Automatic Margin Computation for Risk-Limiting Audits

Bernhard Beckert[1], Michael Kirsten[1](✉), Vladimir Klebanov[1], and Carsten Schürmann[2]

[1] Institute of Theoretical Informatics, Am Fasanengarten 5, 76131 Karlsruhe, Germany
{beckert,kirsten,klebanov}@kit.edu
[2] IT University of Copenhagen (ITU), Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
carsten@itu.dk

Abstract. A risk-limiting audit is a statistical method to create confidence in the correctness of an election result by checking samples of paper ballots. In order to perform an audit, one usually needs to know what the election margin is, i.e., the number of votes that would need to be changed in order to change the election outcome.

In this paper, we present a fully automatic method for computing election margins. It is based on the program analysis technique of bounded model checking to analyse the implementation of the election function. The method can be applied to arbitrary election functions without understanding the actual computation of the election result or without even intuitively knowing how the election function works.

We have implemented our method based on the model checker CBMC; and we present a case study demonstrating that it can be applied to real-world elections.

Keywords: Risk-limiting audit · Margin computation · Software bounded model checking · Static analysis

1 Introduction

One reliable method to create confidence in the outcome of an election among the electorate is to audit the election result against the physical evidence, i.e., the ballots. Different methods for auditing elections exist, some of them require the computation of a margin, that is the minimal number of ballots to be changed, misfiled, etc. to affect the election outcome. For those methods, the precise definition of the margin is often hidden inside the theory, as it depends on the election function—or social choice function—and the particular auditing methodology. This means, that (1) for many election functions, including Ranked Choice Voting (RCV) and Single Transferable Vote (STV), or election functions that combine different electoral systems, for example on state and federal level, it is difficult if not impossible to give closed forms for how to compute a margin, and

(2) even if one manages to find a closed form for how to compute the margin, the implementations of election function and margin computation differ, for example in the way ambiguities are resolved, when and how to which precision to round, how tie-breaking rules are implemented, etc.

In this paper, we focus on auditing methods that require the margins to be known before they can be applied. Examples of these methods are, e.g., risk-limiting audits that draw a random sample of paper ballots [14] whose size is computed from (a) a risk-limit, i.e., how confident we wish to be in the election result, and (b) the margin. For a *comparision audit*, the margin of a risk-limiting audit is defined as the minimal number of votes that would need to be misfiled in order to change the election outcome. The margin is identical to the number of votes that would have had to be miscounted or tampered with during tabulation. If the election margin is large, only a small sample needs to be drawn and audited. The smaller the margin, the larger the sample. In the worst case, the audit will trigger a full manual recount.

We describe a way to compute the margins that does not presuppose the existence of a closed form for the margin and works directly on the source code (e.g., written in C/C++). Our technique can be applied to any election function, but it will perform best on those that are conceptually simple, such as D'Hondt and Sainte-Laguë. The technique can in principle also be applied to more complex election functions, such as instant-runoff voting (IRV), but only for small elections with a small number of seats and candidates. For bigger elections, such as the national elections in Australia, our technique does not scale – yet.

Our technique takes advantage of the state-of-the-art in program analysis, in particular software bounded model checking (SBMC). We compute the margin directly from the implementation of the election function. The trick is to use software bounded model checking for determining whether tampering with (at most) n votes can lead to a change in the election result. If yes, we have found an upper bound for the margin; and, if no, we have found a lower bound. The model checker is then called iteratively with different values for n, using binary search to determine the exact value of the margin. Our method is agnostic to the mathematics behind the election function, and the statistics behind the audit sample size computations. It can be applied to arbitrary C/C++ implementations of election functions without understanding the actual computation of the election result or without even intuitively knowing how the election function works.

Contents of this Paper. In Sect. 2, we recapitulate the idea of risk-limiting audits and describe how election margins influence the audit; and in Sect. 3, we give an introduction to software bounded model checking. Then, in Sect. 4, we introduce our method that, based on SBMC, allows to automatically compute election margins for arbitrary election functions. In Sect. 5, we illustrate our approach using an election function based on the D'Hondt method. An extension that leads to increased efficiency is described in Sect. 6. In Sect. 7, we present a case study where we apply our method to compute the election margin for the main part of the 2015 Danish national parliamentary elections. Finally, in Sect. 8, we draw conclusions and discuss future work.

Related Work. The contribution of our paper is a *generic* method that infers election margins for any election function, for which an implementation is available. In contrast to our work, there has been a lot of research on how to compute margins for *specific* election functions, for which that problem is particularly hard. The most prominent example is Instant-Runoff Voting (IRV) where margin computation is NP-hard [2]. Methods for computing lower bounds on margins for IRV have been developed by Cary [6] and Sarwate et al. [16]; and methods for computing the exact margin have been presented by Magrino et al. [15] and, recently, by Blom et al. [5].

To compute the margin of an election is an instance of the general problem of inverting a function for which an implementation is given, i.e., to ask for an input to the implementation that leads to a particular kind of output. The idea of using model checkers for solving such problems has also been applied in the field of test-case generation, where one is looking for input values leading to some specific program behaviour [20]. For example, the software model checker CBMC has been integrated into the extensive test-suite FShell [12]. Similar techniques have been used for generating high-quality game content, such as well-designed puzzles that are hard to solve [17].

