B or the elimination of candidate C results in a seat for candidate A, who is the Condorcet loser, and consequently the Condorcet loser criterion is violated by this implementation of STV.

5.6. Summary

In the preceding part of this chapter, we analysed the four voting schemes plurality voting, approval voting, instant-runoff voting (IRV), and single transferable vote (STV) with the two correctness properties monotonicity criterion (for the former two voting schemes) and Condorcet loser criterion (for the latter two voting schemes). For this matter, we used the SBMC approach as to do automatic verification within some specified bounds, which were determined by the number of candidates, voters, and seats. In Fig. 5.9 and Fig. 5.10 The run-times of LLBMC depending on the number of voters and candidates are shown in Fig. 5.9 (separate curves for 3, 6, 9, 12, and 15 candidates) and Fig. 5.10 (separate curves for 2, 4, 6, 8, and 10 voters) respectively.

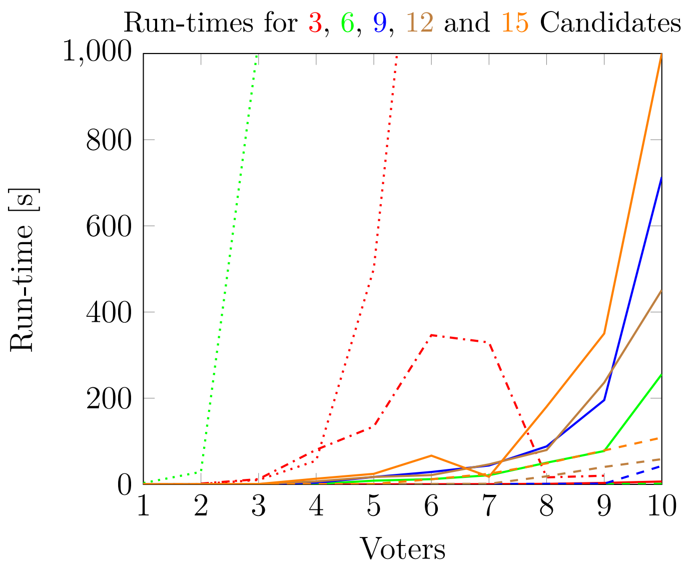

Figure 5.9.: LLBMC Verification Performance per Voter

Therein, the solid lines show the results for plurality voting, the dashed lines for approval voting, the dotted lines for IRV, and the dash-dotted lines show the results for STV with one vacancy. Clearly, the results for IRV show the highest run-times and the verification of the monotonicity criterion for approval voting outperforms all other results.

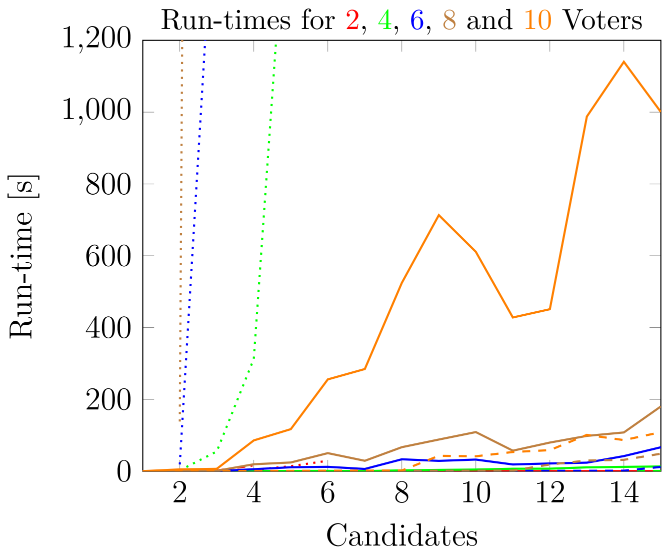

Figure 5.10.: LLBMC Verification Performance per Candidate

The results indicate the general feasibility of the approach, but show great variations between the different voting schemes and especially IRV and STV seem impractical for the SBMC approach. When comparing, STV with IRV, the run-times are surprisingly low for STV, but for bigger numbers, the available memory quickly impedes the experiments. We also learned in the experiments with STV that the compliance of a single-seat voting scheme with a correctness property does not easily imply the same for a similarly constructed multi-seat voting scheme. This observation gives rise to investigations on more specialised or tailor-made properties for such voting schemes.

5.7. Generation of Counterexamples

As we have seen in Section 5.5, LLBMC can produce precise counterexamples for assertions which do not hold. Additionally to bounded correctness proofs, this is a strength of the SBMC approach. Although this is often only possible for small numbers, we base our justification for its validity on the *small scope hypothesis* (Section 3.1.2) and argue that a high proportion of bugs can be found for test inputs within some small scope and it is unlikely that a bug occurs only for high numbers. In the following, we examine the two complex voting schemes instant-runoff voting and single transferable vote with the Condorcet criterion as correctness property. Both voting schemes are well-known to violate the Condorcet criterion [Fel12b] and we hence explicitly want to analyse the disproof of this formal property and generate counterexamples.

5.7.1. Instant-Runoff Voting

In this experiment, we want to disprove for our implementation of instant-runoff voting that the Condorcet winner is always elected if a Condorcet winner exists. As in the previous experiments, we assume a well-formed preferential ballot box as in Definition 12 on page 62. Based on the general Condorcet criterion (Definition 7 on page 21) and very similarly to the Condorcet loser criterion (general version in Definition 8 on page 22 and in Definition 11 on page 61 for preferential voting schemes), we formulate the Condorcet criterion for instant-runoff voting as follows:

Definition 14 (Condorcet Criterion for Preferential Voting) *For a set of candidates C, a set of voters V, a set of preferences P and a set of ballot boxes B, we say a* single-winner preferential voting scheme *s satisfies the Condorcet winner criterion or Condorcet criterion iff*

$$\begin{aligned}
&\forall b \in B, c \in C: \\
&\quad (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid \exists p \in P : b_{v,p} = k \\
&\qquad\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p' \ \wedge\ b_{v,p'} = c)\ \vee\ \forall p' \in P : b_{v,p'} \neq c)\} \\
&\qquad\qquad < \#\{v \in V \mid \exists p \in P : b_{v,p} = c \\
&\qquad\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p' \ \wedge\ b_{v,p'} = k)\ \vee\ \forall p' \in P : b_{v,p'} \neq k)\}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow s(b) = c
\end{aligned}$$

We perform the translation to C code along the lines of the previous sections as explained in Section 5.1 and performed in Section 5.4, and embed this implementation of instant-runoff voting in our usual experiment setup as in Section 5.4.

When running LLBMC on it, we observe that the smallest counterexample can be generated in 2.6 seconds for three candidates and three voters and within our timeout we find the biggest counterexample in 776 seconds for four candidates and 19 voters. In the following, we present the smallest counterexample and demonstrate its validity in order to justify our approach.

Counterexample for the Condorcet Criterion with IRV Let us take three candidates A, B and C, and the following three votes:

Voter 1	Voter 2	Voter 3
A	B	C
C	C	
B	A	

Table 5.3.: Counterexample for Condorcet Criterion with Instant-Runoff Voting

We need to check that this is not a spurious counterexample. Firstly, we determine the Condorcet winner by comparing all head-to-head matches. Therein, candidate C wins against candidate A with two winning matches and one loss, candidate C wins against candidate B with two winning matches and one loss, and candidate A achieves a tie of one to one with candidate B. This means there is no Condorcet loser, but the Condorcet winner is candidate C, since this candidate is defeated by every other candidate in head-to-head matches. Secondly, we examine possible outcomes of an election with instant-runoff voting considering this example as an input. As no candidate attains the necessary quota in the first preferences, we start by eliminating one candidate. However, all three candidates have the same number of first preferences and we consequently need to choose at random. Taking into account that we can also eliminate candidate C, this shows already that there is an outcome where the Condorcet winner, who is candidate C, is not elected. Thus, we have a violation with the Condorcet criterion for this implementation (the complete specification can be found in Listing A.13 and Listing A.15 in Appendix A) of instant-runoff voting and have a valid counterexample.

Results As expected, finding a counterexample is significantly faster than doing a bounded proof as we were not able to prove the Condorcet loser criterion for four candidates and more than 6 voters within 1500 seconds, but found a counterexample for the Condorcet winner criterion for four candidates and 8 voters in only 77 seconds. As we can see in Fig. 5.11, we find many counterexamples in less than 500 seconds and compared to the number of possible ballot boxes of more than $2^V * C!^V$, the run-times seem much faster. The results indicate an exponential run-time complexity in the number of voters (Fig. 5.11) and an at most quadratic run-time complexity in the number of candidates (Fig. 5.12) However, as we only have results for 3, 4 and 5 candidates, the data pool for comparing different numbers of candidates is too small for a meaningful justification of a quadratic complexity.

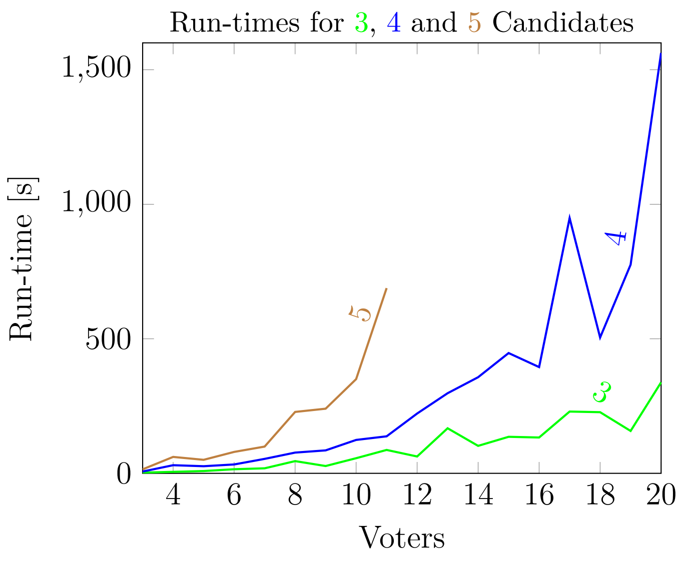

Figure 5.11.: LLBMC Performance of Counterexample Generation with Condorcet Criterion for Instant-Runoff Voting per Voter

In Fig. 5.11, we also observe occasionally faster run-times for higher numbers of voters. However, this is not very surprising when looking for counterexamples as we do not need to make a proof for the whole search space, but instead only need to find one counterexample for the disproof. Assuming a higher number of counterexamples for higher numbers of voters, a better performance for some higher numbers appears likely at times.

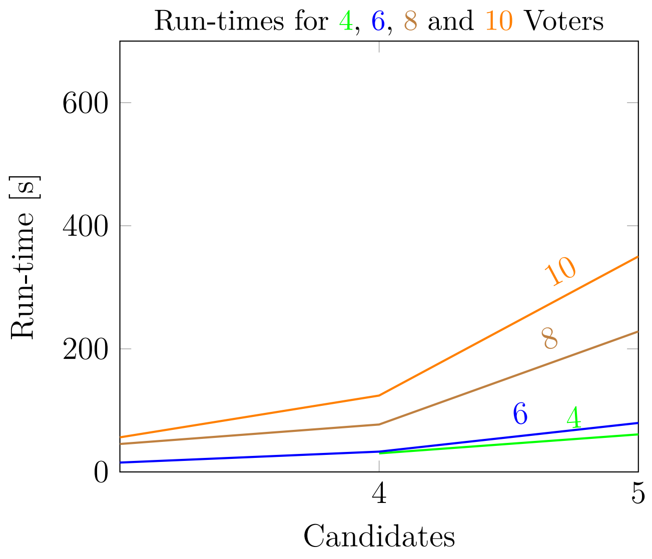

Figure 5.12.: LLBMC Performance of Counterexample Generation with Condorcet Criterion for Instant-Runoff Voting per Candidate

5.7.2. Single Transferable Vote

Having gained confidence in the generation of counterexamples for instant-runoff voting in Section 5.7.1, we examine the same for single transferable vote, which is more complex, but largely based on instant-runoff voting. Now we want to disprove for our implementation of single transferable vote the same property as for IRV (that the Condorcet winner is always elected if a Condorcet winner exists). We take exactly the same precondition as for instant-runoff voting in Section 5.7.1, since the possible ballot boxes are exactly identical.
Hence, the Condorcet criterion for STV is also almost identical to the Condorcet criterion for instant-runoff voting (Definition 14 on page 72) and only differs in the postcondition. As STV elects a set of candidates, only one of them can be the Condorcet winner. We hence formulate the Condorcet criterion for single transferable vote as follows:

Definition 15 (Condorcet Criterion for Multi-Seat Preferential Voting) *For a set of candidates C, a set of voters V, a set of preferences P and a set of ballot boxes B, we say a* multi-winner preferential voting scheme *s satisfies the Condorcet winner criterion or Condorcet criterion iff*

$$\begin{aligned}
&\forall b \in B, c \in C : \\
&\quad (\forall k \in C : k \neq c \Rightarrow \#\{v \in V \mid \exists p \in P : b_{v,p} = k \\
&\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p'\ \wedge\ b_{v,p'} = c)\ \vee\ \forall p' \in P : b_{v,p'} \neq c)\} \\
&\qquad\quad < \#\{v \in V \mid \exists p \in P : b_{v,p} = c \\
&\qquad\qquad\qquad \wedge\ ((\exists p' \in P : p < p'\ \wedge\ b_{v,p'} = k)\ \vee\ \forall p' \in P : b_{v,p'} \neq k)\}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow (s(b) = \emptyset \vee c \in s(b))
\end{aligned}$$

Then, we translate this formula to C code, include our implementation of single transferable vote as in Section 5.5 and run LLBMC on our program. The smallest counterexample can be generated in 4.1 seconds for three candidates and three voters with one vacancy. Within our timeout we find the biggest counterexample for three candidates, seven voters and one vacancy within 35 seconds. As STV with only one vacancy effectively does not differ from instant-runoff voting, we present the smallest counterexample with at least two vacancies and demonstrate its validity. Therein, we have three candidates, three voters and two vacancies.

