Testing with Functional Reference Implementations

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
{pieter, rinus}@cs.ru.nl

Abstract. This paper discusses our approach to test programs that determine which candidates are elected in the Scottish Single Transferable Vote (STV) elections. Due to the lack of properties suited for model-based testing, we have implemented a reference implementation in a pure functional programming language. Our tests revealed issues in the law regulating these elections as well as the programs implementing the rules that are offered for certification. Hence, certification by testing with a reference implementation is able to reveal problems in the software to be certified. Functional programming languages appeared to be an excellent tool to implement reference implementations. The reference implementation was developed quickly and none of the differences found was due to an error in the reference implementation.

1 Introduction

In traditional testing techniques the test cases are designed by a test engineer. The test engineer specifies some inputs and the expected output of the implementation under test (iut). These test cases are executed by hand, or by a special purpose automatic test tool. The automatic execution of tests has the advantage that it is faster and more reliable. This is especially convenient for repeated testing (called regression testing) after changes in the iut.

Model-based testing is a powerful technique to increase the confidence in the quality of software. In model-based testing the test engineer specifies a general property rather than concrete input-output pairs. Usually the inputs for the test can be derived automatically. Even if this is not possible, the model-based approach has the advantage that the specified property makes it clearer what is tested, it is easy to execute more tests, and it is much easier to adapt the tests if changed system properties require this.

In order to use model-based testing it is crucial to have a property relating input and output in a sufficiently powerful way, or a set of these properties. Usually such properties can be deduced from the specification of the program, even if the specification is informal. In this paper we show how model-based testing can be used if it is not possible to formulate sufficiently powerful properties relating input and output. We construct a reference implementation, ri, for the iut and require that this ri produces the same results as the iut for all

inputs. This is expressed by the property: $\forall\, i \in \mathsf{Input}\,.\,\mathsf{iut}\ i = \mathsf{ri}\ i$. This property is tested by evaluating it for a large number of inputs i. Preferably these inputs are generated automatically. Even if the inputs are generated manually instead of automatically, this approach is beneficial when we want to execute a large number of tests since we have to create and maintain only one ri instead of a large number of handcrafted input-output pairs. Obviously this approach only works if we can create a reference implementation for reasonable costs.

The main requirements for ri are clearness and a low cost and fast development. Execution speed is less important, a slow ri will only slowdown the test execution, but not the iut. Also a nice user interface is not required, in the automatic tests we compute iut i = ri i by a program rather than manually. Maintainability is not required for single certification of the iut, but becomes important if we want to perform regression tests on the iut. Given these requirements for the ri, functional programming languages are the ideal tool to construct the ri. Given that the iut is usually not developed in a functional programming language and that the ri is developed completely independently of the iut, it is very unlikely that both implementations contain the same mistakes and hence pass these tests unnoticed. Hence, correctness of the ri is desirable, but not absolutely required.

In this paper we illustrate this approach of software testing by a real world example. We were asked to certify two different election programs to be used in Scottish local elections by testing. This software has to implement a specific version of a Single Transferable Vote (STV) system [9, 10].

The trend we indicate in this paper is the use of functional programs as reference implementation during the testing phase of programs written in mainstream languages like Java. This paper presents an example that shows that this can work very well.

In our tests we focus on input-output of the systems under test, this is called functional testing in standard test terminology. Since we are not able to monitor what happens inside the implementation under test our test are usually called black-box tests: the iut is treated as a black-box. Since our tests cover the entire system this level of testing is called system testing (in contrast to for example unit testing that concentrates on testing individual functions or methods). Our tests are based on a general property or model of the system, hence this way of testing is called model-based testing. Model-based testing is significantly more advanced than automated testing. In automated testing a user defined test suite, set of tests, is executed automatically by some program. In model-based testing the test tool generates a test suite, usually on-the-fly, executes these tests and gives a verdict based on the test results.

In Section 2 we explain the STV election rules in some more detail. We explain how the iut was tested in Section 3. Some details about the implementations tested are given in Section 4 and details about the reference implementation in Section 5. The issues found during testing are categorized in Section 6 and discussed in Section 7. Finally we draw conclusions in Section 8.