In the context of elections, SBMC with SAT/SMT solvers can furthermore be used for analysing, whether the given election function does indeed compute the correct result with respect to some given formal criteria [3].

2 Risk-Limiting Audits and Election Margins

A risk-limiting audit is a statistical method to create confidence in the correctness of an election result by checking samples of paper ballots. Lindeman and Stark [14] distinguish *ballot-polling audits*, where they draw a carefully chosen random sample of ballots to check whether the sample gives sufficiently strong evidence for the correctness of the published election result. In contrast, a *comparison audit* checks the ballot interpretation for a random sample during the audit against the ballot's respective interpretation in a vote-tabulation system.

Both auditing techniques, ballot-polling and comparison audits, rely on the availability of the ballot manifest which describes in detail how the ballots are organised and stored, including how many stacks there are and how many ballots can be found in each stack. This information is needed for drawing the sample.

In addition, one needs to know what the *election margin* is, i.e., the number of votes that would need to be changed in order to change the election outcome. This is also the number of votes that would have had to be miscounted or tampered with in order to change the election outcome. If the election margin is large, only a small ballot sample needs to be audited. If it is small, the required sample size increases.

We assume that the election function we consider has the anonymity property, i.e., identical ballots have the same effect on the election outcome. Then, for a given election with `TOTAL` votes, during the counting process, the votes are accumulated into stacks $S_1, \ldots, S_k$, where each stack holds p_i identical votes

($p_i \geq 0$ is the size of S_i) and $\texttt{TOTAL} = \sum_i p_i$. This allows us to use $\langle p_1, \ldots, p_k \rangle$ as input to the election function. In the following, we assume that each stack is associated with a political party and that `PARTIES` is the number of the running parties, i.e., $k = \texttt{PARTIES}$ (there can also be stacks for special cases such as invalid votes). We call $\langle p_1, \ldots, p_k \rangle$ the vote table for the election.

The election margin is the smallest number of votes that need to be put on stacks different from where they are in order to change the outcome of the election.

Definition 1. *The* election margin *for an election function E and a vote table $\langle p_1, \ldots, p_k \rangle$ is the smallest number* `MARGIN` *such that there is a vote table $\langle p'_1, \ldots, p'_k \rangle$ with*

$$E(\langle p_1, \ldots, p_k \rangle) \neq E(\langle p'_1, \ldots, p'_k \rangle)$$

and

1. $\texttt{MARGIN} = \sum_{i=1}^{k} d_i$ *where* $d_i = p'_i - p_i$ *if* $p'_i > p_i$ *and* $d_i = 0$ *otherwise.*
2. $\sum_{i=1}^{k} p'_i - p_i = 0$.

The first condition in the above definition ensures that the total number of votes that are moved between stacks is of size `MARGIN`. Furthermore, the second condition ensures that a vote is moved from one stack to the other and is not created or removed.

Besides the (global) margin defined above, our approach allows as well to compute other margins that are defined by different types of changes in the vote table or by particular effects on the election result. For example, one may compute the margin for increasing the number of mandates allocated to a particular party.

It is important to note that our technique is a *generic* one, and is hence also applicable to different kinds of margins and types of changes in the votes, than the ones defined in Definition 1. Instead of distinguishing between different types, in the following we focus on *two-vote overstatements* of the margin, as these are suitable for a variety of election functions. An audited ballot is a *two-vote overstatement* if it witnesses simultaneously two mistakes, namely that it was counted wrongly towards someone who won, while it should have been counted towards someone who lost. In contrast, a *one-vote overstatement* refers to a ballot that was erroneously not counted towards the loser, but neither was it counted towards the winner. For the purposes of this paper, both one-vote and two-vote overstatements are counted as one change in the vote tabulation. Our methods can be extended to distinguish between the two types of error, but as we want our method for margin computation to be general and the distinction between one-vote and two-vote overstatements does not exist for all election functions (e.g., approval voting), we do not address it within this paper.

Next, we review the statistics underlying margin-based risk-limiting audits following [18]. Risk-limiting audits are performed in stages. At every stage, the theory requires that we audit at least $n = \rho/\mu$ ballots, which is also called the *sample size*. The value ρ is called the *sample-size multiplier* and defined below. Each ballot is randomly chosen among all the ballots, and the audit verifies that they were each counted for the correct stack S_i. The fraction μ refers to the *diluted margin*, i.e., the percentage of votes that would have to be changed to change the election outcome. It is computed as $\mu = {}^{\mathtt{MARGIN}}/_{\mathtt{TOTAL}}$, where `MARGIN` is the election margin (Definition 1), and `TOTAL` is the total number of ballots cast.

Before the audit can start, a set of auditing parameters needs to be determined, which allows us to calculate the size of the sample to be drawn. The *auditing parameters* include

- the *risk-limit* α, which determines the largest chance that an incorrect outcome will not be corrected by the audit (if we want to be 99% sure that the election outcome is correct, then we choose $\alpha = 0.01$);
- the *error inflation factor* γ, which controls the trade-off between initial sample size and the additional counting required if the audit finds too many errors;
- and lastly the *tolerance factor* λ, which describes the tolerance towards errors; it is the number of detected errors that is tolerated, expressed as a fraction of the election margin (i.e., $\lambda = 0.1$ means that 5 errors are tolerated when $\mathtt{MARGIN} = 50$).