Counterexample for the Condorcet Criterion with STV Let us take three candidates A, B and C, and the following three votes:
Hence, we need to check that this is not a spurious counterexample. Firstly, we determine the Condorcet winner by comparing all head-to-head matches. Therein, candidate C wins against candidate A with two winning matches and one loss, candidate C wins against candidate B with two winning matches and one loss, and candidate A wins against candidate B with two winning matches and one loss. This means the Condorcet loser is candidate B and the Condorcet winner is candidate C, since this candidate is defeated by every other candidate in head-to-head matches. Secondly, we examine possible outcomes

Voter 1	Voter 2	Voter 3
A	B	C
C	C	A
B	A	B

Table 5.4.: Counterexample for Condorcet Criterion with Single Transferable Vote

of an election with instant-runoff voting considering this example as an input. As no candidate attains the necessary quota in the first preferences, we start by eliminating one candidate. However, all three candidates have the same number of first preferences and we consequently need to choose at random. Taking into account that we can also eliminate candidate C, this shows already that there is an outcome where the Condorcet winner, who is candidate C, is not elected, as also for second vacancy this candidate stays eliminated in STV. There are variations of STV, which resurrect eliminated candidates after a seat is allocated, but these are argued to not belong to the STV family [BGS13]. Thus, we have a violation with the Condorcet criterion for this implementation of single transferable vote.

Results For this implementation (the complete specification can be found in Listing A.17 and Listing A.19 in Appendix A), LLBMC is able to disprove that single transferable vote satisfies the Condorcet criterion for up to three candidates and seven voters or four candidates and three voters. Only for three candidates and three voters, we get a counterexample for more than one vacancy. The run-times depending on the number of voters are shown in Fig. 5.13 for three candidates and one seat. The run-times for all disproofs are shown in Table 5.5, where besides the numbers of candidates (C), seats (S) and voters (V) as well as the actual run-time in seconds, we also depict the size of the generated formula (Formula) in terms of the number of expressions.

C	S	V	Formula Size [# expressions]	Run-time [s]
3	1	3	49082	4.040
3	1	5	75326	14.516
3	1	6	88798	29.670
3	1	7	101567	34.269
3	2	3	120223	6.874
4	1	3	99255	8.406

Table 5.5.: Generation of Counterexamples for the Condorcet Criterion with STV

In fact all run-times in Table 5.5 are very fast. However, LLBMC quickly exceeds the memory resources (leading to segmentation faults) and compared to the results of the experiments in Section 5.7.1, the number of successful disproofs is very small. This is also the case for only one vacancy, where the actual procedure is the same for both voting schemes, but detecting this simplification seems to be difficult for LLBMC. The generation of a counterexample for more than one vacancy succeeds only in one case.

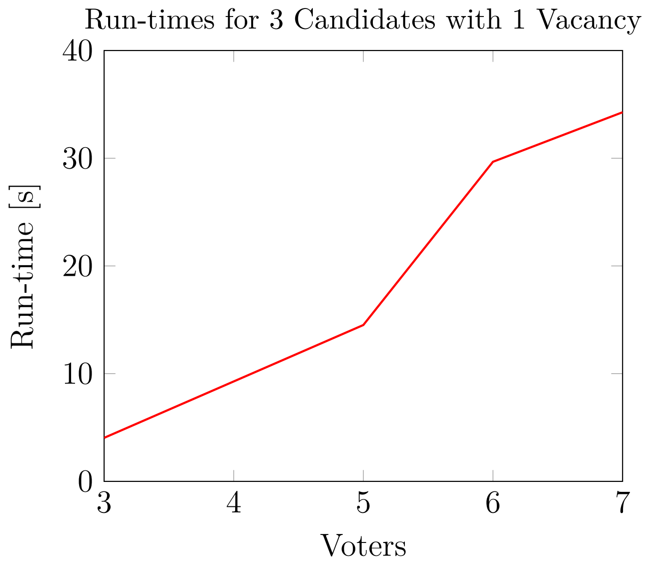

Figure 5.13.: LLBMC Verification Performance with Condorcet Criterion for STV per Voter

Yet when examining the results for three candidates and one vacancy depending on the number of voters in Fig. 5.13, the measurements only indicate an at most quadratic run-time complexity. But for a meaningful complexity analysis, the data pool of this experiment is too small and we would need to repeat our experiment on a faster machine with more memory in order to obtain meaningful results.

6. Improving Performance of Bounded Verification by Symmetry Reduction

In this chapter, we examine the potential of narrowing or reducing the search space, i.e. the set of all possible solutions, within the given boundaries. Therein we verify the voting scheme to be correct with respect to the given correctness criteria. The idea for this examination is based on and inspired by the so-called "symmetry-breaking predicates", which are a means to prevent redundant search space exploration [Cra+96]. The essential idea is to only check a subset of states or a representative from each class of symmetric states. Thereby, we can alleviate the effect of the state space explosion, which is inherent to model checking. Applications of this technique, as e.g. in the Alloy Analyzer, often lead to reductions in performance of several orders of magnitude [Jac06]; [Tur+07]. Generating these symmetry predicates or constraints automatically is justified to be challenging or even intractable in the general case [TBL10]. As such, this section explores the feasibility and potential of symmetry-breaking predicates for the field of voting schemes with respect to the verification of their correctness. We base our investigations on the well-known symmetry properties *anonymity* and *neutrality* of voting schemes and their formalisation in first order logic.

6.1. Plurality Voting with Anonymity

As we have seen in the first proofs of the monotonicity criterion for plurality voting in Section 5.2, the run-times grow especially quickly for increasing numbers of voters. We have found out that for C candidates and V voters the number of possible ballot boxes is C^V, meaning that the number of voters has a large effect on the run-time, much more than the number of candidates. However, this is also based on our program implementation of a ballot box, which is a simple array, having the voter number as index and the candidate number a value. This means e.g. that for the candidates 1 and 2, the ballot boxes $b_1 = \{1, 1, 2\}$, $b_2 = \{1, 2, 1\}$ and $b_3 = \{2, 1, 1\}$ are three separately considered ballot boxes, although we expect identical election results for all three ballot boxes in plurality voting.
In Section 2.1, we defined voting schemes to be *anonymous*, i.e. renaming or permuting any voters does not affect the election result. This is also something one would expect for any real-world election as usually the votes in a ballot box are not ordered and permutations in the order can easily happen also when the votes are being tallied. As for renaming, a voter would also expect the same election result in the event of a voter changing his or her name without changing his or her actual vote. However, this

abstraction is not reflected in the ballot boxes considered for our verification as many ballot box instances are now redundant with respect to *anonymity*. A solution to this is the reduction of the ballot box instances to only one representative for each group of ballot boxes which are equal modulo anonymity. E.g., for the exemplary set of ballot boxes b_1, b_2 and b_3 as described above, we only need to examine one of them, e.g. the ballot box b_1. This ballot box contains the same votes as the other two, but therein the votes are ordered by their names (in our case, these are natural numbers). When we apply this observation to the general case for plurality voting, it suffices to only examine ballot boxes, where the next vote is always equal to or greater than the vote before. Thereby, we achieve a *reduced* search space of ballot box instances (compared to the set of all ballot box instances). In first order logic, this can be formalised with the following formula:

Definition 16 (Symmetry Reduction for Anonymous Plurality Voting) *For an ordered set of voters V, a plurality voting ballot box b belongs to the* anonymously symmetry-reduced *search space iff*

$$\forall i : 0 < i < |V| : b_{(V_{i-1})} \leq b_{(V_i)}$$

Coverage of Entire Relevant Search Space Yet, the intuition or expectancy that our implementation of first-past-the-post plurality voting satisfies anonymity still needs a formal justification as the reduction of ballot box instances may cause us to miss out on relevant instances and not cover the whole relevant search space. By relevant, we mean that the instance behaves differently than its representative with respect to (a) the elected candidate and (b) the elected candidate when one vote is being raised as in the precondition for the monotonicity criterion. We start by proving that an election with an arbitrary ballot box has the same outcome as in the election with an ordered representative ballot box (case a):

Proof Let us consider an arbitrary ballot box b containing n votes and b's sorted representative r also containing n votes, which are the same as the votes in b, but in a different order. Then there exists a bijective permutation $p : \mathbb{N} \to \mathbb{N}$, which returns the position in the representative ballot box r for every position in ballot box b such that

$$\forall i : 0 < i \leq n \Rightarrow r_{p(i)} = b_i.$$

As the elements in b and r are exactly the same, we also have

$$\forall i : 0 < i \leq n \Rightarrow \#\{r_{p(i)} \in r\} = \#\{b_i \in b\}.$$

Let us assume without loss of generality that if the election with ballot box b elects a candidate, the elected candidate is b_e with $0 < e \leq n$. In our implementation of plurality voting, we elect a candidate if this candidate receives strictly more votes than any other candidate, i.e.:

$$\forall i : 0 < i \leq n \wedge b_i \neq b_e \Rightarrow \#\{b_i \in b\} < \#\{b_e \in b\}.$$

Hence, we can deduce the following:

$$b_e = r_{p(e)} \wedge \forall i : 0 < i \leq n \wedge r_{p(i)} \neq r_{p(e)} \Rightarrow \#\{r_{p(i)} \in b\} < \#\{r_{p(e)} \in b\}.$$

Consequently, if a candidate gets elected by plurality voting with ballot box b, the same candidate also gets elected with ballot box r. In case there is a tie in the election with b, this means the following:

$$\forall i : 0 < i \leq n \wedge 1 < |n| \Rightarrow \exists i' : b_i \neq b_{i'} \wedge \#\{b_i \in b\} \leq \#\{b_{i'} \in b\}.$$

As such, there is no candidate with strictly more votes than any other candidate. But then we can also deduce the following:

$$\forall i : 0 < i \leq n \wedge 1 < |n| \Rightarrow \exists i' : r_{p(i)} \neq r_{p(i')} \wedge \#\{r_{p(i)} \in b\} \leq \#\{r_{p(i')} \in b\}.$$

Thus, there is also no winner in the election with ballot box r. Taking these two results, we can conclude that the outcomes are indeed the same. ■

We proved that plurality voting elects the same candidate for a ballot box b as for its representative r. In the following, we prove that this also holds for the relationship between b' and r', where b' is identical to b in all votes except for one and the same holds for the relationship of r' and r (case b). The changed vote is the raised vote from the monotonicity criterion (Definition 9 on page 37), where the voter identity as well as the candidate, which is being voted for in the raised vote, are fixed. Our formal justification is as follows:

Proof For an arbitrary valid voter x and an arbitrary valid candidate y, we define the ballot box b' to match with b, except for the element at position x, for which we define ${b'}_x = y$. This means the two ballot boxes b and b' differ only in the one vote cast by voter x. We denote the vote by voter x in ballot box b as $b_x = y'$. Hence, we have the following formal relationship:

$$b_x = y' \wedge {b'}_x = y \wedge \forall i : 0 < i \leq n \wedge i \neq x \Rightarrow b_i = {b'}_i.$$