2 STV Election Rules

STV is a preferential voting system designed to minimize wasted votes and to provide proportional representation while votes are explicitly for candidates and not for party lists. When STV is used in multi-seat constituencies as in this case, it is also called proportional representation through the single transferable vote (PR-STV). STV usually refers to PR-STV as it does here. In Australia STV is known as the Hare-Clark Proportional method, while in the United States it is sometimes called choice voting or preference voting.

The key idea is that each voter gives a preference list of candidates as a vote. This list contains at least one candidate and at most all candidates in order of preference. When a candidate has more votes than needed to be elected or is eliminated in the voting process, the additional votes are transferred (partly) to the next candidate on the ballots. The orders of vote transfer and elimination are important. The exact rules to be followed are very operationally specified in the law for these elections. The law states how a human being should determine the result of the election by sorting ballots and transferring ballots from one pile to another pile with a specific weight.

For a real election there are a large number of ballots and the STV system often needs a large number of stages to decide which candidates are elected. We have seen up to 100 stages during the tests. To compute the election results fast and accurately it is necessary to use a computer program that implements the law and determines which candidates are elected based on the ballots. Obviously such a program should be correct. Since there is no way to check the election results for voters and candidates, one has to trust such a program. In order to improve the confidence in this software we were asked to certify it by performing black box tests for this system. For the certification of one iut we had to verify the results of a test suite of over 800 elections, for the other iut no test suite was specified.

As authors of the model-based test tool G∀st [5, 6] we initially planned to state a sufficiently strong set of logical properties and test the election software with G∀st. G∀st is a model-based test system that automatically tries to find behavior of the iut that is not allowed by the specification. A specification is either a logical property, or an extended state machine. It appeared however impossible to derive sufficiently strong logical properties relating the election results to the input of the program (the ballots). For instance, one odd aspect of STV is that it is non-monotonic. This means that getting more votes may prevent a candidate from being elected in some rare situations [1, 4]. Also a candidate that occurs on the second position of each and every ballot is not necessarily elected [2]. Since the voting software only receives one input, the set of ballots, a state-based approach does not work either.

Instead of trying to derive logical properties from the law specifying this version of the STV elections we decided to construct a ri that implements the law and check for all test cases if it yields the same result as the iut. We used the functional programming language Clean [8] to implement this ri. In this example Input is the set of all possible elections E. This set is infinite. Even if we pose

upper bounds on the number of candidates and ballots, the set of elections with realistic upper bounds is way too large to enable exhaustive testing. In order to ensure that differences between the iut and ri are found during testing our first customer supplied a set S of about 800 elections to be used as test suite. S contains hand crafted normal elections, borderline cases for the STV-rules, as well as real elections from STV-elections[1]used world wide, see e.g. [11]. We also generated test suites and investigated their capability to find errors. Our experiments show that generated test suites are able to spot the same issues as the test suite S.

2.1 Specification of the Scottisch STV

In the first step of the election process the votes are given to the first candidate in the ballots. The quota Q is computed as:

$$Q = \lfloor \frac{\text{number\_of\_votes}}{\text{number\_of\_seats} + 1} \rfloor + 1$$

The floor brackets $\lfloor x \rfloor$ indicate rounding down of the number x. The fraction of x, if any, is simply removed. This definition is called the *Droop quota* [3], it ensures that the number of candidates that reach the quota is at most equal to the number of seats. Each candidate that reaches the quota is elected. When a candidate is elected and has more votes than required, the surplus of votes is transferred proportionally to the next candidates on the ballots assigned to that candidate. Similarly, if a candidate is eliminated, their votes transfer to the next candidates on the ballots of that candidate.

The election rules [10], give an operational description that guides a human to determine the result of the election by putting ballots on piles and transferring ballots to other piles. In more abstract terms this algorithm is:

```
assign_ballots_to_the_first_candidate;
while (not all_seats_are_filled)
{
    declare_any_candidate_having_Q_votes_or_more_elected;
    if (number_of_candidates = number_of_seats)
        elect_all_remaining_candidates;
    else if (there_are_candidates_with_untransfered_surplus_of_votes)
        transfer_votes_of_candidate_with_the_most_votes;
    else
        eliminate_candidate_with_fewest_votes_and_transfer_votes;
}
```

In the situation that there is more than one candidate with the highest number of votes, we look back in the history. If one of the candidates had more votes on one of the previous iterations of the algorithm, the surplus of that candidate is transferred first. Otherwise there is a *tie*. The law prescribes that a candidate

[1] There are many variants of the STV rules used world wide. Although different rules might yield different results, these elections still provide realistic test cases.

is chosen by lot in these situations. In a real election a human has to decide which candidate is treated first. During testing various fixed orders of elimination are used in order to speed up testing and to obtain reproducible results. If there are several candidates with the least number of votes for elimination the same algorithm is used: look for a difference in the history, if that is not available it is a tie.