Finally, we have everything in place needed to define the sample-size multiplier ρ, which only needs to be computed once for each audit, as follows:

$$\rho = \frac{-\log\alpha}{\frac{1}{2\gamma} + \lambda\log(1 - \frac{1}{2\gamma})}.$$

In summary, the auditing process as described by Stark [18] adheres to the following steps:

First, the auditor commits values for α, γ, and λ and computes the value ρ as shown above. Then, the diluted margin μ is computed, which explicitly depends on the election margin `MARGIN`. Next, the real audit commences by drawing the sample of size $n = \rho/\mu$ at random. If the audit encounters too many errors (more than $\lambda * \mathtt{MARGIN}$), a new stage is triggered, with a sample size that is increased by the factor γ; otherwise the audit is successfully concluded. In the worst case, the technique proceeds to a full hand-count when the sample size exceeds `TOTAL`. For a more detailed description on by how to compute by how much the sample must grow from stage to stage, consult [18].

In all of this, the true challenge is to compute the correct election margin. Different election functions require different margin computations, and for many an algorithm to compute the margin is unknown. This is the challenge that we are going to solve with this paper.

3 Software Bounded Model Checking

The technique of *software bounded model checking* (SBMC) statically analyses programs. The method is static in the sense that programs are analysed without executing them on concrete values. Instead, programs are symbolically executed and exhaustively checked for errors up to a certain bound, restricting the number of loop iterations.

Even though this check is bounded, SBMC also checks whether the chosen bound is sufficiently large to cover all possible program executions. Therefore, if firstly the analysed program is shown to be correct up to the specified bound and, secondly, SBMC verifies that this bound is sufficiently large, we obtain a full proof which says there does not exist any counterexample—neither for the specified nor any other bound. In case there exists no counterexample within the bound, but there may exist one for a larger bound, SBMC outputs that a larger bound is needed. Theoretically, we can always choose a sufficiently high bound to be sure we compute the correct margin. As, however, the analysis for very large bounds may require a considerable amount of computation time and memory resources, the feasibility of SBMC generally relies on the small-scope hypothesis [13], which argues that a high proportion of bugs can be found for inputs within some small scope [1]. For our purposes, moreover, the search for a sufficiently large bound is usually very simple, because we apply the method for concrete elections. Here, the numbers of parties, mandates, etc., affecting the number of required loop iterations, are known at the time when we compute the election margin.

SBMC is a fully automatic technique and provides full verification covering all possible inputs (within the scope of the given bound), including a verification that the specified bound is sufficiently large. An SBMC tool unrolls the control-flow graph of the program observing the bound for loop iterations and then checks whether an assertion can be violated (leading to a counterexample) [4]. Other than generating a counterexample or proving the assertion, an SBMC tool may also run into a timeout, or indicate that the specified bounds may need to be increased for the assertion to be proven. Hence, one can simply increase the specified bound until the assertion is fully proven. Additionally, SBMC analyses the program beforehand, and—if no bound is specified by the user—infers a sufficiently large bound if the program is simple enough, as it is the case for the experiments within this paper. The graph resulting from symbolic execution is transformed into a formula in a decidable logic (in our case propositional) that is satisfiable if and only if a counterexample exists, reducing the verification problem to a decidable satisfiability problem. Then, modern SAT/SMT-solving technology is used to check whether such a counterexample exists. Furthermore, SBMC tools support features of common complex programming languages such as complex memory models or standard data types in order to check a wider range of correctness properties, e.g., correct memory allocation.

In contrast to more heavy-weight verification techniques, SBMC does not aim to establish universal correctness guarantees or full reliability for all possible input parameters. It is usually being used to find general low-level bugs in programs, such as memory access errors or other sources of non-deterministic behaviour. Nevertheless, SBMC can also be used to check more complex functional properties – as we do for the purposes of this paper. SBMC considers only a finite state space by cutting off program execution paths at a certain length. Thus, it 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 exhaustive testing by far.

Within this work, we use the model checker CBMC [7], which takes C/C++ or Java programs as input. The programs are annotated with specifications in the form of assumptions and assertions. Since universal and existential quantifiers are not supported by CBMC using the SAT back end, quantified expressions need to be expressed as assumptions/assertions within a loop. CBMC internally models all data structures as bit vectors. The symbolically executed programs are translated into equations over bit vectors, which are then processed by a powerful SAT solver modulo theories.

For our experiments, we use CBMC 5.3 with the built-in solver based on the SAT solver MiniSat 2.2.0 [9]. All experiments are performed on an Intel(R) Core(TM) i5-3360M CPU at 2.80 GHz with 4 cores and 16 GB of RAM.

4 Automated Margin Computation Using SBMC

We assume that an election function is given as an imperative program (a C function called `election_function` in our case) as well as a concrete input (denoted as `vote_table`) for that election function. The `vote_table` is the result of vote counting and tabulation. We model `vote_table` as an integer array of size `PARTIES`, where `PARTIES` is the number of different stacks into which identical votes are accumulated during counting.