And hence we can make a statement about the number of occurrences:

$$\begin{aligned}\#\{y' \in b\} = \#\{y' \in b'\} + 1 \wedge \#\{y \in b'\} = \#\{y \in b\} + 1 \\ \wedge \forall i : 0 < i \leq n \wedge i \neq x \Rightarrow \#\{b_i \in b \setminus \{y, y'\}\} = \#\{{b'}_i \in b' \setminus \{y, y'\}\}.\end{aligned}$$

We then specify the representative r by ordering the votes in b as in the previous proof, where we know the following:

$$\forall i : 0 < i \leq n \Rightarrow r_{p(i)} = b_i.$$

And also the representative r' by ordering the votes in b' likewise, hence:

$$\forall i : 0 < i \leq n \Rightarrow {r'}_{p(i)} = {b'}_i.$$

We then need to establish a relationship between r' and r. For this, we take the equality of occurrences between b and r:

$$\forall i : 0 < i \leq n \Rightarrow \#\{r_{p(i)} \in r\} = \#\{b_i \in b\}.$$

And the same relationship between b' and r':

$$\forall i : 0 < i \leq n \Rightarrow \#\{{r'}_{p(i)} \in r'\} = \#\{{b'}_i \in b'\}.$$

Thus, we can deduce the following relationship between r and r' as between b and b':

$$\begin{aligned}&\#\{y' \in b\} = \#\{y' \in b'\} + 1 \wedge \#\{y \in b'\} = \#\{y \in b\} + 1\\ &\qquad \wedge \forall i : 0 < i \leq n \wedge i \neq x \Rightarrow \#\{b_i \in b \setminus \{y, y'\}\} = \#\{{b'}_i \in b' \setminus \{y, y'\}\}.\end{aligned}$$

Hence, we can conclude the outcome of our implementation of plurality voting with ballot box r' is the same as with b' and we have proven that our symmetry reduction condition does not eliminate any relevant instances for the analysis of monotonicity with first-past-the-post plurality voting. ■

Results By applying the symmetry reduction in Definition 16 to the input ballot box, we reduce the original C^V instances to $\frac{(C+V-1)!}{V! * (C-1)!}$ ballot box instances, which is considerably less than for plurality voting without this symmetry reducing assumption. LLBMC is able to show that our implementation (the complete specification can be found in Listing A.6 in Appendix A) of plurality voting with these symmetry assumptions satisfies the monotonicity criterion for up to 5 candidates and 40 voters. The run-times of the tool depending on the number of voters and candidates are shown in Fig. 6.1 (separate curves for 3, 6, 9, 12, and 15 candidates) and Fig. 6.2 (separate curves for 2, 4, 6, 8, and 10 voters) respectively. The lower number of ballot box instances is also clearly visible in the run-times of our experiment, when we compare our results to Fig. 5.1 and Fig. 5.2.

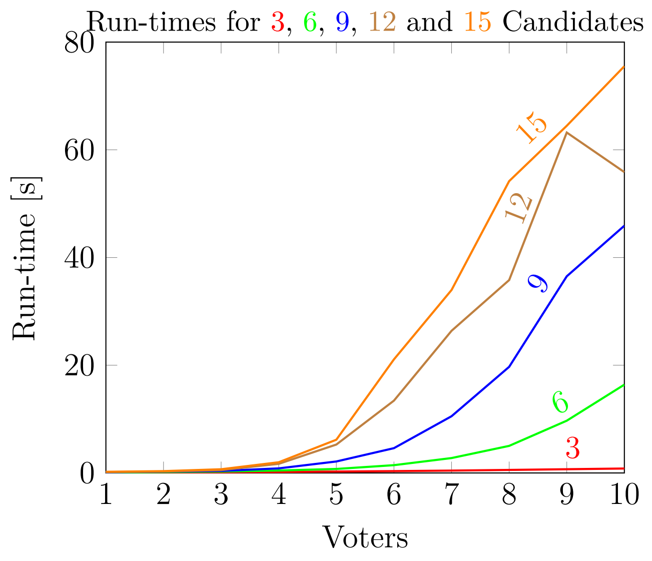

Figure 6.1.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting with Symmetry Breaking (Anonymity) per Voter

Similar to the previous experiments for plurality voting, we notice again some unusual deviations with respect to the number of candidates for the graph in Fig. 6.2. As the run-times of the results using the symmetry-breaking predicates outperform the ones from the previous chapter to a big extent, we learn that the number of ballot box instances has a great effect in the verification of the monotonicity criterion for plurality voting and might also be adaptable to other correctness criteria and voting schemes. However, the number of ballot box instances can still be reduced even further without losing any significance of our verification results, as demonstrated in the following section.

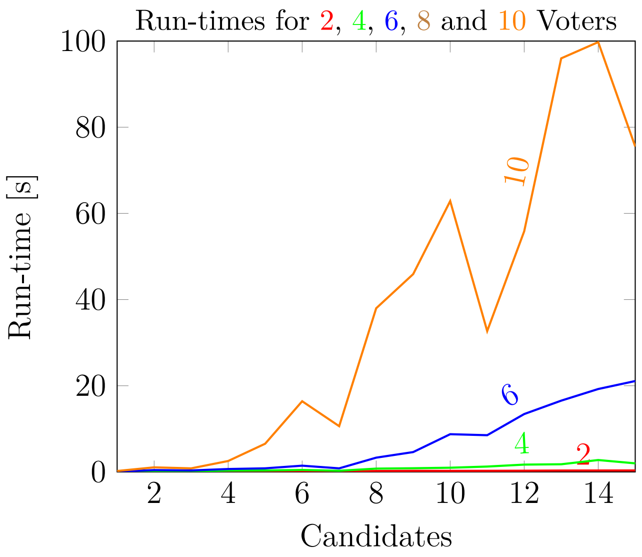

Figure 6.2.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting with Symmetry Breaking (Anonymity) per Candidate

6.2. Plurality Voting with Anonymity and Neutrality

In the definition of voting schemes in Section 2.1 we also defined voting schemes to be *neutral*, i.e. renaming or permuting candidates does not affect the election result. E.g. for three candidates, we could have the ballot boxes $b_1 = \{1, 2, 3\}$, $b_2 = \{1, 2, 5\}$, $b_2 = \{1, 3, 5\}$, $b_3 = \{1, 9, 13\}$ and $b_4 = \{7, 17, 42\}$. All these ballot boxes b_1 through b_4 satisfy *anonymity*, but we expect identical election results for all four ballot boxes in plurality voting modulo renaming some candidates. In our implementation, all four ballot boxes are different instances, which are analysed separately. However, the actual representation of candidate does not matter if there is a one-to-one mapping, where no two distinguishable candidates get the same names. Hence, we can also apply such a renaming to the previously mentioned ballot boxes, such that the ballot boxes above have one representative $b = \{1, 2, 3\}$, wherein the first vote is always for the first candidate and for a consecutive vote, the candidates being voted for have a “distance” of one or zero, when having a one-to-one mapping from the candidate names to the set of natural numbers.

In first order logic, this can be formalised with the following formula:

Definition 17 (Symmetry Reduction for Neutral Plurality Voting) *For an ordered set of voters V, an ordered set of candidates C and a candidate index function $I : C \to \mathbb{N}_0$ with $I : C_i \mapsto i$, we say a plurality voting ballot box b belongs to the* neutrally symmetry-reduced *search space iff*

$$b_{V_0} = C_0 \wedge \forall i : 0 < i < |V| \Rightarrow 0 \le |I(b_{(V_i)}) - I(b_{(V_{i-1})})| \le 1$$

The basic intuition for the correctness is essentially the same as for anonymity, only considering a different dimension.

Results We can apply the symmetry-breaking predicates from Definition 17 considering *neutrality* additionally to the symmetry-breaking predicates from Definition 16 considering *anonymity* to the ballot box instances, and thereby reduce the previous $\frac{(C+V-1)!}{V!*(C-1)!}$ instances even further. In our experiments, LLBMC is able to show that this implementation (the complete specification can be found in Listing A.7 in Appendix A) of plurality voting with these even stricter symmetry assumptions satisfies the monotonicity criterion for up to 5 candidates and 45 voters or 30 candidates and 21 voters.

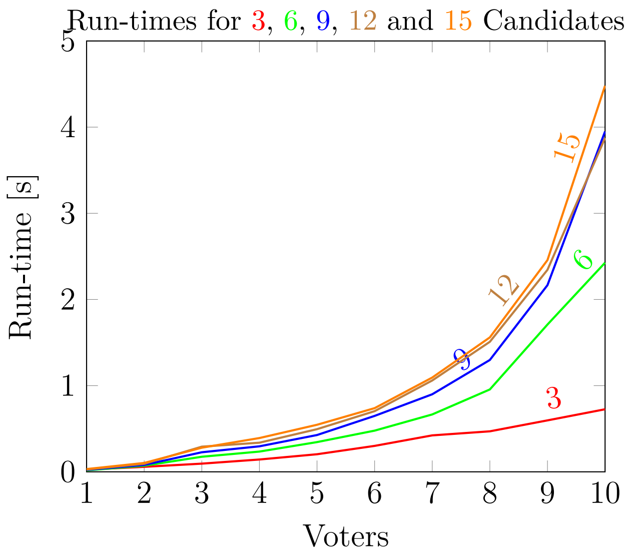

Figure 6.3.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting with Strict Symmetry Breaking per Voter

The run-times depending on the number of voters and candidates are shown in Fig. 6.3 (separate curves for 3, 6, 9, 12, and 15 candidates) and Fig. 6.2 (separate curves for 2, 4, 6, 8, and 10 voters) respectively. Now we run into memory problems before we reach the timeout for most of our executions, from which we can deduce that the generated formula is much more concise and holds much less redundancies than without the symmetry reduction. However, we are still not rid of the unusual deviations in Fig. 6.4 depending on the number of candidates.

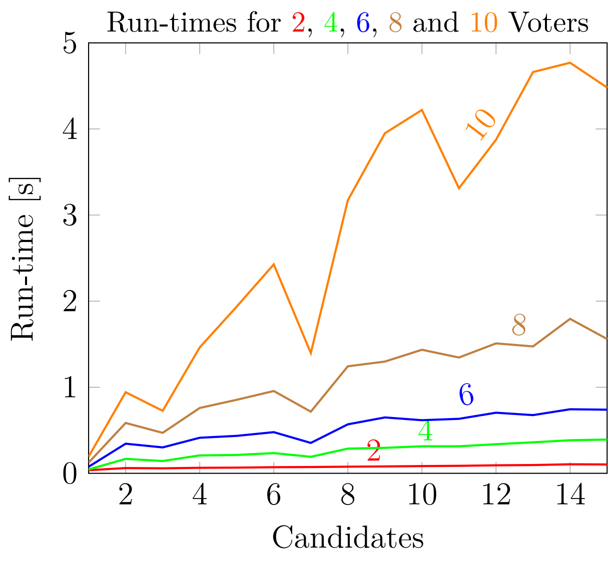

Figure 6.4.: LLBMC Verification Performance with Monotonicity Criterion for Plurality Voting with Strict Symmetry Breaking per Candidate

6.3. Approval Voting with Anonymity and Neutrality

Also for approval voting, our implementation for the ballot box comprises a high percentage of symmetric instances. We found out in Section 5.3 that the number of ballot box instances is 2^{C*V} and hence suspect a big potential to reduce this number similarly to the previous experiments with plurality voting. This time, both the number of candidates as well as the number of voters are part of the exponent and hence have a large effect on the complexity of the bounded verification. However, this is also based on our program implementation of a ballot box, which is a two-dimensional array, having the voter number as first index, the candidate number as a second one and either 1 or 0 as value. Very similarly to the previous section on symmetry reduction for plurality voting, we assume *anonymity* and *neutrality* as defined in Section 2.1. From these assumptions we can assume an ordered relation along both array dimensions, the one along the first dimension reflecting *anonymity* and the one along the second dimension reflecting *neutrality*. Hence, it suffices to only examine ballot boxes, where the approval or disapproval for a certain candidate is always equal to or greater than in the vote before and where for any voter the approval or disapproval for a candidate is always equal to or greater than for the candidate before. In first order logic, this can be formalised with the following formula:

Definition 18 (Symmetry Reduction for Approval Voting) *For an ordered set of voters V and an ordered set of candidates C, we say an approval voting ballot box b belongs to the* symmetry-reduced *search space iff*

$$\begin{aligned}\forall i, j :& ((0 < i < |V| \land 0 \leq j < |C|) \Rightarrow b_{(V_{i-1},C_j)} \leq b_{(V_i,C_j)}) \\ \land& ((0 \leq i < |V| \land 0 < j < |C|) \Rightarrow b_{(V_i,C_{j-1})} \leq b_{(V_i,C_j)})\end{aligned}$$

The basic intuition for the correctness is essentially the same as for plurality, only considering a slightly different data structure for the ballot box.