Initially all votes have value one. In the transfer of votes they get a new value. The transfer value, tv is computed as

$$tv = \frac{(votes\_of\_candidate - Q) \times current\_value}{votes\_of\_candidate}$$

The tv is truncated to five decimal places.

The votes are transferred per ballot pile to the next candidate on that ballot that is neither elected nor eliminated. If there is not such a candidate available, the votes are marked as nontransferable votes. Also fractions of votes that are lost by truncation to five decimal places are added to the nontransferable votes. The nontransferable votes are only recorded to monitor that no votes are lost.

2.2 Format of the Test Cases

Each test case is stored in a separate text file. Some typical examples are listed in the tables 1 and 3. The first line of the file contains the number of candidates and the number of seats. Then there is optionally a line indicating which candidates are withdrawn, indicated by a sequence of negative numbers. Withdrawals are not possible in the Scottish elections, the data format contains them since they occur in some other STV elections used as test case. Then there is a series of lines indicating the values on the ballot papers. This sequence is terminated by a line beginning with 0. Each line starts with the number of ballot papers with this vote distribution, followed by the numbers of the candidates and terminated with 0.

After the votes, there are some lines containing the names of the candidates between quotes and the name of the election. Optionally this data is followed by some comments.

2.3 Example Election

A very small example election is the selection of 3 out of 5 candidates with 402 votes is shown in Table 1. On the left we show the actual data, on the right an explanation. This example is designed such that fractions do no occur, all candidates will have a natural number of votes during the entire election process.

Table 2 contains the transfer table of votes produced by election 1. The row labeled void contains the nontransferable votes.

Since there are 402 votes (all valid) and 3 seats, the quota Q equals $\frac{402}{3+1} + 1 = 101$. This implies that Alice and Bob are elected immediately. Since Alice has

5 3	5 candidates, 3 positions
200 1 2 4 0	200 ballots with preference 1 2 4
125 2 5 0	125 ballots with preference 2 5
1 4 0	1 ballot with only candidate 4
76 5 0	76 ballots with only candidate 5
0	end of ballots
"Alice"	name of candidate 1
"Bob"	name of candidate 2
"Carol"	...
"Dave"	
"Ed"	
"Example election 1"	Name of this election

Table 1. The input data for example election 1 and an explanation of this input.

name	initial	trans 1	votes	trans 2	votes	elim 3	votes	elim 5	votes	final
Alice	200.0	-99.0	101.0	-	101.0	-	101.0	-	101.0	Elected
Bob	125.0	-	125.0	-24.0	101.0	-	101.0	-	101.0	Elected
Carol	-	-	-	-	-	-	-	-	-	
Dave	1.0	99.0	100.0	-	100.0	-	100.0	-	100.0	Elected
Ed	76.0	-	76.0	24.0	100.0	-	100.0	-100.0	-	
void	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	100.0	
total	402.0		402.0		402.0		402.0		402.0	

Table 2. The transfer table of example election 1

more votes than Bob her votes are transferred first (column trans 1). After each transfer there is a column indicating the new number of votes for all candidates. Since Bob is already elected, he does not receive votes from Alice. In the next iteration the votes of Bob are transferred (trans 2).

Now there are no votes to transfer and we have to eliminate a candidate. Since Carol has less votes than Dave and Ed, she is eliminated first (elim 3). This empty column indicating the transfer of zero votes. The absence of votes for a candidate is indicated with a $-$.

Next Dave or Ed has to be eliminated since none of the remaining candidates reaches the quota. In the current round they have an equal amount of votes, but in a previous round Dave had more votes than Ed. So, Ed is eliminated. There is no next candidate on the ballots of Ed, this implies that the votes are non-transferable. Note that we start at the last vote distribution and look back in history. If we would start at the first round, Dave is eliminated instead of Ed, since he had less votes there.