The idea of our approach is to use an SBMC tool to check an assertion claiming that, when `vote_table` is changed by putting at most a certain number m of votes on other stacks than they were on, the outcome of the election is *not* changed. If that assertion is provable, we know that the actual election margin is greater than m. If the assertion is not provable, we know that the actual election margin is less than or equal to m. In the latter case, the SBMC tool generates a counterexample to the assertion demonstrating that the election outcome can be changed by changing m votes. Having this proof obligation as a basis, we can use binary search to find a value for m such that the assertion holds for $m - 1$ but fails for m, i.e., m is exactly the election margin.

The check for a particular prospective margin m can be executed by running the SBMC tool CBMC on the program shown in Listing 1, where the variables written in capital letters are given as concrete input values, and the method `nondet_int()` is a CBMC feature in order to denote non-deterministic, i.e., potentially different for each function call, and symbolic, i.e., unknown, integer values.

The changes in the sizes of the vote stacks are non-deterministically chosen (Line 4) in such a way that the total difference is zero (assumption in Line 15), i.e., votes can be moved from one stack to the other but not removed or created, and such that the number of votes in each stack cannot become negative (Line 6). Other types of margins for other kinds of changes to the vote table can be computed using different assumptions on the chosen values for `diff`.

```
 1 void verify() {
 2     int new_votes[PARTIES], diff[PARTIES], total_diff, pos_diff;
 3     for (int i = 0; i < PARTIES; i++) {
 4         diff[i] = nondet_int();
 5         __CPROVER_assume (-1 * MARGIN ≤ diff[i] ≤ MARGIN);
 6         __CPROVER_assume (0 ≤ ORIG_VOTES[i] + diff[i]);
 7     }
 8
 9     for (int i = 0, total_diff = 0, pos_diff = 0; i < PARTIES; i++) {
10         new_votes[i] = ORIG_VOTES[i] + diff[i];
11         if (0 < diff[i]) pos_diff += diff[i];
12         total_diff += diff[i];
13     }
14     __CPROVER_assume (pos_diff ≤ MARGIN);
15     __CPROVER_assume (total_diff == 0);
16
17     int *result = election_function(new_votes);
18     assert (equals(result, ORIG_RESULT));
19 }
```

Listing 1. Implementation of the margin computation for CBMC.

The changes are added to the original vote table for computing the new table (Line 10). And the election result for the new vote table is computed by calling the method `election_function` (Line 17).

Finally, the program contains the assertion to be checked by CBMC (Line 18), expressing that the new election result is equal to the original one. Intuitively, we have encoded any difference between the original election outcome and the new one as a bug to be found by the model checker. This also means that our approach gives us a concrete redistribution of votes for the computed margin, as CBMC encodes detected bugs as concrete paths through the program, which lead to the assertion violation, i.e., the changed outcome.

The algorithm performing a binary search for the exact election margin is shown in Table 1 (for our experiments we use a shell script implementation of this algorithm). The algorithm takes as input the implementation of an election function and a concrete vote table. Its output is the exact election margin.

The algorithm first calls `election_function` to obtain the original election result (Line 3). The left and right bounds of the binary search are initialised to zero resp. the total number of votes (Lines 5 to 6). Then, a while loop (Lines 9 to 17) performs the binary search and calls CBMC on the program from Listing 1 with different values for `MARGIN`, i.e., different candidate margins, until the solution is found. If the result of CBMC indicates that `MARGIN` is too low, the left bound is increased (Line 13), and if CBMC indicates that `MARGIN` is either the correct margin or is too high, then the right bound is decreased (Line 15). To be more precise, if the result of calling CBMC reads `SUCCESS`, we know that the assertion in the program in Listing 1 holds, i.e., the election outcome cannot be

Table 1. Binary search for election margin using SBMC.

Input:
    `election_function`: implementation of the election function
    `ORIG_VOTES`: array with the size of each of the stacks of identical votes, i.e., the input for the election function
    `PARTIES`: size of the vote table array
Output:
    `MARGIN`: computed election margin

```
function searchMargin
    // initialisation
    ORIG_RESULT ← election_function(ORIG_VOTES)
    MARGIN ← 0
    $left \leftarrow 0$
    $right \leftarrow \sum_{i=1,\ldots,\mathtt{PARTIES}}$ ORIG_VOTES[$i$] // total number of votes

    // search for margin
    while $left < right$ do
        MARGIN ← $left + \left[\frac{right-left}{2}\right]$
        $result$ ← cbmc(verify(), MARGIN, PARTIES, ORIG_VOTES, ORIG_RESULT)
        if $result$ = SUCCESS then
            $left$ ← MARGIN + 1, MARGIN ← MARGIN + 1
        else
            $right$ ← MARGIN
        end if
    end while

    return MARGIN
end function
```

affected and the speculative margin `MARGIN` is too low; otherwise `MARGIN` either is the correct election margin or it is too high.

Note that neither the algorithm in Table 1 nor the program in Listing 1 make any further assumptions regarding the election function. Our method can be applied to arbitrary implementations of `election_function` without making any changes, only influencing the computation time needed by the satisfiability solver used as a back end, e.g., for more complex mathematical operations. The approach can also be adapted to more complex ballot structures. And, as said above, margins for different notions of vote changes can be computed by using different assumptions on the array `diff` in Listing 1, and margins for different notions of changes in the election outcome can be computed by using different versions of the function `equal` called in Line 18 from Listing 1.