Results By applying the symmetry-breaking predicates from Definition 18 to our verification of the monotonicity criterion for approval voting, we can reduce the run-times even further (compared to the results for approval voting from the last chapter) and LLBMC is able to show that our implementation (the complete specification can be found in Listing A.10 in Appendix A) of approval voting with the symmetry assumptions satisfies the monotonicity criterion for up to 5 candidates and 161 voters or 20 candidates and 39 voters. The run-times depending on the number of voters and candidates are shown in Fig. 6.5 (separate curves for 2, 4, 8, 12, 16 and 20 candidates) and Fig. 6.6 (separate curves for 8, 16, 24, 32, and 40 voters) respectively. When we compare these results to the outcome of the experiments for approval voting without any symmetry-breaking constraints in Fig. 5.3 and Fig. 5.4, we observe a drastic reduction of the development of run-times depending on the number of candidates, but also considering the development based on the number of voters. Although we cannot give an explicit formula for the reduction in run-time complexity or reduction of ballot box instances, these findings are very promising and indicate a potential in adapting this approach to similar voting schemes or further correctness criteria for approval voting.

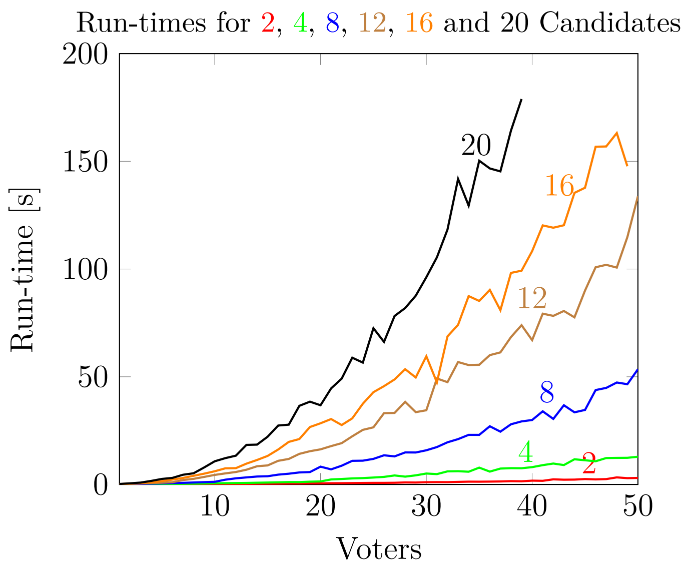

Figure 6.5.: LLBMC Verification Performance with Monotonicity Criterion for Approval Voting with Symmetry Breaking per Voter

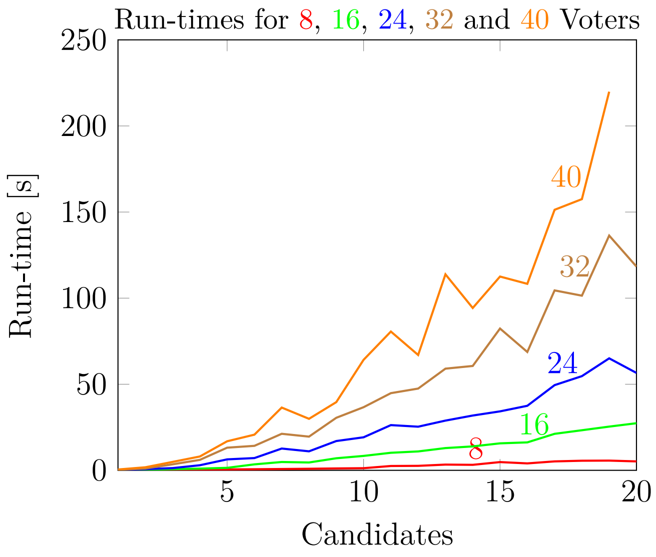

Figure 6.6.: LLBMC Verification Performance with Monotonicity Criterion for Approval Voting with Symmetry Breaking per Candidate

7. Conclusion

Voting schemes, as a fundamental part of democratic elections, are essential for the establishment of trust, credibility and thus legitimacy in a democratic system as a whole. However, designing a voting scheme without flaws, which still gives significant democratic guarantees, is a difficult task as a trade-off between desirable properties is non-trivial and error-prone. For this matter, various differing and growingly complex voting schemes are used throughout the world and some of them have revealed serious flaws and undesired side-effects. We tackle this issue by proposing an incremental and iterative process in order to develop correct voting schemes with appropriate and understandable correctness criteria based on automated formal methods of reasoning. In particular, this thesis analyses two different forms of verification with respect to their role in this development process targeting formal correctness of voting schemes. Thereby, we perform a comprehensive set of case studies by using “medium-weight” and “light-weight” verification techniques.

7.1. Related Work

Other research related to our work exists in various areas. The first and perhaps most obvious relation concerns other approaches in the analysis of voting schemes. Besides the examination of abstract voting schemes as done within this work and closely tied to the field of *social choice theory*, some approaches consider concrete implementations of voting schemes, sometimes on a more technical level, i.e. considering concrete real-world implementations. A mostly different level relates to the formal verification techniques, which we use and examine in our analysis.

Analysis of Voting Schemes Regarding the analysis of voting schemes, there does not appear to be much research using automated reasoning methods by the means of formal logics. There are some case studies on specifying very specific voting schemes given by national jurisdictions and they commonly reveal serious flaws [Coc12][YS97]. Besides well-known formal criteria for voting systems, also more informal criteria or vulnerabilities such as strategic manipulation [Bas14] or fairness and efficiency [BCE13] are examined. As well as analysing existent voting schemes, there is research on designing new voting schemes based on existent ones [AK11]. Some more fundamental research analyses general approaches on how to deal with existing notions of correctness with respect to voting schemes as there is already a multitude of different criteria, which are often mutually exclusive with each other [Nur12]. Moreover, the specification and

verification of existing complex voting schemes according to their jurisdictions is often a difficult and labour-intensive task [Meu14].
There exists extensive research in the field of social choice theory analysing theoretical voting schemes on a more intuitive or experimental level involving empirical experiments or comparisons of previous elections [Pac12]; [Gal13]; [BF88]; [RG98]. Other research in the field of mathematics and computer science covers algorithmic analysis and classification of voting schemes [EFS10]. This analysis defines mathematical distances using rather general voting scheme properties such as neutrality and consistency, which are usually pre-assumed in our analysis. Furthermore, there is research on the verification of concrete voting systems or implementations, i.e. considering a concrete voting software [DYJ08]. However, it is argued and justified by the employing bounded model checking using the SMT solver Z3 that formal methods can and should be used to ensure an intended behaviour of voting schemes [Bec+14b].

Verification Techniques Within this work, we also examined potential improvements by adapting the used verification techniques. Some research on the trade-offs between bounded and modular verification focuses on how to make automatic verification more efficient and powerful. This involves adding and effectively using some external decision procedures or SMT solvers e.g. for quantifiers or integer arithmetics [AMP09] or even using a whole different logical foundation. There is also research on the combination of bounded and modular verification techniques [HN09], often focused on different logical theories [GL94]. Evaluations on this field are promising, but commonly envisage general data structures [DCJ06]. A general integration of deductive verification into bounded model checking in order to exploit modularisation to achieve bounded guarantees, as opposed to doing it the other way around as we examined, experienced limited but promising examination [Bec+12].
Another aspect of our analysis is the improved efficiency of bounded verification tasks by applying effective symmetry-breaking predicates in order to reduce the search space of possible input instances. Related work exists that focuses on breaking symmetries on the problem specification level [MC05][CM05] and also on methods for automatically generating symmetry-breaking predicates on classes of combinatorial objects for search problems [Shl07]. More applied work on this task provides tools, which detect symmetries in structured graphs generated from CNF formulas [Dar+04].
Finally, related work on relational verification, as we did in analysing the relation between two runs of a voting scheme with a given relation on their input for checking the monotonicity criterion, using product programs [BCK11] seems promising for small implementations of voting schemes.

7.2. Summary and Results

In this thesis, we analysed the general applicability of two verification techniques *software bounded model checking* (SBMC) and *auto-active* deductive annotation-based verification, which we classify as “light-weight” and “medium-weight” verification techniques

respectively, to the formal verification of voting schemes. Thereto, we examined general implementations of the widely used voting schemes: plurality voting, approval voting, instant-runoff voting (IRV) and single transferable vote (STV) with respect to the three well-known voting scheme criteria monotonicity, Condorcet loser and Condorcet winner. The verification was realised with the Verifying C Compiler (VCC) and the Low-Level Bounded Model Checker (LLBMC). For the process of this analysis, we formalised the voting scheme criteria in first-order predicate logic and translated them to the specification languages of VCC and LLBMC and provided auxiliary specifications where necessary. Whereas the translation for VCC did not include significant transformations as its specification language directly supports first-order predicate logic, we established a translation method for LLBMC, which supports a specification language with the set of legal C/C++ statements of type boolean.
Furthermore, the specification for VCC included the formalisation of auxiliary specifications to enable its modular verification methodology, in particular for a verification of the involved loop-statements. This process turned out surprisingly difficult with a significant amount of user interaction regarding the finding of useful auxiliary specifications. In particular, the complex combination of quantified formulas demanded an efficient and effective mechanism to provide sensible quantifier instantiations, which was not feasible with our case study to provide a full and general proof within our limits of memory capacity. Yet, we were able to push the number of voters by using an intermediate lemma in order to benefit from VCC's possibilities for modularisation and abstraction. As however we did achieve a successful verification for a bounded number of voters, we analysed the applicability of bounded verification methods such as loop unrolling to VCC's verification process. This approach did only lead to increased run-times, but no meaningful advances considering the verification achievements. For the applicability of VCC's verification methodology for the simple voting scheme implementation, we can conclude this poses a major challenge as the specification elements and especially their combination turned out to be too complex for an effective and efficient deduction.
The verification with LLBMC necessitated a translation of our problem specification to be handled by LLBMC, but the verification was fully automatic without any user interaction. We achieved successful proofs for small numbers of voters and candidates, however with an exponential run-time in the number of voters for all voting schemes and properties. The more complex preferential voting schemes IRV and STV turned out to produce hardly any meaningful results. Furthermore, we analysed the capability of LLBMC to produce counterexamples for non-complying criteria, which is a more general purpose of SBMC. This is a desirable verification scenario for real voting schemes as violated properties are often hidden in the intricate structure of voting schemes and discovered only when it is already too late. Our analysis showed a better applicability to the complex voting schemes IRV and STV than for proving correctness criteria. Still, the numbers were small, but we observed a smaller increase in run-times for increasing numbers of voters and candidates. Moreover, we established improved results in proving correctness criteria for voting schemes with non-preferential ballots as plurality voting and approval voting by exploiting the inherent symmetric structures *anonymity* and *neutrality* of the set of possible ballot boxes. We formalised these two properties for

plurality voting and approval voting in first-order predicate logic as *symmetry-breaking predicates* and were able to achieve successful verification results for meaningful numbers of candidates and voters with respect to the *small scope hypothesis*.

7.3. Outlook and Future Work

The verification of voting schemes is a challenging task. We observed a promising applicability of bounded verification techniques and that using *auto-active* annotation-based deductive verification turns out challenging. Apart from investigations for a better performance of SMT solvers for complex combinations of quantified formulas, an enhanced support of *auto-active* verification tools for user interaction with respect to quantifier instantiation has potential to achieve progress towards our objective. Furthermore, the technique of relational verification as introduced in the section of related work (Section 7.1) seems promising for simple voting schemes. In this thesis, we analysed the use of the bounded verification technique loop unrolling within a modular deductive verification tool, but the application of modular deductive verification techniques within a bounded model checking tool remains to be analysed for verifying voting schemes. We also analysed the potential of *symmetry-breaking predicates* within bounded verification and achieved promising results. Hence, we see potential in expanding these investigations on several levels. This is firstly the expansion on voting schemes with a more complex ballot structure as an input, and secondly the advance of even more effective predicates to break further existing symmetries and establish a framework for sensible symmetry properties.
Moreover, we want to apply our approach to more concrete (i.e. less abstract) voting schemes as used in real elections. Further future work concerns applying "heavy-weight" verification techniques as to observe the benefit of more powerful user interaction to the case studies within this thesis. More in the long-term, we envision the integration of our approach in a development process for new voting schemes with formally verified correctness properties.