The remaining candidate, Dave, is now deemed to be elected since the number of remaining candidates and the number of seats to be filled are equal. Note that Dave never reached the quota. This ends the election algorithm.

4 2
9 1 0
5 2 3 0
6 3 0
4 4 2 0
2 *4* *3* 0
0
Alice
Bob
Carol
Dave
example 2

name	initial	elim 2	votes	final
Alice	9.0	-	9.0	Elected
Bob	5.0	-5.0	-	
Carol	6.0	5.0	11.0	Elected
Dave	6.0	-	6.0	
void	0.0	0.0	0.0	
total	26.0	26.0	26.0	

4 2
9 1 0
5 2 3 0
6 3 0
4 4 2 0
2 *3* *4* 0
0
Alice
Bob
Carol
Dave
example 3

name	initial	elim 3	votes	final
Alice	9.0	-	9.0	Elected
Bob	5.0	4.0	9.0	Elected
Carol	8.0	-	8.0	
Dave	4.0	-4.0	-	
void	0.0	0.0	0.0	
total	26.0	26.0	26.0	

Table 3. Examples showing nonlinearity of STV

2.4 Paradoxes

A number of strange effects are known in STV elections. These effects are often called paradoxes. Paradoxes are relevant for model-based testing since they make it harder to state properties that can be used in model-based testing.

Table 3 illustrates that being elected is nonlinear in the number of votes. The only difference between example election 2 and 3 is that two voters swapped their ballots from 4 3 (Dave, Carol) to 3 4 (Carol, Dave). As a consequence Carol has more votes in the initial stage of election 3 than in election 2. Nevertheless Carol is elected in election 2, but not in election 3. In these examples there are 26 votes and 2 candidates need to be elected, hence $Q = 9$. This paradox is not due to the small number of votes. The same effect occurs if we multiply the number of votes by any positive constant.

These kind of paradoxes limit the general properties one can use for testing election software. Among the best properties encountered are: 1) the total amount of votes (including nontransferable votes) is equal in all stage of the election, and 2) if a candidate reaches the quota in the first round she is always elected. These properties are way too weak for serious tests of election software. For this reason we have to test with a reference implementation.

An obvious condition to be satisfied is that no votes are lost in this algorithm. The sum of the votes of the candidates and the nontransferable votes should be equal to the initial number of valid votes after each iteration of the algorithm.

3 Testing Election Software

Testing the election software is approximating the property $\forall e \in E.\mathsf{iut}\ e = \mathsf{ri}\ e$ by executing $\forall e \in S\,.\,\mathsf{iut}\ e = \mathsf{ri}\ e$ for some finite set of elections $S \subset E$. The test suite S must be large enough to spot differences between the iut and the ri with high probability, but as small as possible to obtain results quickly. Since we had to do certification of tested software, we expected a correct iut. Hence we did not optimize the order of elements in S and preferred coverage above a small size.

3.1 The Notion of Equivalence of Election Results

In order to determine if $\mathsf{iut}\ e = \mathsf{ri}\ e$ holds for some election e we need to compare election results. In first approximation one is tempted to compare if the same candidates are elected. We use a much more precise notion of equivalence. Both programs yield a vote transfers table as shown above. Our notion of equivalence checks for textual equivalence of these vote transfer tables.

This notion of equivalence appears to be much more sensitive for slight differences in the iut and the ri. By looking only at the elected candidates we need a test case where such a difference influences the results of the election in order to note the difference. Although there will always exists such test cases that reveal the differences between the iut and the ri based on the elected candidates for all relevant problems, it will take much more effort to find those test cases. This would require special attention in the creation of test cases and most likely requires much larger test suites. Our more precise notion of equivalence is always right and spots differences quicker.

3.2 Test Suites

For our first certification we had to check a test suite of 808 test cases. This test suite contained handcrafted elections to test the correct implementation of specific aspects of the law as well as test cases taken from STV-elections world wide. Since this test suite was largely undocumented we added our own test cases to test borderline cases of the rules. It was easier and faster to design these test cases than to check if these things were covered in the provided test suite.

For the next certification we developed a similar test suite. During certification we added test cases to check our assumptions on incorrect behaviour of the iut. By coincidence the final test suite also contains 808 test cases.