5 Margin Computation for the D'Hondt Method

Margin computation also plays a central role for risk-limiting audits regarding the results after performing seat allocation methods such as the D'Hondt or Saint-Laguë method [19]. In this section, we exemplarily apply our technique to the D'Hondt method, which allocates mandates to a number of parties based on the votes cast for these parties. Before the D'Hondt election function is applied, vote counting and tabulation sorts the votes into stacks where each stack contains votes for a single party. The input for the election function then is the number of votes for each party (i.e., the number of votes in the corresponding stack).

The D'Hondt method proportionally allocates mandates to parties in such a way that the number of votes represented by mandates is maximised, i.e., the votes-per-seats ratio—intuitively the price in number of votes to be paid by a party to get one seat—is made as high as possible while still allocating all seats in parliament. By this means, D'Hondt achieves an—as far as possible—proportional representation in parliament [11].

D'Hondt can be implemented as a *highest averages* method: the number of votes for each party is divided successively by a series of divisors, which produces a table of quotients (or averages). In that table, there is a row for each divisor and a column for each party. For the D'Hondt method, these divisors are the natural numbers $1, 2, \ldots,$ `MANDATES`, where `MANDATES` is the total number of mandates to be distributed. Then, the highest numbers in the quotient table—resp. the parties in whose columns these numbers are—are each allocated one seat. The "final" seat goes to the `MANDATES`'th highest number. Hence, the threshold level of the votes-per-seats-ratio lies in the interval between the `MANDATES`'th highest number and the (`MANDATES` + 1)'st highest number of all computed averages in the quotient table.

An efficient C implementation of D'Hondt is shown in Listing 2. There, the constants `PARTIES` and `MANDATES` encode the numbers of parties and the number of mandates to be allocated, respectively. The input is given in the array `vote_table`, which holds the numbers of votes cast for each individual party. This implementation avoids constructing the complete quotient table. Instead, it stops as soon as the `MANDATES`'th highest quotient has been found. For this purpose, the divisors currently under consideration for finding the next highest value are stored in the array `divisor` for each party. Note that in case of a tie, the order in `vote_table` is the tie-breaker, i.e., the first party in `vote_table` which is tied with the current maximum divisor takes the seat.

After initialising the arrays `mandates` and `divisor` (Lines 5 and 6), we execute the outer loop (Lines 9 to 15) `MANDATES` times. Each time, it uses the inner loop (Lines 10 to 12) to find the maximum

$$\texttt{elected} = \max_{i=1,\ldots\texttt{PARTIES}} \frac{\texttt{vote\_table}[i]}{\texttt{divisor}[i]}$$

and then assigns one seat to the `elected`'th party (Line 13), and increases the divisor for that party (Line 14). To find the maximum, the comparison

```
int *election_function(int vote_table[PARTIES]) {
    int *mandates = malloc(PARTIES * sizeof(int));
    int divisor[PARTIES];

    for (int i = 0; i < PARTIES; i++) mandates[i] = 0;
    for (int i = 0; i < PARTIES; i++) divisor[i] = 1;

    int elected = 0;
    for (int j = 0, j < MANDATES; j++) {
        for (int i = 0; i < PARTIES; i++)
            if (divisor[i] * vote_table[elected]
                    < divisor[elected] * vote_table[i]) elected = i;
        mandates[elected]++;
        divisor[elected]++;
    }
    return mandates;
}
```

Listing 2. Implementation of the D'Hondt method as a C program.

$^{\texttt{vote\_table[elected]}}/_{\texttt{divisor[elected]}} < ^{\texttt{vote\_table[i]}}/_{\texttt{divisor[i]}}$ is replaced by `divisor[i]` $*$ `vote_table[elected]` $<$ `divisor[elected]` $*$ `vote_table[i]`, which is equivalent as the divisors are positive numbers. The advantage of using the latter form for the comparison is to avoid dealing with fractional numbers and rounding effects in C. This is a sensible choice for any implementation of D'Hondt as, depending on the programming language and hardware, rounding may both show unexpected behaviour and potentially lead to faulty election results.

In order to test our margin computation for D'Hondt, we used the preliminary official results of the Schleswig-Holstein state elections in 2005[1]. In that election, $1,367,095$ votes were cast and 69 mandates were to be allocated. Out of the 13 parties running, four parties received the necessary quota of 5% to be eligible for the mandate allocation. The fifth party to receive seats, the South Schleswig Voter Federation, represents the Danish minority and is exempted from the quota rule for reasons of minority protection. The mandates (seats in parliament) were allocated using the D'Hondt method. The parties, their votes, and the allocated mandates are shown in Table 2.

We applied our approach to the vote numbers (i.e., the `vote_table`) of the Schleswig-Holstein election for various values of `MANDATES`. In doing so, we were able to compute the margin of the election with the runtime increasing for higher values of `MANDATES` as shown in Fig. 1a and b. The runtime for the final check is shown in Fig. 1a. This check requires showing that the election result can be changed by changing m votes (counterexample generation) but cannot be changed by changing $m - 1$ votes (margin verification), implying that m is the