A. Implementations and Specifications

A.1. Plurality Voting for VCC

In Listing A.1, the first argument is the upper bound of voters and the second one is the filename.

```
@call vcc /p:"/DBOUND=%%1" /z3:"memory:1500" %%2
```

Listing A.1: Script with Parameters for VCC Call

A.1.1. Full Specification of Monotonicity for Plurality Voting

```
#include <vcc.h>
#include <limits.h>

extern int C, V;

#ifndef BOUND
#define BOUND 3
#endif
#define valid_cand(a) ((0 < a) && (a <= C))
#define valid_candZero(a) ((0 <= a) && (a <= C))
#define valid_voter(a) ((0 <= a) && (a < V))
#define valid_voteCount(a) ((0 <= a) && (a <= V))

_(ghost _(pure) \integer count(int votes[int],
                               int upto,
                               int cand)
    _(decreases upto)
    _(returns (upto <= 0) ?
            0 : count(votes, upto-1, cand)
               + (votes[upto-1] == cand ? 1 : 0))
{
    return (upto <= 0) ?
            0 : count(votes, upto-1, cand)
               + (votes[upto-1] == cand ? 1 : 0);
})
```

```

void main(_(ghost int v1[int]) _(ghost int v2[int])
          _(out int elect1) _(out int elect2)
          int m, int n)
    _(requires 0 < V && V < BOUND)
    _(requires 0 < C && C < INT_MAX - 1)
    _(requires \forall int i; {v1[i]}
            (valid_voter(i) ==> valid_cand(v1[i]))
         && (valid_voter(i) ==> valid_cand(v2[i])))
    _(requires v1[n] != v2[n])
    _(requires valid_cand(m) && valid_voter(n))
    _(requires v2 == \lambda int i; i == n ? m : v1[i])
    _(ensures (m == elect1) ==> (elect1 == elect2))
{
    _(ghost int res1[int] = \lambda int i; 0;)
    _(ghost int res2[int] = \lambda int i; 0;)

    //================FIRST ELECTION================

    _(ghost int tmp = 0;)
    _(ghost for (int i = 0; i < V; i++)
        _(invariant valid_voteCount(i)
            && valid_candZero(tmp))
        _(invariant \forall int k; (valid_candZero(k)
            ==> 0 <= res1[k] && res1[k] <= i))
        _(invariant \forall int k;
            {res1[k], count(v1, i, k)}
            (valid_candZero(k) ==>
                (res1[k] == count(v1, i, k))))
    {
        tmp = v1[i];
        res1[tmp] = res1[tmp] + 1;
    })

    _(ghost int max1 = -1;)
    _(ghost elect1 = 0;)

    _(ghost for (int i = 1; i <= C; i++)
        _(invariant valid_candZero(i-1) && elect1 <= C)
        _(invariant elect1 != 0 ==> res1[elect1] == max1)
        _(invariant elect1 != 0
            <==> (\exists int k; {res1[k]}
                {\match_long(k)} {:hint \match_long(k)}
                    0 < k && k < i
```

```
                  && (\forall int j;
                         0 < j && j < i && j != k
                      ==> res1[j] < res1[k])))
        _(invariant 0 <= max1 ==>
            \exists int k; 0 < k && k < i
                          && res1[k] == max1)
        _(invariant \forall int k; 0 < k && k < i
                ==> res1[k] <= max1)
    {
        if (max1 < res1[i]) {
            max1 = res1[i];
            elect1 = i;
        } else if (max1 == res1[i]) elect1 = 0;
    })

    //================SECOND ELECTION===============

    _(ghost int tmp = 0;)
    _(ghost for (int i = 0; i < V; i++)
        _(invariant valid_voteCount(i)
            && valid_candZero(tmp))
        _(invariant \forall int k; (valid_candZero(k)
            ==> 0 <= res2[k] && res2[k] <= i))
        _(invariant \forall int k;
            {count(v2, i, k), res2[k]}
            (valid_candZero(k) ==>
                (res2[k] == count(v2, i, k))))
    {
        tmp = v2[i];
        res2[tmp] = res2[tmp] + 1;
    })

    _(ghost int max2 = -1;)
    _(ghost elect2 = 0;)

    _(ghost for (int i = 1; i <= C; i++)
        _(invariant valid_candZero(i-1)
            && 0 <= elect2 && elect2 < i)
        _(invariant elect2 != 0 ==> res2[elect2] == max2)
        _(invariant elect2 == 0 && 1 < i ==>
            \exists int j, k; 0 < j && j < i && 0 < k
                && k < i && k != j && res2[k] == res2[j])
        _(invariant 0 <= max2
            ==> \exists int k; 0 < k && k < i
```

```
                             && res2[k] == max2)
        _(invariant \forall int k; 0 < k && k < i
            ==> res2[k] <= max2)
        _(invariant (\forall int j; {res2[j]}
            0 < j && j < i && elect2 != j
            ==> (\exists int k; 0 < k && k < i
                    && k != j && res2[j] <= res2[k])))
    {
        if (max2 < res2[i]) {
            max2 = res2[i];
            elect2 = i;
        } else if (max2 == res2[i]) {
            elect2 = 0;
        }
    })
}
```

Listing A.2: VCC Specification of Monotonicity for Plurality Voting

A.1.2. Lemma for Monotonicity Criterion

```
#include <vcc.h>
#include <limits.h>

extern int C;
extern int V;

#define valid_cand(a) ((0 < a) && (a <= C))
#define valid_candZero(a) ((0 <= a) && (a <= C))
#define valid_voter(a) ((0 <= a) && (a < V))

_(ghost _(pure) \integer count(int votes[int],
                               int upto,
                               int cand)
    _(decreases upto)
    _(returns (upto <= 0) ?
            0 : count(votes, upto-1, cand)
               + (votes[upto-1] == cand ? 1 : 0))
{
    return (upto <= 0) ?
            0 : count(votes, upto-1, cand)
               + (votes[upto-1] == cand ? 1 : 0);
})
```

```

void main(_(ghost int v1[int]) _(ghost int v2[int])
          _(ghost int elect1) _(ghost int elect2)
          _(ghost int res1[int]) _(ghost int res2[int])
          int m, int n)
    _(requires 0 < V && V < INT_MAX - 1)
    _(requires 0 < C && C < INT_MAX - 1)
    _(requires v1 != v2)
    _(requires v1[n] != v2[n])
    _(requires valid_cand(m) && valid_voter(n))
    _(requires v2 == \lambda int i; i == n ? m : v1[i])
    _(requires \forall int k; (valid_candZero(k)
        ==> (res1[k] == count(v1, V, k))))
    _(requires \forall int k; (valid_candZero(k)
        ==> (res2[k] == count(v2, V, k))))
    _(requires elect1 != 0
        ==> (\forall int k; 0 < k && k <= C && k != elect1
                ==> res1[k] < res1[elect1]))
    _(requires (\forall int j; {res2[j]}
        0 < j && j <= C && elect2 != j
        ==> (\exists int k; 0 < k && k <= C && k != j
                && res2[j] <= res2[k])))
    _(requires \forall int j; \match_long(res1[v1[j]]))
    _(requires \forall int j; valid_voter(j)
                    ==> \match_long(res2[v2[j]]))
    _(ensures (m == elect1) ==> (elect1 == elect2))
{
}
```

Listing A.3: VCC Lemma Specification for Monotonicity

A.2. Plurality Voting for LLBMC

In Listing A.4, the first argument is the number of candidates, the second one the number of voters, the third one the filename (without file extension), and the fourth one is for optional additional parameters.

```
#!/bin/bash
clang -D C=$1 -D V=$2 -c -g -emit-llvm $3.c -o $3.bc
llbmc -log-level=verbose -only-custom-assertions --smt-
   solver=stp-cms --stp-eager-read-axioms $4 $3.bc
```

Listing A.4: Script with Parameters for LLBMC Call of Plurality Voting

A.2.1. General Plurality Voting

```c
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 3
#endif

#ifndef C
#define C 3
#endif

unsigned int voting(unsigned int votes[V]) {
    unsigned int res[C + 1];
    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes[i]);
        assume (votes[i] <= C);
    }

    for (unsigned int i = 0; i <= C; i++) res[i] = 0;

    unsigned int tmp = 1;
    for (unsigned int i = 0; i < V; i++) {
        tmp = votes[i];
        res[tmp] = res[tmp] + 1;
    }

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

void monotonicity(unsigned int votes1[V],
                  unsigned int votes2[V],
                  unsigned int m,
                  unsigned int n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);

    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes1[i]);
        assume (0 < votes2[i]);
        assume (votes1[i] <= C);
        assume (votes2[i] <= C);
        if (i != n) assume (votes1[i] == votes2[i]);
    }
    assume (votes1[n] != m);
    assume (votes2[n] == m);

    unsigned int elect1 = voting(votes1);
    unsigned int elect2 = voting(votes2);
    if (m == elect1) assert (elect1 == elect2);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V], v2[V];
    for (unsigned int i = 0; i < V; i++) {
        v1[i] = __llbmc_nondef_unsigned_int();
        v2[i] = __llbmc_nondef_unsigned_int();
    }
    unsigned int vote  = __llbmc_nondef_unsigned_int();
    unsigned int voter = __llbmc_nondef_unsigned_int();

    monotonicity(v1, v2, vote, voter);
    return 0;
}
```

Listing A.5: LLBMC Specification for Plurality Voting

A.2.2. FPTP with Symmetry Breaking Predicates for Anonymity

```
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 3
#endif

#ifndef C
#define C 3
```

```
#endif

unsigned int voting(unsigned int votes[V]) {
    unsigned int res[C + 1];
    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes[i]);
        assume (votes[i] <= C);
        if (0 < i) assume (votes[i-1] <= votes[i]);
    }

    for (unsigned int i = 0; i <= C; i++) res[i] = 0;

    unsigned int tmp = 1;
    for (unsigned int i = 0; i < V; i++) {
        tmp = votes[i];
        res[tmp] = res[tmp] + 1;
    }

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

void monotonicity(unsigned int votes1[V],
                  unsigned int votes2[V],
                  unsigned int m,
                  unsigned int n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);

    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes1[i]);
        assume (0 < votes2[i]);
        assume (votes1[i] <= C);
        assume (votes2[i] <= C);
        if (0 < i) {
            assume (votes1[i-1] <= votes1[i]);
            assume (votes2[i-1] <= votes2[i]);
        }
        if (i != n) assume (votes1[i] == votes2[i]);
    }

    assume (votes1[n] != m);
    assume (votes2[n] == m);

    unsigned int elect1 = voting(votes1);
    unsigned int elect2 = voting(votes2);
    if (m == elect1) assert (elect1 == elect2);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V];
    unsigned int v2[V];
    for (unsigned int i = 0; i < V; i++) {
        v1[i] = __llbmc_nondef_unsigned_int();
        v2[i] = __llbmc_nondef_unsigned_int();
    }
    unsigned int vote  = __llbmc_nondef_unsigned_int();
    unsigned int voter = __llbmc_nondef_unsigned_int();

    monotonicity(v1, v2, vote, voter);
    return 0;
}
```