The size of the test cases ranges from 2 candidates to over 100 candidates, and from a few votes to 99999. In the resulting vote transfer tables this takes up to 98 stages to determine the election result.

3.3 Test Suite Generation

To investigate the power of automatic test case generation we developed a straightforward data type representing elections (number of candidates, number of candidates to be elected, and a list of ballots). We created a generator

that generates valid instances of this data type (number of candidates to be elected smaller or equal to the number of candidates, and only valid ballots), see [7]. We can tune the size of the test cases by setting upper bounds for the number of candidates and the number of ballots. The generated instances were turned to election files and handled like the other test suites.

It appeared that even with small test cases the issues found by the standard test suites are also found by the generated test suite. We typically need more test cases to find an issue, but that is compensated by the ease of test suite generation. Even the total test time to find the issues was not larger for the generated test suite since the test cases are smaller and hence testing is faster.

4 Implementations Under Test

We tested two programs to compute election results in this setup. Both programs were only available as an executable, no source code or additional information was available. Hence the only option was black box testing. When an issue was found we had to deduce from the transfer tables what happened. We could not look into the code of the iut to verify our expectations of possible errors.

iut A This program was written in Delphi. Actually there were two versions of this program. One interactive version (2.6 MB) suited for experiments and single election results, and one bulk test driver (525 KB) that executes all elections in a given directory.

iut B This program was given as two Java archives (3.1 MB and 7.4 MB). We used an external script to apply it to all test cases in a given directory.

The programs obtained for testing were tested by their manufacturers and were given to us for certification. Our tests were no beta tests, but final testing to obtain certification.

5 The Functional Reference Implementation

The reference implementation was written in the functional programming language Clean. Since the reference implementation, ri, was written in first approach to test one iut, maintainability of the program was not considered to be an issue. We did not expect to test a second implementation ever in the future. Since the program ri was used as reference implementation correctness was considered to be very important. Since the program ri would be used for certification and the speed of certification was not an issue, we always chose simple and obviously correct solutions instead of smart and more efficient solutions. For instance we always used lists of values and candidates instead of search trees or some other advanced data structure.

Whenever possible we used G∀st in the standard way to test components of this program.

5.1 Numbers with Five Digit Precision

Numbers with five digit precision and truncation after operations play an important role in the election software. All calculations of votes and vote transfers have to be done in five digit precision. For our reference implementation we need a stable implementation of those numbers.

In order to avoid rounding errors and possible overflows we represent numbers with five digit precision used for administration of the number of votes by multiplying them by 10^5 and storing this number as an integer with infinite precision (BigInt).

```
:: Fixed = Fixed BigInt

class toFixed a :: a → Fixed
instance toFixed Int where toFixed i = Fixed (FACTOR * toBigInt i)

FACTOR =: toBigInt (10^PRECISION)
PRECISION =: 5
```

The most important numerical operations on this type are implemented as:

```
instance + Fixed where (+) (Fixed x) (Fixed y) = Fixed (x + y)
instance − Fixed where (−) (Fixed x) (Fixed y) = Fixed (x − y)
instance * Fixed where (*) (Fixed x) (Fixed y) = Fixed ((x*y)/FACTOR)
instance / Fixed where (/) (Fixed x) (Fixed y) = Fixed ((FACTOR*x)/y)
```

The implementation of these operations is tested by stating some standard properties of numbers as property in G∀st.

```
pAssocAddFixed :: Fixed Fixed Fixed → Bool
pAssocAddFixed a b c = (a+b)+c == a+(b+c)

pAddSubFixed :: Fixed Fixed Fixed → Bool
pAddSubFixed a b c = a−b−c == a−(b+c)
```

Due to truncation many arithmetic properties do not hold for Fixed numbers. Some examples are:

```
pDistMulFixed :: Fixed Fixed Fixed → Bool
pDistMulFixed a b c = (a+b)*c == a*c+b*c

pAssocMulFixed :: Fixed Fixed Fixed → Bool
pAssocMulFixed a b c = (a*b)*c == a*(b*c)
```

As expected properties pAssocAddFixed and pAddSubFixed test successfully in G∀st. When testing the properties pDistMulFixed and pAssocMulFixed it quickly finds relevant counterexamples. The first counterexample found for pDistMulFixed after 75 tests was $-0.33333\ 1.0\ 0.5$. The first counterexample for pAssocMulFixed was $0.33333\ 0.5\ 2.0$ after 259 tests.