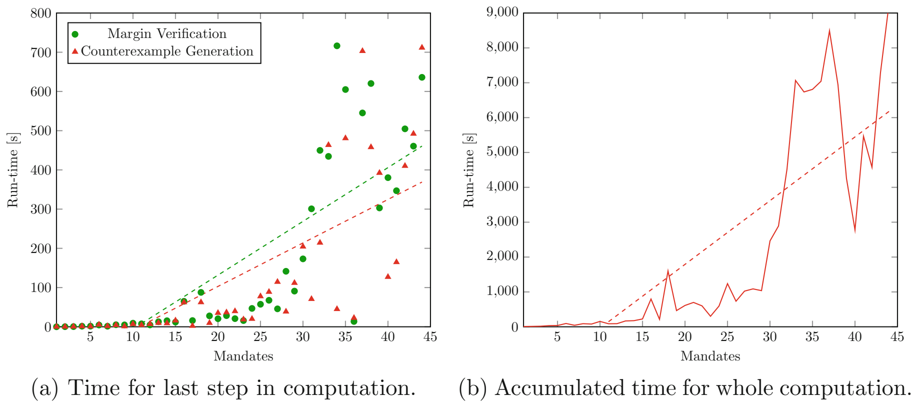
[1] The results of that election are also used as an example in the German Wikipedia article on the D'Hondt method (http://de.wikipedia.org/wiki/D'Hondt-Verfahren).

Table 2. Preliminary official results for the 2005 Schleswig-Holstein elections.

Party	Votes	%	Mandates	%
Christian Democratic Union (CDU)	576 100	42.1	30	43.4
Social Democratic Party (SPD)	554 844	40.6	29	42.0
Free Democratic Party (FDP)	94 920	6.9	4	5.8
Alliance ’90/The Greens	89 330	6.5	4	5.8
South Schleswig Voter Federation (SSW)	51 901	3.7	2	2.9
Totals	1 367 095		69	

true margin. Figure 1a shows the accumulated time for the complete binary search that computes m. For values of `MANDATES` between 2 and 45, the computed margins range from only 433 (for `MANDATES` $= 23$) to $177,863$ (for `MANDATES` $= 2$). Note that, with only two mandates, the CDU and the SPD each get a seat; the margin of $177,863$ then is the number of votes that have to be moved from the SPD to the CDU so that the CDU gets both mandates instead of only one, which is smaller than the number of votes that would have to be moved from the SPD to the FDP so that the FDP gets a seat instead of the SPD.

The runtimes shown in the figure do not form a smooth curve because they depend on the margin that is computed, which is, e.g., smaller for 40 mandates than for 35. But the numbers increase with the value of `MANDATES`. And as can be seen from the figure, they get prohibitively large for more than about 45 mandates.



(a) Time for last step in computation. (b) Accumulated time for whole computation.

Fig. 1. Runtimes of automatic margin computation for the D’Hondt method with various values for `MANDATES`.

Thus, our approach can be applied to real implementations of real election functions, but only if the number of loop iterations does not go beyond a few

```
int *election_function(int votes[PARTIES]) {
    int *mandates = malloc(PARTIES*sizeof(int));
    for (int i = 0; i < PARTIES; i++) mandates[i] = 0;

    int quotaNumerator   = nondet_int();
    int quotaDenominator = nondet_int();

    __CPROVER_assume (0 < quotaNumerator   ≤ INT_MAX);
    __CPROVER_assume (0 < quotaDenominator ≤ MANDATES);
    __CPROVER_assume (quotaDenominator < quotaNumerator);

    for (int i = 0; i < PARTIES; i++) {
        __CPROVER_assume (0 ≤ quotaDenominator * votes[i] ≤ INT_MAX);
        mandates[i] = ((quotaDenominator * votes[i]) / quotaNumerator);
        __CPROVER_assume (0 ≤ mandates[i] ≤ MANDATES);
    }

    int total_mand = 0;
    for (int i = 0, total_mand = 0; i < PARTIES; i++)
        total_mand += mandates[i];
    __CPROVER_assume (total_mand == MANDATES);

    return mandates;
}
```

Listing 3. Implementation of the Jefferson method as a symbolic C program.

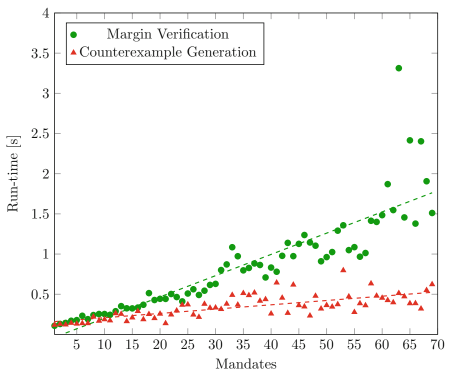
hundred (about 5 parties times 45 mandates in this case). For elections with a larger number of parties and mandates or election functions with more complex loop nestings, improvements are required. One such improvement is discussed in the following section.

6 Using SBMC to Find Parameters in Election Function

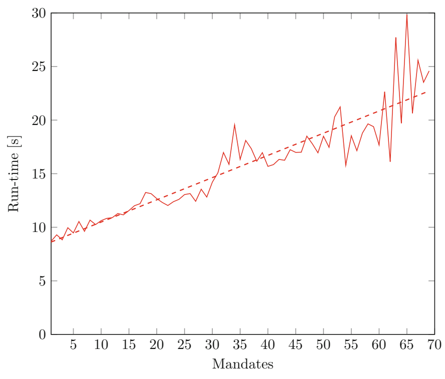
The election function defined by the D'Hondt method can also, equivalently, be described without a quotient table. Instead, a quota is chosen, i.e., a number of votes needed to "buy" one mandate, such that the resulting mandates per party, when rounded down to the next natural number, sum up to the required total number of mandates. This is known as Jefferson's method and is similar to *largest-remainder* methods such as the Hare-Niemeyer method. The quota corresponds to the lowest quotient in the D'Hondt table for which a mandate is allocated.