Listing A.6: LLBMC Specification for Anonymous Plurality Voting

A.2.3. FPTP with Predicates for Anonymity and Neutrality

```
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 3
#endif

#ifndef C
#define C 3
#endif

unsigned int voting(unsigned int votes[V]) {
    unsigned int res[C + 1];
    assume (votes[0] == 1);
    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes[i]);
        assume (votes[i] <= C);
        if (0 < i) {
            assume (0 <= (votes[i] - votes[i-1]));
            assume ((votes[i] - votes[i-1]) <= 1);
        }
    }

    for (unsigned int i = 0; i <= C; i++) res[i] = 0;

    unsigned int tmp = 1;
    for (unsigned int i = 0; i < V; i++) {
        tmp = votes[i];
        res[tmp] = res[tmp] + 1;
    }

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

void monotonicity(unsigned int votes1[V],
                  unsigned int votes2[V],
                  unsigned int m,
                  unsigned int n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);

    assume (votes1[0] == 1);
    for (unsigned int i = 0; i < V; i++) {
        assume (0 < votes1[i]);
        assume (0 < votes2[i]);
        assume (votes1[i] <= C);
        assume (votes2[i] <= C);
        if (0 < i) {
            assume (0 <= (votes1[i] - votes1[i-1]));
            assume ((votes1[i] - votes1[i-1]) <= 1);
        }
        if (i != n) assume (votes1[i] == votes2[i]);
    }

    assume (votes1[n] != m);
    assume (votes2[n] == m);

    unsigned int elect1 = voting(votes1);
    unsigned int elect2 = voting(votes2);
    if (m == elect1) assert (elect1 == elect2);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V];
    unsigned int v2[V];
    for (unsigned int i = 0; i < V; i++) {
        v1[i] = __llbmc_nondef_unsigned_int();
        v2[i] = __llbmc_nondef_unsigned_int();
    }
    unsigned int vote  = __llbmc_nondef_unsigned_int();
    unsigned int voter = __llbmc_nondef_unsigned_int();

    monotonicity(v1, v2, vote, voter);
    return 0;
}
```

Listing A.7: LLBMC Specification for Anonymous and Neutral Plurality Voting

A.3. Approval Voting for LLBMC

In Listing A.8, the first argument is the number of candidates, the second one the number of voters, the third one the filename (without file extension), and the fourth one is for optional additional parameters.

```
#!/bin/bash
clang -D C=$1 -D V=$2 -c -g -emit-llvm $3.c -o $3.bc
llbmc -log-level=verbose -only-custom-assertions $4 $3.bc
```

Listing A.8: Script with Parameters for LLBMC Call of Approval Voting

A.3.1. General Approval Voting

```c
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 3
#endif

#ifndef C
#define C 3
#endif

unsigned int voting(unsigned int votes[V][C]) {
    unsigned int res[C];
    for (unsigned int i = 0; i < V; i++) {
        for (unsigned int j = 0; j < C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= 1);
        }
    }

    for (unsigned int i = 0; i < C; i++) res[i] = 0;

    for (unsigned int i = 0; i < V; i++)
        for (unsigned int j = 0; j < C; j++)
            res[j] += votes[i][j];

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

void monotonicity(unsigned int votes1[V][C],
                  unsigned int votes2[V][C],
                  unsigned int m,
                  unsigned int n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);

    for (unsigned int i = 0; i < V; i++) {
        for (unsigned int j = 0; j < C; j++) {
            assume (0 <= votes1[i][j]);
            assume (0 <= votes2[i][j]);
            assume (votes1[i][j] <= 1);
            assume (votes2[i][j] <= 1);
            if ((i != n) || (j != (m-1)))
                assume (votes1[i][j] == votes2[i][j]);
        }
    }

    assume (votes1[n][m-1] == 0);
    assume (votes2[n][m-1] == 1);

    unsigned int elect1 = voting(votes1);
    unsigned int elect2 = voting(votes2);
    if (m == elect1) assert (elect1 == elect2);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V][C];
    unsigned int v2[V][C];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < C; j++) {
            v1[i][j] = __llbmc_nondef_unsigned_int();
            v2[i][j] = __llbmc_nondef_unsigned_int();
        }
    }
    unsigned int vote  = __llbmc_nondef_unsigned_int();
    unsigned int voter = __llbmc_nondef_unsigned_int();

    monotonicity(v1, v2, vote, voter);
    return 0;
}
```

Listing A.9: LLBMC Specification for Approval Voting

A.3.2. Approval Voting with Symmetry Breaking Predicates

```
#include <stdlib.h>
#include "llbmc.h"

```

```
#ifndef V
#define V 3
#endif

#ifndef C
#define C 3
#endif

unsigned int voting(unsigned int votes[V][C]) {
    unsigned int res[C];
    for (unsigned int i = 0; i < V; i++) {
        for (unsigned int j = 0; j < C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= 1);
            if (0 < i)
                assume (votes[i-1][j] <= votes[i][j]);
            if (0 < j)
                assume (votes[i][j-1] <= votes[i][j]);
        }
    }

    for (unsigned int i = 0; i < C; i++) res[i] = 0;

    for (unsigned int i = 0; i < V; i++)
        for (unsigned int j = 0; j < C; j++)
            res[j] += votes[i][j];

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

void monotonicity(unsigned int votes1[V][C],
                  unsigned int votes2[V][C],
                  unsigned int m,
                  unsigned int n) {
    assume (0 < m && m <= C);
    assume (0 <= n && n < V);

    for (unsigned int i = 0; i < V; i++) {
        for (unsigned int j = 0; j < C; j++) {
            assume (0 <= votes1[i][j]);
            assume (0 <= votes2[i][j]);
            assume (votes1[i][j] <= 1);
            assume (votes2[i][j] <= 1);
            if (0 < i)
                assume (votes1[i-1][j] <= votes1[i][j]);
            if (0 < j)
                assume (votes1[i][j-1] <= votes1[i][j]);
            if ((i != n) || (j != (m-1)))
                assume (votes1[i][j] == votes2[i][j]);
        }
    }

    assume (votes1[n][m-1] == 0);
    assume (votes2[n][m-1] == 1);

    unsigned int elect1 = voting(votes1);
    unsigned int elect2 = voting(votes2);
    if (m == elect1) assert (elect1 == elect2);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V][C];
    unsigned int v2[V][C];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < C; j++) {
            v1[i][j] = __llbmc_nondef_unsigned_int();
            v2[i][j] = __llbmc_nondef_unsigned_int();
        }
    }
    unsigned int vote  = __llbmc_nondef_unsigned_int();
    unsigned int voter = __llbmc_nondef_unsigned_int();

    monotonicity(v1, v2, vote, voter);
    return 0;
}
```

Listing A.10: LLBMC Specification for Anonymous and Neutral Approval Voting

A.4. Instant-Runoff Voting for LLBMC

In Listing A.11, the first argument is the number of candidates, the second one the number of voters, the third one the filename (without file extension), and the fourth one is for optional additional parameters.

```
#!/bin/bash
clang -D C=$1 -D V=$2 -c -g -emit-llvm $3.c -o $3.bc
llbmc -log-level=verbose -only-custom-assertions  -no-max-
   loop-iterations-checks --max-loop-iterations=$1 $4 $3.bc
```

Listing A.11: Script with Parameters for LLBMC Call of Instant-Runoff Voting

A.4.1. Implementation of Deterministic IRV

```
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 2
#endif

#ifndef C
#define C 2
#endif

unsigned int voting(unsigned int votes[V][C+1]) {
    unsigned int res = 0;
    unsigned int count[C+1];
    unsigned int cc = C;
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, l = 0;

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }
        }
    }

    unsigned int quota = 0;
    if (V % 2 != 0) quota = (V - 1) / 2;
    else quota = V / 2;

    unsigned int min = quota;
    while (res == 0 && 0 < cc) {
        for (i = 0; i <= C; i++) count[i] = 0;
        for (i = 0; i < V; i++)
            for (j = 1; j <= C; j++)
                if (votes[i][1] == j) count[j]++;
        for (i = 1; i <= C && res == 0; i++)
            if (quota < count[i]) res = i;
        if (res == 0) {
            min = quota;
            for (i = 1; i <= C; i++)
                if (count[i] < min && count[i] != 0)
                    min = count[i];
            for (i = 1; i <= C; i++) {
                if (count[i] == min) {
                    for (j = 0; j < V; j++) {
                        for (k = 1; k <= C; k++) {
                            if (votes[j][k] == i) {
                                for (l = k; l < C; l++)
                                    votes[j][l]
                                        = votes[j][l + 1];
                                votes[j][C] = 0;
                            }
                        }
                    }
                    cc--;
                }
            }
        }
    }
    return res;
}
```

Listing A.12: LLBMC Specification for Deterministic IRV

A.4.2. Implementation of Exhaustive IRV

```c
#include <stdlib.h>
#include "llbmc.h"

#ifndef V
#define V 2
#endif

#ifndef C
#define C 2
#endif

unsigned int voting(unsigned int votes[V][C+1]) {
    unsigned int res = 0, weakest = 0, choose = 0;
    unsigned int count[C+1];
    unsigned int cc = C;
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, l = 0;

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }
        }
    }

    unsigned int quota = 0;
    if (V % 2 != 0) quota = (V - 1) / 2;
    else quota = V / 2;

    unsigned int min = quota;
    while (res == 0 && 0 < cc) {
        for (i = 0; i <= C; i++) count[i] = 0;
        for (i = 0; i < V; i++)
            for (j = 1; j <= C; j++)
                if (votes[i][1] == j) count[j]++;
        for (i = 1; i <= C && res == 0; i++)
            if (quota < count[i]) res = i;
        if (res == 0) {
            min = quota;
            weakest = 0;
            for (i = 1; i <= C; i++) {
                if (count[i] < min && count[i] != 0) {
                    min = count[i];
                    weakest = 1;
                } else if (count[i] == min) weakest++;
            }
            choose = __llbmc_nondef_unsigned_int();
            assume (0 < choose && choose <= weakest);

            weakest = 0;
            for (i = 1; i <= C; i++) {
                if (count[i] == min) weakest++;
                if (count[i] == min && weakest == choose) {
                    for (j = 0; j < V; j++) {
                        for (k = 1; k <= C; k++) {
                            if (votes[j][k] == i) {
                                for (l = k; l < C; l++)
                                    votes[j][l]
                                        = votes[j][l + 1];
                                votes[j][C] = 0;
                            }
                        }
                    }
                    cc--;
                }
            }
        }
    }
    return res;
}
```

Listing A.13: LLBMC Specification for Exhaustive IRV

A.4.3. Specification of Condorcet Loser Criterion for IRV

```
void condorcet_loser(unsigned int votes[V][C+1],
                     unsigned int loser) {
    assume (0 < loser && loser <= C);
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, tmp = 0;
    int defeat[C+1], counted[C+1];
    unsigned int found = 0;
    for (i = 0; i <= C; i++) {
        defeat[i] = 0;
        counted[i] = 0;
    }

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        found = 0;
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }

            if (votes[i][j] == loser) found = 1;
            if (votes[i][j] != 0) {
                tmp = votes[i][j];
                defeat[tmp] += (found ? (-1) : 1);
                counted[tmp] = 1;
            }
        }

        for (k = 1; k <= C; k++) {
            if (counted[k] == 0) {
                if (found) defeat[k]--;
            } else counted[k] = 0;
        }
    }

    defeat[loser] = V + 1;
    for (i = 1; i <= C; i++) assume (0 < defeat[i]);

    unsigned int elect = voting(votes);
    assert (elect != loser);
}
```

```
int main(int argc, char *argv[]) {
    unsigned int v1[V][C+1];
    unsigned int i = 0, j = 0;
    for (i = 0; i < V; i++) {
        v1[i][0] = 0;
        for (j = 1; j <= C; j++)
            v1[i][j] = __llbmc_nondef_unsigned_int();
    }
    unsigned int cand  = __llbmc_nondef_unsigned_int();

    condorcet_loser(v1, cand);
    return 0;
}
```

Listing A.14: LLBMC Specification of Condorcet Loser Criterion for IRV

A.4.4. Specification of Condorcet Winner Criterion for IRV

```
void condorcet_winner(unsigned int votes[V][C+1],
                      unsigned int winner) {
    assume (0 < winner && winner <= C);
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, tmp = 0;
    int defeat[C+1], counted[C+1];
    unsigned int found = 0;
    for (i = 0; i <= C; i++) {
        defeat[i] = 0;
        counted[i] = 0;
    }

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        found = 0;
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }

```

```
                if (votes[i][j] == winner) found = 1;
                if (votes[i][j] != 0) {
                    tmp = votes[i][j];
                    defeat[tmp] += (found ? 1 : (-1));
                    counted[tmp] = 1;
                }
            }

            for (k = 1; k <= C; k++) {
                if (counted[k] == 0) {
                    if (found) defeat[k]++;
                } else counted[k] = 0;
            }
    }

    defeat[winner] = 1;
    for (i = 1; i <= C; i++) assume (0 < defeat[i]);

    unsigned int elect = voting(votes);
    assert (elect == 0 || elect == winner);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V][C+1];
    unsigned int i = 0, j = 0;
    for (i = 0; i < V; i++) {
        v1[i][0] = 0;
        for (j = 1; j <= C; j++)
            v1[i][j] = __llbmc_nondef_unsigned_int();
    }
    unsigned int cand  = __llbmc_nondef_unsigned_int();

    condorcet_winner(v1, cand);
    return 0;
}
```