Our test results shoes clearly that using some form of floating point numbers, e.g. doubles, and a special print function that truncates to five digits does not

work correctly. One of the systems under test used such an implementation in early versions. This was a source of a large number of problems.

It is obvious that rounding problems can be avoided by performing all computations with rational numbers of infinite precision. However, this is not what the law on the elections prescribes. Computing with such rational numbers and truncating them to five digit precision whenever they are needed would introduce a source of new computational differences.

5.2 Administration of Candidates

A pile of identical ballots is represented by the type Ballot.

```
:: Ballot
 = { count   :: Fixed   // number of ballots in this pile
   , value   :: Fixed   // the value of these ballots, initially the value is one
   , order   :: [Int]   // the order of candidates on this ballot
   }
```

In a similar way we use a record to store the information about candidates.

```
:: Candidate
 = { ballots :: [Ballot] // the ballots currently assigned to this candidate
   , votes   :: Fixed    // the current number of votes
   , status  :: Status   // the status of this candidate, see below
   , cName   :: String   // the candidate name
   , cNumber :: Int      // the candidate number
   , trace   :: [String] // trace info to build transfer table
   , history :: [Fixed]  // the number of votes in previous stages
   }

:: Status = Eliminated | Elected | Running
```

The implementation of the election algorithm is basically just a parser for a data file and careful bookkeeping according to the rules in the law.

5.3 Size of Executable

The size of the executable that generates the vote transfer table for ri and compares it with the table generated by iut is only 141 KB. This is more than an order of magnitude smaller than the iut's. This is partly caused by the absence of a GUI in the ri. Our ri consists of 591 lines of Clean code.

There is no information available about the development time of the iut's. Hence we cannot compare it with the time needed to develop our ri. Our ri was developed and tested within a week.

6 Issues Found

The test system signals an issue each time the vote transfer tables generated by the iut and ri are not identical. All issues are examined manually to find

the reasons causing these differences. These reasons can be categorized in the following groups.

1. Syntactical differences in the generated output. Since the vote transfer tables are compared textually the system is very sensitive to further irrelevant layout details. Some examples of differences in this class are:
   Trailing zeros A single vote can be formatted as `1`, `1.0` and `1.00000`. Although these representations are clearly equivalent, they are textually different.
   Votes of eliminated candidates All votes that were assigned to an eliminated candidate are transferred to other candidates, or added to the nontransferable votes. Hence these candidates will always have no votes left at all. This can be indicated by a blank field, a `-`, or `0.0`.
   Different number layout One of the iut's used the database Microsoft SQL Server 2005. The way the numbers are printed and parsed by this database system depends on the language settings. If Microsoft Vista is set to Dutch the number `1.0` is displayed as `1,0`.
   Removing spaces from names This obviously has no meaning for the election result, but does cause textual differences in the vote transfer table.
   String quotes There were different rules used to enclose strings (like the names of candidates) in string quotes (i.e. `"Koopman"` or `Koopman`).
   Most issues are solved by adapting the generated vote transfer table of the reference implementation to the iut since this was the fastest way to progress.
2. Syntactically incorrect number of votes (e.g. 9 digits precision instead of 5).
3. Losing the last vote in a completely full ballot. This was an error in the iut that was corrected.
4. One of the iuts looses candidates if the number of candidates in a test case was larger than some fixed number. This results in unpredictable behavior since results of previous elections might be used instead of the current election. This was caused by the loader component of the iut that used a upper bound of the number of candidates that was too small (25 in the first approach, 100 in a later version). The problem is handled by setting this upper bound of the iut sufficiently large for all test cases.
5. Unexpected characters (like digits and characters like '-') in the name of candidates caused similar effects.
6. The law [10] states in rule 48 about vote transfer that "*the calculation being made to five decimal places (any remainder being ignored)*". In the calculation
$$\text{tranfer} = \frac{\text{surplus} \times \text{value\_of\_ballot}}{\text{total\_number\_of\_votes}}$$
   this can be interpreted in two ways: **1)** truncate to 5 places after each operation, or **2)** truncate to 5 places after the entire computation. In some cases this produces different results. Hence it might influence which candidates are elected.
7. The law [10] states in rule 50:

(3) "*The returning officer shall, in accordance with this article, transfer each parcel of ballot papers referred to in paragraph (2)(a) to the continuing candidate for whom the next available preference is given on those papers and shall credit such continuing candidates with an additional number of votes calculated in accordance with paragraph (4).*"

(5) "*This rule is subject to rule 52.*"

Where rule 52 states: (2) "*Where the last vacancies can be filled under this rule, no further transfer shall be made*".