If the implementation of an election function is based on choosing or searching for some parameter (here the quota), then the margin computation can be made much more efficient by replacing the search for the parameter by a non-deterministic choice to be resolved by the SBMC tool.

An implementation of the Jefferson method in C is shown in Listing 3. It uses a non-deterministic choice of `quota` = `quotaNumerator`/`quotaDenominator` (Lines 5 to 6). Assumptions are made to limit the range of the quota (Lines 8 to 10 and Line 13). The number of mandates for each party is computed (Line 14), as well as the total number of mandates (Lines 18 to 20). Then, the assumption is checked that the total number of mandates for the chosen quota is the correct one (Line 21). This final check is an assumption and not an assertion, i.e., we want to consider only the case(s) where the total number of mandates is correct; other cases are irrelevant. An assertion, on the other hand, would have to be true for all cases where the (other) assumptions are fulfilled. Note that this implementation does not deal with tie-breaking, as in this case no such quota can be found, and no program execution path can satisfy the assumption in Line 21. However, tie-breaking mechanisms can easily be integrated in the program.



(a) Time for last step in computation.



(b) Accumulated time for whole computation.

Fig. 2. Runtimes of automatic margin computation for the Jefferson method with various values for `MANDATES`.

The runtimes of the automatic margin computation for the 2005 Schleswig-Holstein state elections with various values for `MANDATES`, i.e., the total number mandates to be allocated, are shown in Fig. 2a and b. Note that these runtimes are much lower than those for the D'Hondt method in Fig. 1a and b. Now, all computations stay well below the time-out of 9,000 s (i.e., 2.5 h), even below 30 s. And the computation of the election margin for the original number of mandates in the election, which is 69, is now easily possible; that margin is 634. The computed margins range from only 42 (for `MANDATES` = 62) to 177,863 (for `MANDATES` = 2). Performing our method for various values for `MANDATES` scales well on the Jefferson method, as we got rid of the loop depending on the value of `MANDATES`. However, further experiments also indicated a non-exponential dependency on the value for `PARTIES`. For example, an allocation of 69 mandates to 10 parties takes about 55 s, whereas for 20 parties, the analysis runs in ca. 300 s.

Naturally, the implementation in Listing 3 cannot be compiled and executed to produce a binary file using standard C compilers, because it contains constructs only understood by the model checker CBMC. However, it can nevertheless be compiled and executed using CBMC, which also allows for performing tests and similar measures in order to generate confidence in the implementation. Furthermore, when any C implementation of the Jefferson method is given, it is easy to construct a CBMC version in a uniform way by replacing the search for `quota` by a non-deterministic choice. The same principle for making margin computations more efficient can uniformly be applied to any election function where parameters such as quotas are chosen or computed within the election function.

Table 3. Official results for the 2015 national Danish elections [8].

Party	Votes	%	Mandates	%
Socialdemokratiet	924 940	26.3	43	31.9
Radikale Venstre	161 009	4.6	2	1.5
Det Konservative Folkeparti	118 003	3.4	0	0.0
SF – Socialistisk Folkeparti	147 578	4.2	2	1.5
Liberal Alliance	265 129	7.5	9	6.7
Kristendemokraterne	29 077	0.8	0	0.0
Dansk Folkeparti	741 746	21.1	33	24.4
Venstre, Danmarks Liberale Parti	685 188	19.5	33	24.4
Enhedslisten – De Rød-Grønne	274 463	7.8	10	7.4
Alternativet	168 788	4.8	3	2.2
Totals[a]	3 515 921		135	

[a]Excluding non-party votes.

7 Computing the Margin for National Danish Elections

In this section, we demonstrate the applicability of our approach to a further, more complex real-world election, namely the Danish parliamentary elections in 2015. The Danish elections use a two-tier system, further classified as an *adjustment-seat system*, where the main part of the seats (135 mandates) is allocated using the D'Hondt method for each of the lower-tier electoral districts (so-called constituencies) separately [10]. The remaining seats (40 mandates) are used for adjusting the proportionality with respect to the three higher-tier districts using the Saint-Laguë method (which is also a *highest averages method*, bounded by the Hare quota).

The aggregated results for the 2015 election are shown in Table 3. For the sake of readability, the table only contains the total numbers of votes, not the numbers for each constituency. In the following, we perform our analysis on the first tier, i.e., the distribution of the 135 mandates which are allocated separately within each constituency.

Using the Jefferson-version of D'Hondt, we compute a margin of 10 votes within 7,815 s, i.e., around 2 h and 10 min. The final verification (proving that a change in 9 votes cannot change the election outcome) takes 53 s and a counterexample for 10 votes (i.e., an example ballot box that does change the election outcome) can be found within 27 s. The generated counterexample shows that shifting – only – 10 votes from *SF – Socialistisk Folkeparti* to *Venstre, Danmarks Liberale Parti* in the constituency of Sjællands Storkreds results in a different election outcome where one mandate goes the same way as the 10 votes. That is, SF loses its single seat, and Venstre then has five seats. The vote table and election results for the constituency of Sjællands Storkreds are shown in Table 4.