Listing A.15: LLBMC Specification of Condorcet Winner Criterion for IRV

A.5. Single Transferable Vote for LLBMC

In Listing A.16, the first argument is the number of candidates, the second one the number of seats, the third one the number of voters, the fourth one the filename (without file extension), and the fifth one is for optional additional parameters.

```
#!/bin/bash
clang -D S=$2 -D C=$1 -D V=$3 -c -g -emit-llvm $4.c -o $4.bc
llbmc -log-level=verbose -only-custom-assertions -no-max-
   loop-iterations-checks --max-loop-iterations=$(($1 * $2))
    $5 $4.bc
```

Listing A.16: Script with Parameters for LLBMC Call of STV

```
#include <stdlib.h>
#include "llbmc.h"

#ifndef C
#define C 2
#endif

#ifndef V
#define V 2
#endif

#ifndef S
#define S 1
#endif

unsigned int *voting(unsigned int votes[V][C+1]) {
    unsigned int *r = malloc(S*sizeof(unsigned int));
    unsigned int res = 0;

    unsigned int count[C+1];
    unsigned int cc = C, e = 0, weakest = 0, choose = 0;
    unsigned int i = 0, j = 0, j_prime = 0,
                 k = 0, l = 0, t = 0;

    for (i = 0; i < S; i++) r[i] = 0;

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }
        }
    }

    unsigned int quota = 0;
    if (V % 2 != 0) quota = (V - 1) / (S + 1);
    else quota = V / (S + 1);

    unsigned int min = quota;
    while (res == 0 && 0 < cc && e < S && (S - e) < cc) {
        for (i = 0; i <= C; i++) count[i] = 0;
        for (i = 0; i < V; i++)
            for (j = 1; j <= C; j++)
                if (votes[i][1] == j) count[j]++;
        for (i = 1; i <= C && res == 0; i++)
            if (quota < count[i]) res = i;

        if (res != 0) {
            r[e] = res;
            e++;
            for (t = 0; t <= quota; t++) {
                i = 0;
                while (votes[i][1] != res) i++;
                for (j = 1; j <= C; j++) votes[i][j] = 0;
            }
            for (j = 0; j < V; j++) {
                for (k = 1; k <= C; k++) {
                    if (votes[j][k] == res) {
                        for (l = k; l < C; l++)
                            votes[j][l]
                                = votes[j][l + 1];
                        votes[j][C] = 0;
                    }
                }
            }
            res = 0;
            cc--;
        } else {
            min = quota;
            weakest = 0;
            for (i = 1; i <= C; i++) {
                if (count[i] < min && count[i] != 0) {
                    min = count[i];
                    weakest = 1;
                } else if (count[i] == min) weakest++;
            }
            choose = __llbmc_nondef_unsigned_int();
            assume (0 < choose && choose <= weakest);

            weakest = 0;
            for (i = 1; i <= C; i++) {
                if (count[i] == min) weakest++;
                if (count[i] == min && weakest == choose) {
                    for (j = 0; j < V; j++) {
                        for (k = 1; k <= C; k++) {
                            if (votes[j][k] == i) {
                                for (l = k; l < C; l++)
                                    votes[j][l]
                                        = votes[j][l + 1];
                                votes[j][C] = 0;
                            }
                        }
                    }
                    cc--;
                }
            }
        }
    }

    if (e < S - 1) {
        for (i = e; i < S && 0 < cc; i++) {
            res = 0;
            for (k = 1; k <= C && res == 0; k++)
                for (j = 0; j < V && res == 0; j++)
                    if (votes[j][1] == k) res = k;
            r[i] = res;
            for (j = 0; j < V; j++) {
                for (k = 1; k <= C; k++) {
                    if (votes[j][k] == res) {
                        for (l = k; l < C; l++)
                            votes[j][l] = votes[j][l + 1];
                        votes[j][C] = 0;
                    }
                }
            }
            cc--;
        }
    }
    return r;
}
```

Listing A.17: LLBMC Specification for STV

A.5.1. Specification of Condorcet Loser Criterion

```
void condorcet_loser(unsigned int votes[V][C+1],
                     unsigned int loser) {
    assume (0 < loser && loser <= C);
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, tmp = 0;
    int defeat[C+1], counted[C+1];
    unsigned int found = 0;
    for (i = 0; i <= C; i++) {
        defeat[i] = 0;
        counted[i] = 0;
    }

    for (i = 0; i < V; i++) {
        assume (votes[i][1] != 0);
        found = 0;
        for (j = 1; j <= C; j++) {
            assume (0 <= votes[i][j]);
            assume (votes[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes[i][j] != 0) && (j != j_prime))
                    assume (votes[i][j]
                        != votes[i][j_prime]);
                if ((votes[i][j] == 0) && (j <= j_prime))
                    assume (votes[i][j_prime] == 0);
            }

            if (votes[i][j] == loser) found = 1;
            if (votes[i][j] != 0) {
                tmp = votes[i][j];
                defeat[tmp] += (found ? (-1) : 1);
                counted[tmp] = 1;
            }
        }

        for (k = 1; k <= C; k++) {
            if (counted[k] == 0) {
                if (found) defeat[k]--;
            } else counted[k] = 0;
        }
    }

    defeat[loser] = V + 1;
    for (i = 1; i <= C; i++) assume (0 < defeat[i]);

    unsigned int *elect = voting(votes);
    for (i = 0; i < S; i++)
        assert (C == S || elect[i] != loser);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V][C+1];
    unsigned int i = 0, j = 0;
    for (i = 0; i < V; i++) {
        v1[i][0] = 0;
        for (j = 1; j <= C; j++)
            v1[i][j] = __llbmc_nondef_unsigned_int();
    }
    unsigned int cand  = __llbmc_nondef_unsigned_int();

    condorcet_loser(v1, cand);
    return 0;
}
```

Listing A.18: LLBMC Specification of Condorcet Loser Criterion for STV

A.5.2. Specification of Condorcet Winner Criterion

```
void condorcet_winner(unsigned int votes1[V][C+1],
                      unsigned int winner) {
    assume (0 < winner && winner <= C);
    unsigned int i = 0, j = 0, j_prime = 0, k = 0, tmp = 0;
    int defeat[C+1], counted[C+1];
    unsigned int found = 0;
    for (i = 0; i <= C; i++) {
        defeat[i] = 0;
        counted[i] = 0;
    }

    for (i = 0; i < V; i++) {
        assume (votes1[i][1] != 0);
        found = 0;
        for (j = 1; j <= C; j++) {
            assume (0 <= votes1[i][j]);
            assume (votes1[i][j] <= C);
            for (j_prime = 1; j_prime <= C; j_prime++) {
                if ((votes1[i][j] != 0) && (j != j_prime))
                    assume (votes1[i][j]
                        != votes1[i][j_prime]);
                if ((votes1[i][j] == 0) && (j <= j_prime))
                    assume (votes1[i][j_prime] == 0);
            }

            if (votes1[i][j] == winner) found = 1;
            if (votes1[i][j] != 0) {
                tmp = votes1[i][j];
                defeat[tmp] += (found ? 1 : (-1));
                counted[tmp] = 1;
            }
        }

        for (k = 1; k <= C; k++) {
            if (counted[k] == 0) {
                if (found) defeat[k]++;
            } else counted[k] = 0;
        }
    }

    defeat[winner] = 1;
    for (i = 1; i <= C; i++) assume (0 < defeat[i]);

    unsigned int *elect = voting(votes1);
    int no_elects = 1;
    int cond_winner = 0;
    for (i = 0; i < S; i++) {
        no_elects = no_elects && (elect[i] == 0);
        cond_winner = cond_winner || elect[i] == winner;
    }
    assert (no_elects || cond_winner);
}

int main(int argc, char *argv[]) {
    unsigned int v1[V][C+1];
    unsigned int i = 0, j = 0;
    for (i = 0; i < V; i++) {
        v1[i][0] = 0;
        for (j = 1; j <= C; j++)
            v1[i][j] = __llbmc_nondef_unsigned_int();
    }
    unsigned int cand  = __llbmc_nondef_unsigned_int();

    condorcet_winner(v1, cand);
    return 0;
}
```

Listing A.19: LLBMC Specification of Condorcet Winner Criterion for STV

References

[ADK03] Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid. *Evaluating the "Small Scope Hypothesis"*. Tech. rep. Cambridge, MA: MIT Laboratory for Computer Science, 2003 (cit. on pp. 26, 29, 51).

[AK11] Fuad Aleskerov and Alexander Karpov. *A New Method of the Single Transferable Vote and its Axiomatic Justification*. Tech. rep. Moscow, Russia: National Research University Higher School of Economics, Dec. 2011 (cit. on p. 89).

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. "Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers". In: *STTT* 11.1 (2009), pp. 69–83. DOI: `10.1007/s10009-008-0091-0` (cit. on p. 90).

[Arr50] Kenneth J. Arrow. "A Difficulty in the Concept of Social Welfare". In: *Journal of Political Economy* 58.4 (1950), pp. 328–346. ISSN: 00223808. URL: `http://www.jstor.org/stable/1828886` (cit. on p. 19).

[Arr51] Kenneth Joseph Arrow. *Social Choice and Individual Values*. Monographs / Cowles Commission for Research in Economics ; 12. New York: Wiley [u.a.], 1951 (cit. on pp. iii, v, 2, 7).

[Bar+06] Mike Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. Boer et al. Vol. 4111. LNCS. Springer Berlin Heidelberg, 2006, pp. 364–387. ISBN: 978-3-540-36749-9. DOI: `10.1007/11804192_17` (cit. on p. 29).

[Bas14] Anna Bassi. "Voting Systems and Strategic Manipulation: An Experimental Study". In: *Journal of Theoretical Politics* (2014) (cit. on p. 89).

[BCE13] Felix Brandt, Vincent Conitzer, and Ulle Endriss. "Computational Social Choice". In: *Multiagent Systems*. Ed. by G. Weiss. MIT Press, 2013, pp. 213–283 (cit. on pp. 5–7, 89).

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. LNCS. Springer, 2011, pp. 200–214. ISBN: 978-3-642-21436-3. DOI: `10.1007/978-3-642-21437-0_17` (cit. on p. 90).

[Bec+12] Bernhard Beckert et al. “Integration of Bounded Model Checking and Deductive Verification”. In: *Formal Verification of Object-Oriented Software, International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. Vol. 7421. LNCS. Springer, 2012, pp. 86–104 (cit. on p. 90).

[Bec+14a] Bernhard Beckert et al. “Reasoning About Vote Counting Schemes Using Light-Weight and Heavy-Weight Methods”. In: *Proceedings, 8th International Verification Workshop (VERIFY) in connection with IJCAR 2014 at FLoC 2014, July 23–24, 2014, Vienna, Austria*. To appear. 2014 (cit. on p. 2).

[Bec+14b] Bernhard Beckert et al. “Verifying Voting Schemes”. In: *Journal of Information Security and Applications* 19.2 (2014), pp. 115–129 (cit. on p. 90).

[BF88] Steven J. Brams and Peter C. Fishburn. “Does Approval Voting Elect the Lowest Common Denominator?” In: *PS: Political Science & Politics* 21.02 (1988), pp. 277–284 (cit. on pp. 23, 90).

[BGB13] Bundesgesetzblatt 2013 1/22. *Zweiundzwanzigstes Gesetz zur Änderung des Bundeswahlgesetzes*. May 2013. URL: `http://www.bgbl.de/banzxaver/bgbl/text.xav?start=//*[@node_id='251990']&skin=pdf` (cit. on pp. iii, v, 1).

[BGS13] Bernhard Beckert, Rajeev Goré, and Carsten Schürmann. “Analysing Vote Counting Algorithms Via Logic and its Application to the CADE Election System”. In: *Proceedings, 24th International Conference on Automated Deduction (CADE), Lake Placid, NY, USA*. Ed. by Maria Paola Bonacina. LNCS 7898. Springer, 2013, pp. 135–144. DOI: `10.1007/978-3-642-38574-2_9` (cit. on pp. 17, 76).