For the election of candidates it does not matter if we give (3) preference over (5) or the other way around, for the resulting vote transfer tables it does matter. All programs give rule 50 (3) preference over rule 50 (5), and hence rule 52.

8. If the rules specify that some candidate has to be eliminated, the candidate with the lowest number of votes has to be eliminated. If two or more candidates have this amount of votes we have to look into the history to see if there was a difference in amount of votes. One of the implementations did this wrong.
9. The rules do not specify how to treat blank ballots. In some election systems a blank vote is valid. In this system they are invalid. This is done since blank votes have an influence on the quota Q if they were treated as valid.
10. Also other kind of invalid ballots are not covered by the provided rules. A ballot containing a nonexisting candidate is invalid. The entire ballot, not only the invalid candidate, is ignored by the election algorithm. The Scottish Executive has advised that this occurrence would not be an issue in the May elections as any non-existing candidates on the ballot paper would simply be ignored.
11. Also forms containing a candidate twice are considered to be invalid and are ignored in the election process. However, such a form will be harmless in the election algorithm. When the second occurrence of the candidate is considered by the algorithm the candidate has either been elected or eliminated. In both situations the candidate number will be ignored by the algorithm.
12. In general an STV election contains the possibility for candidates to withdraw themselves. In the Scottish elections this cannot occur. Some test cases taken from other STV elections contain withdraw candidates. Programs `iut A` and `ri` handles this correctly, `iut B` does not handle this. Since candidates cannot withdraw themselves in the Scottish elections, these test cases are ignored.

7 Test Results

During the tests we found a large number of issues. These issues can be grouped as indicated above. In this Section we indicate the issues found by source.

7.1 The law

The Scottish law on these elections [10] specifies in an imperative way how a human being can compute the election result. On a number of minor points the law is not absolutely clear:

1. It is specified how to handle invalid ballots, but not how to handle an invalid ballot. Obvious possibilities are ignoring the ballot altogether, move it to the nontransferable pile immediately, treating it as an empty ballot, and using the part of the vote that is valid (if any).
2. Rule 48 states that numbers must have 5 decimal places precision and numbers must be truncated rather than rounded. It is unclear whether this must be done after each step, or after an entire computation. A representative of the Scottish executive indicated that the last interpretation is preferred.
3. Rule 50 allows two possible interpretations on the necessity of vote transfer if after the elimination of some candidate the remaining number of vacancies is equal to the remaining number of candidates. As indicated above, this does not influence the elected candidates. All programs do the vote transfer.

7.2 The Reference Implementation

After building and testing ri on its own only one real change was necessary. As 'obvious' in a functional programming language like Clean we implemented the five digit numbers as an abstract data type and implemented the needed operators (addition, subtraction, multiplication and division) for this type. This implies truncation after each operation, but that is not the interpretation of the law preferred by the Scottish executive. Hence we had to turn the abstract type into an ordinary type and adapt the computation of the vote transfer.

Other modifications are layout details of the vote transfer table to make it textually identical to the details of the corresponding tables of iut A and iut B. This implies that there are two versions of this layout.

7.3 The IUTs

All other issues where due to problems with iut A and iut B. There were more problems as expected for a certification project. The iut B caused significantly more issues than iut A.

7.4 Execution Speed

Much to our surprise the execution speed of the ri was considerably higher than the execution speed of iut A. The speed difference was a factor 2 to 5, depending a little on the size of the input and the number of rounds needed in the election. This was unexpected since lazy functional languages are not famous for their speed. Especially since we have always chosen the simple and obviously correct solution instead of smart and efficient solutions we did not expect ri to be faster than any iut. We did nothing special to make ri efficient.

The speed difference between ri and iut B was striking: ri is about a factor 250 faster than iut B. This is partly caused by the fact iut B uses a database intensively. The amount of data to be maintained by the election software is not that large that a database is required.