Table 4. Results for the Danish constituency Sjællands Storkreds [8].

Party	Votes	%	Mandates	%
Socialdemokratiet	146 464	27.9	7	35.0
Radikale Venstre	16 906	3.2	0	0.0
Det Konservative Folkeparti	15 083	2.9	0	0.0
SF - Socialistisk Folkeparti	20 575	3.9	1	5.0
Liberal Alliance	32 598	6.2	1	5.0
Kristendemokraterne	1 996	0.4	0	0.0
Dansk Folkeparti	134 195	25.6	6	30.0
Venstre, Danmarks Liberale Parti	102 818	19.6	4	20.0
Enhedslisten - De Rød-Grønne	35 374	6.7	1	5.0
Alternativet	18 202	3.5	0	0.0
Totals[a]	524 211		20	

[a]Excluding non-party votes.

With the table-based D'Hondt method as a basis (Listing 2), the margin computation takes 16,860 s (around 4 h and 40 min). The final verification takes 659 s and a counterexample can be found within 652 s. Using the table-based D'Hondt implementation, for which margin computation is less efficient, is possible in this case because the number of mandates for each constituency is sufficiently low (around 20).

8 Conclusion and Future Work

In this paper, we have presented a method that computes election margins fully automatically. It can be applied to arbitrary implementations of election functions without understanding or even knowing how the election result is computed. Our approach can be applied to real implementations of real election functions if the number of loop iterations in the election function does not go beyond a few hundred. With the improvement from Sect. 6 for guessing parameters needed in the computation, the method scales up to larger and more complex elections.

Future work includes the computation of different types of election margins and an integration with software for supporting real-world risk-limiting audits. Further, we plan to apply our method to election functions for which margin computation is notoriously hard (such as instant-runoff voting). First experiments indicate that such functions are hard for our method as well. But it will be possible to adapt our method to computing lower bounds for margins in IRV elections using techniques described in the literature [6,16].

Acknowledgements. This work has been partly supported by COST Action IC1205 on Computational Social Choice. This publication was made possible in part by the DemTech grant 10-092309 from the Danish Council for Strategic Research, Program Commission on Strategic Growth Technologies and in part by NPRP Grant #7-988-1-178 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

1. Andoni, A., Daniliuc, D., Khurshid, S.: Evaluating the “small scope hypothesis”. Technical report, MIT Laboratory for Computer Science, Cambridge, MA (2003)
2. Bartholdi, J.J., Orlin, J.: Single transferable vote resists strategic voting. Soc. Choice Welf. **8**, 341–354 (1991)
3. Beckert, B., Goré, R., Schürmann, C., Bormer, T., Wang, J.: Verifying voting schemes. J. Inf. Secur. Appl. **19**(2), 115–129 (2014)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). doi:10.1007/3-540-49059-0_14
5. Blom, M.L., Stuckey, P.J., Teague, V., Tidhar, R.: Efficient computation of exact IRV margins. Computing Research Repository (CoRR) abs/1508.04885 (2015)
6. Cary, D.: Estimating the margin of victory for instant-runoff voting. In: Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/-WOTE). USENIX Association (2011)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
8. Statistik, D.: Befolkning og valg (2015). http://www.dst.dk/valg/Valg1487635/other/2015-Folketingsvalg.pdf. Accessed 23 August 2016
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: International Conference on Theory and Applications of Satisfiability Testing (SAT), Selected Revised Papers, pp. 502–518 (2003)
10. Elklit, J., Pade, A.B., Nyholm Miller, N.: The parliamentary electoral system in Denmark (2011). http://www.ft.dk/Dokumenter/Publikationer/Engelsk/The_Parliamentary_Electorial_System_Denmark.aspx. Accessed 23 August 2016
11. Gallagher, M.: Proportionality, disproportionality and electoral systems. Elect. Stud. **10**(1), 33–51 (1991)

12. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: systematic test case generation for dynamic analysis and measurement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_20
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
14. Lindeman, M., Stark, P.B.: A gentle introduction to risk-limiting audits. IEEE Secur. Priv. **10**(5), 42–49 (2012)
15. Magrino, T.R., Rivest, R.L., Shen, E., Wagner, D.: Computing the margin of victory in IRV elections. In: Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE). USENIX Association (2011)
16. Sarwate, A., Checkoway, S., Shacham, H.: Risk-limiting audits and the margin of victory in nonplurality elections. Stat. Polit. Policy **4**(1), 29–64 (2013)
17. Smith, A.M., Butler, E., Popovic, Z.: Quantifying over play: constraining undesirable solutions in puzzle design. In: International Conference on the Foundations of Digital Games (FDG), pp. 221–228 (2013)
18. Stark, P.B.: Super-simple simultaneous single-ballot risk-limiting audits. In: Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE), pp. 1–16 (2010)
19. Stark, P.B., Teague, V.: Verifiable european elections: risk-limiting audits for D'Hondt and its relatives. USENIX J. Elect. Technol. Syst. (JETS) **1**, 18–39 (2014)
20. Vorobyov, K., Krishnan, P.: Combining static analysis and constraint solving for automatic test case generation. In: Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 915–920 (2012)