[Bie+99] Armin Biere et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W.Rance Cleaveland. Vol. 1579. LNCS 1597. Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-65703-3. DOI: `10.1007/3-540-49059-0_14` (cit. on pp. 26, 28).

[BLS05] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. “The Spec# Programming System: An Overview”. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 2005, pp. 49–69 (cit. on p. 31).

[BTD12] Bundestagsdrucksache 17/11819. *Entwurf eines Zweiundzwanzigsten Gesetzes zur Änderung des Bundeswahlgesetzes*. Dec. 2012. URL: `http://dipbt.bundestag.de/dip21/btd/17/118/1711819.pdf` (cit. on p. 1).

[CM05] Marco Cadoli and Toni Mancini. “Using a Theorem Prover for Reasoning on Constraint Problems”. In: *AI* IA 2005: Advances in Artificial Intelligence*. Springer, 2005, pp. 38–49 (cit. on p. 90).

[Coc12] Dermot Cochran. “Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms”. PhD thesis. IT-Universitetet i København, IT University of Copenhagen, 2012 (cit. on p. 89).

[Coh+09] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. LNCS. Springer Berlin Heidelberg, 2009, pp. 23–42. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_2 (cit. on pp. 29, 32, 33).

[Coh+11] Ernie Cohen et al. *Verifying C Programs: A VCC Tutorial*. Tech. rep. MSR Redmond, EMIC Aachen, 2011 (cit. on p. 32).

[Con85] Marquis de Condorcet. *Essay on the Application of Analysis to the Probability of Majority Decisions*. Tech. rep. 1785 (cit. on p. 21).

[Cra+96] James M. Crawford et al. “Symmetry-Breaking Predicates for Search Problems”. In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR’96), Cambridge, Massachusetts, USA, November 5-8, 1996*. Ed. by Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro. Morgan Kaufmann, 1996, pp. 148–159. ISBN: 1-55860-421-9 (cit. on p. 79).

[CS12] Vincent Conitzer and Tuomas Sandholm. “Common Voting Rules as Maximum Likelihood Estimators”. In: *CoRR* (2012). URL: http://arxiv.org/abs/1207.1368 (cit. on p. 3).

[Dar+04] Paul T. Darga et al. “Exploiting Structure in Symmetry Detection for CNF”. In: *Proceedings of the 41st Annual Design Automation Conference*. DAC ’04. San Diego, CA, USA: ACM, 2004, pp. 530–534. ISBN: 1-58113-828-8. DOI: 10.1145/996566.996712 (cit. on p. 90).

[DB07] Leonardo De Moura and Nikolaj Bjørner. “Efficient E-matching for SMT Solvers”. In: *Automated Deduction–CADE-21*. Springer, 2007, pp. 183–198 (cit. on p. 31).

[DB11] Leonardo De Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: Introduction and Applications”. In: *Communications of the ACM* 54.9 (2011), pp. 69–77 (cit. on p. 27).

[DCJ06] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. “Modular Verification of Code with SAT”. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM. 2006, pp. 109–120 (cit. on p. 90).

[DL05] Robert DeLine and K. Rustan M. Leino. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Tech. rep. Microsoft Research, Mar. 2005 (cit. on p. 29).

[Dre72] John Dreijmanis. “Maurice Duverger, Party Politics and Pressure Groups: A Comparative Introduction, trans. David Wagoner. New York: Thomas Y. Crowell Company [Don Mills: Fitzhenry & Whiteside, Ltd..], 1972, pp. vii, 168”. In: *Canadian Journal of Political Science* 5 (03 Sept. 1972), pp. 459–460. ISSN: 1744-9324. DOI: `10.1017/S0008423900034806` (cit. on p. 11).

[Dut00] Bhaskar Dutta. “Amartya Sen and the Mathematics of Collective Choice”. In: *Resonance* 5.6 (2000), pp. 97–103. ISSN: 0971-8044. DOI: `10.1007/BF02833861` (cit. on p. 7).

[DYJ08] Greg Dennis, Kuat Yessenov, and Daniel Jackson. “Bounded Verification of Voting Software”. In: *VSTTE*. Ed. by Natarajan Shankar and Jim Woodcock. Vol. 5295. Lecture Notes in Computer Science. Springer, 2008, pp. 130–145. ISBN: 978-3-540-87872-8 (cit. on pp. 3, 90).

[EFS10] Edith Elkind, Piotr Faliszewski, and Arkadii M. Slinko. “On The Role of Distances in Defining Voting Rules”. In: *AAMAS*. Ed. by Wiebe van der Hoek et al. IFAAMAS, 2010, pp. 375–382. ISBN: 978-0-9826571-1-9. URL: `http://dblp.uni-trier.de/db/conf/atal/aamas2010.html` (cit. on pp. 3, 90).

[ES03] Niklas Eén and Niklas Sörensson. “An Extensible SAT-Solver”. In: *SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. LNCS. Springer, 2003, pp. 502–518. ISBN: 3-540-20851-8. DOI: `10.1007/978-3-540-24605-3_37` (cit. on p. 34).

[FB83] Peter C. Fishburn and Steven J. Brams. “Paradoxes of Preferential Voting”. In: *Mathematics Magazine* (1983), pp. 207–214 (cit. on p. 5).

[Fel12a] Dan S. Felsenthal. *Electoral Systems : Paradoxes, Assumptions, and Procedures*. Ed. by Moshé Machover. Studies in Choice and Welfare. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-20441-8. DOI: `10.1007/978-3-642-20441-8`.

[Fel12b] Dan S. Felsenthal. “Review of Paradoxes Afflicting Procedures for Electing a Single Candidate”. In: *Electoral Systems*. Ed. by Dan S. Felsenthal and Moshé Machover. Studies in Choice and Welfare. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 19–91. ISBN: 978-3-642-20440-1. DOI: `10.1007/978-3-642-20441-8_3` (cit. on pp. 5, 17, 24, 61, 72).

[Gal13] Michael Gallagher. *Monotonicity and Non-Monotonicity at PR-STV Elections*. 2013 (cit. on pp. 20, 90).

[GD07] Vijay Ganesh and David L. Dill. “A Decision Procedure for Bit-Vectors and Arrays”. In: *CAV*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 519–531. ISBN: 978-3-540-73367-6. DOI: `10.1007/978-3-540-73368-3_52` (cit. on p. 34).

[GL94] O. Grumberg and D.E. Long. “Model Checking and Modular Verification”. In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1994), pp. 843–871. URL: `citeseer.ist.psu.edu/grumberg91model.html` (cit. on p. 90).

[HN09] Yuusuke Hashimoto and Shin Nakajima. “Modular Checking of C Programs Using SAT-Based Bounded Model Checker”. In: *APSEC*. Ed. by Shahida Sulaiman and Noor Maizura Mohamad Noor. IEEE Computer Society, 2009, pp. 515–522. ISBN: 978-0-7695-3909-6. URL: `http://dblp.uni-trier.de/db/conf/apsec/apsec2009.html#HashimotoN09` (cit. on p. 90).

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 978-0-262-10114-1. URL: `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928` (cit. on p. 79).

[JD96] Daniel Jackson and Craig A Damon. “Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector”. In: *Software Engineering, IEEE Transactions on* 22.7 (1996), pp. 484–495 (cit. on p. 28).

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004 (cit. on p. 34).

[Lij87] Arend Lijphart. “The Demise of the Last Westminster System? Comments on the Report of New Zealand’s Royal Commission on the Electoral System”. In: *Electoral Studies* 6.2 (1987), pp. 97–103. ISSN: 0261-3794. DOI: `10.1016/0261-3794(87)90016-3` (cit. on p. 3).

[LM10] K Rustan M Leino and Michał Moskal. “Usable Auto-Active Verification”. In: Workshop on Usable Verification (Nov. 2010) (cit. on pp. 26, 27).

[LN95] Jonathan Levin and Barry Nalebuff. “An Introduction to Vote-Counting Schemes”. In: *The Journal of Economic Perspectives* (1995), pp. 3–26 (cit. on pp. 9, 10).

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS’08*. 2008, pp. 337–340 (cit. on pp. 29, 31).

[MC05] Toni Mancini and Marco Cadoli. “Detecting and Breaking Symmetries by Reasoning on Problem Specifications”. In: *Abstraction, Reformulation and Approximation*. Springer, 2005, pp. 165–181 (cit. on p. 90).

[Meu14] Thomas David Meumann. “Complexity in Electoral Systems: Proving Correctness Using HOL4”. Bachelor Thesis (Honours). Australian National University, Aug. 2014 (cit. on pp. 2, 90).

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR”. In: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Vol. 7152. LNCS. Springer, 2012, pp. 146–161. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_12 (cit. on pp. 33, 60).

[Mil07] Nicholas R. Miller. “The Butterfly Effect Under STV”. In: *Electoral Studies* 26.2 (2007), pp. 503–506. ISSN: 0261-3794. DOI: 10.1016/j.electstud.2006.10.016 (cit. on p. 5).

[Mil72] Robin Milner. *Logic for Computable Functions: Description of a Machine Implementation*. Tech. rep. DTIC Document, 1972 (cit. on p. 26).

[Nur12] Hannu Nurmi. “On the Relevance of Theoretical Results to Voting System Choice”. In: *Electoral Systems*. Ed. by Moshé Machover. Studies in Choice and Welfare. Berlin, Heidelberg: Springer, 2012, pp. 255–274. ISBN: 978-3-642-20441-8. DOI: 10.1007/978-3-642-20441-8 (cit. on pp. 19, 89).

[Nur96] Hannu Nurmi. “It’s Not Just the Lack of Monotonicity”. In: *Representation* 34.1 (1996), pp. 48–52. DOI: 10.1080/00344899608522986 (cit. on p. 5).

[Pac12] Eric Pacuit. “Voting Methods”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2012. 2012 (cit. on p. 90).

[Pnu+02] Amir Pnueli et al. “The Small Model Property: How Small Can It Be?” In: *Information and Computation* 178.1 (2002), pp. 279–293 (cit. on p. 29).

[RG98] Michel Regenwetter and Bernard Grofman. “Approval Voting, Borda Winners, and Condorcet Winners: Evidence from Seven Elections”. In: *Management Science* 44.4 (1998), pp. 520–533 (cit. on pp. 23, 90).

[Sen70] Amartya Kumar Sen. “Collective Rationality”. In: *Collective Choice and Social Welfare*. Mathematical Economics Texts ; 5. San Francisco [u.a.]: Holden-Day, 1970. Chap. 3, pp. 33–46. ISBN: 0-8162-7765-6 (cit. on p. 7).

[SFM10] Carsten Sinz, Stephan Falke, and Florian Merz. “A Precise Memory Model for Low-Level Bounded Model Checking”. In: *Proceedings of the 5th International Conference on Systems Software Verification*. USENIX Association. 2010 (cit. on p. 34).

[Shl07] Ilya Shlyakhter. “Generating Effective Symmetry-Breaking Predicates for Search Problems”. In: *Discrete Applied Mathematics*. {SAT} 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing 155.12 (2007), pp. 1539–1548. ISSN: 0166-218X. DOI: 10.1016/j.dam.2005.10.018 (cit. on p. 90).

[TBL10] Edd Turner, Michael J. Butler, and Michael Leuschel. "A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking". In: *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings.* Ed. by Marc Frappier et al. Vol. 5977. LNCS. Springer, 2010, pp. 231–244. ISBN: 978-3-642-11810-4. DOI: `10.1007/978-3-642-11811-1_18` (cit. on p. 79).

[Tur+07] Edd Turner et al. "Symmetry Reduced Model Checking for B". In: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China.* IEEE Computer Society, 2007, pp. 25–34. DOI: `10.1109/TASE.2007.50` (cit. on p. 79).

[Uni13] Economist Intelligence Unit. *Democracy Index 2012: Democracy is at a Standstill.* Economist Intelligence Unit Limited, 2013 (cit. on p. 1).

[Woo97] Douglas R. Woodall. "Monotonicity of Single-Seat Preferential Election Rules". In: *Discrete Applied Mathematics* 77.1 (1997), pp. 81–98. URL: `http://dblp.uni-trier.de/db/journals/dam/dam77.html#Woodall97` (cit. on p. 19).

[YS97] Hongji Yang and Yong Sun. "1st Irish Workshop on Formal Methods". In: *Proceedings of the 1st Irish Workshop on Formal Methods.* Vol. 3. 1997, p. 4 (cit. on p. 89).