This low performance was a bottleneck in the tests. It takes about 30 minutes for the ri to compute the results for a large test suite and to compare these with the results of the iut. The program iut B takes more than five days to process the entire test suite. Each time we find an error in the iut we have to repeat all tests in the entire test suite.

7.5 Choices

When the votes of a candidate must be transferred and there are two or more candidates having exactly the same amount of votes, also in all previous stages, the law states that a candidates must be chosen by lot. The interactive version of the iut's ask the user to indicate a candidate, iut A has to possibility to chose a candidate pseudo randomly.

In practise this is very rare. However, it is easy to generate test cases where n candidates have the same amount of votes. In the worst case there are $n!$ possibilities to eliminate them one by one (it is often not necessary to eliminate them all).

Our first plan was to generate all possible vote transfer tables and see if one of them is equal to the table generated by the iut. However it is obvious that this does not work for test cases with for instance 10 or more candidates with an equal number of votes. Hence we fixed the elimination order to be used in the test.

7.6 Vote Transfer Tables

In retrospect it would have been easier to transpose the vote transfer table. Now each line contains the votes of one candidate during all stages. If we find a difference it is the first candidate that has in some stage a different number of votes in the tables from the iut and the ri. In tracking down the source of such an issue it is much more practical to have an indication of the first stage that shows a difference. Although these versions of the tables are equivalent, a transposed version would have been more convenient if we could have anticipated the number of issues to investigate more accurately.

8 Conclusions

This paper reports on the certification of election software by black-box testing. Due to the absence of suited properties we tested the iut by comparing its results with a reference implementation. The test results indicate that testing was worthwhile for both implementations tested. None of the iuts was correct. The construction of the reference implementation and the tests also indicate some

points of underspecification in the law regulating these elections. We compared handcrafted test suites extended by real election results with an automatically generated test suite. Both test suites were able to spot the same errors in the iuts. Neither of these test suites was significantly more effective in finding issues.

The trend we signal in this paper is the use of functional programs as reference implementation. Functional languages appeared to be very suited for this purpose, it is easy and fast to produce a reference implementation. We were very pleased that this program caused by far the least number of issues during the tests. Much to our surprise it was also clearly the fastest implementation, although we did nothing to make the reference implementation efficient.

Acknowledgement We thank Steven Castelein for helping us to develop scripts to execute series of test runs automatically and Peter Achten for his feedback on draft versions of this paper.

References

1. H. Aslaksen and G. Mcguire. Mathematical aspects of irish elections. *Irish Mathematics Teachers Association Newsletter*, 105:40–59, 2006.
2. F. P. C. and B. S. J. Paradoxes of preferential voting. *Mathematics Magazine*, 56(4):207–214, 1983.
3. H. Droop. On methods of electing representatives. *Journal of the Statistical Society of London*, 44(2):141–202, 1881.
4. D. M. Farrell. *Comparing electoral systems*. Prentice Hall/Harvester Wheatsheaf, London, New York, 1997.
5. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: generic automated software testing. In R. Peña and T. Arts, editors, *Revised Selected Papers of the 14th International Workshop on the Implementation of Functional Languages, IFL '02*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer-Verlag, 2003.
6. P. Koopman and R. Plasmeijer. Fully automatic testing with functions as specifications. In Z. Horváth, editor, *Proceedings of the 1st Central European Functional Programming School, CEFP '05*, volume 4164 of *Lecture Notes in Computer Science*, pages 35–61, Budapest, Hungary, July 2005. Springer-Verlag.
7. P. Koopman and R. Plasmeijer. Generic generation of elements of types. In *Proceedings of the 6th Symposium on Trends in Functional Programming, TFP '05*, pages 163–178, Tallin, Estonia, 23-24, Sept. 2005. Intellect Books. ISBN 978-1-84150-176-5.
8. R. Plasmeijer and M. van Eekelen. *Concurrent Clean language report (version 2.0)*, Dec. 2001. `http://www.cs.ru.nl/~clean/`.
9. The Electoral Commission. Vote Scotland. `http://www.votescotland.com`.
10. The Scottish Ministers. Scottish local government elections order 2007, 2006. Rule 45–52.
11. Wikipedia. Single transferable vote. `http://en.wikipedia.org/wiki/Single_transferable_vote`.