ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4
July 1997

Paper:

The Single Transferable Voting System: Functional Decomposition in Formal Specifications

Michael R. Poppleton

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

The Single Transferable Voting System: Functional Decomposition in Formal Specification

Michael R. Poppleton

School of Mathematical and Information Sciences, Coventry University
Coventry, West Midlands, UK

Abstract

A formal specification is presented in the Z language for a simplified version of the Single Transferable Vote form of election. This is a correctness-critical application which is one of a class of related and interesting applications, i.e. electoral models. This specification is based on the form of election defined by the Students' Representative Council of the University of Cape Town, and demonstrates the utility of formal specification for requirements validation. A succinct statement of the algorithm is given using the schema calculus.

The specification provides a vehicle for contributing to the current debate on good Z specification style. Brief discussion of specification styles in the literature sets the context for an overview style employed here, one of functional decomposition.

1 Introduction

The Single Transferable Vote (STV) is an example of the class of complex algorithms for vote counting and result determination in the various forms of election by proportional representation (PR). These algorithms are well defined and have been extensively validated in elections around the world. STV is the Anglo-Saxon version of proportional representation and has operated for many years, for example, in parliamentary elections in the Republic of Ireland, Northern Ireland, Malta and Tasmania.

In July 1991 the Church of England, which carries out internal elections by STV, decided to allow the automated computation of election results. This focussed attention on the unusual nature of a computerised STV algorithm: a program requiring the highest possible integrity, without being either safety- or security-critical. Mukherjee and Wichmann [6, 5] report an exercise in formal specification of the algorithm in the VDM-SL language [1], and its validation by animation.

This paper presents a formal specification in the Z language [9] of the STV algorithm, in its simplified form as defined for SRC elections at the University of Cape Town (UCT) [8]. The full definition is available from the Electoral Reform Society (ERS), [7]. The need for the work was identified by Prof. Chris Brink of the Mathematics Department at UCT. A program was written to support the Student Representative Council's (SRC) move to STV. The formal specification was seen as a support to its validation, principally by closely examining and precisely describing the simplified narrative specification produced by the SRC.

This work has found a number of errors in the UCT specification, both of misinterpretation of the ERS rules, as well of exclusion of many necessary boundary and limit conditions addressed by the full algorithm. For this paper it is useful to give a corrected narrative specification, and simply indicate errors in the original in parentheses, e.g. ($e1$). These errors are documented elsewhere.

One element of this simplification of STV is in using real numbers in the specification, and ignoring the

The Single Transferable Voting System: Functional Decomposition in Formal Specification

limited (2 decimal places) precision issues raised in the ERS rules. Otherwise the simplification consists mainly of aggregating ballot transfers that are done on a per-candidate basis in the rulebook. The finer granularity of the rules, which complicates the algorithm considerably, appears to be designed to minimise the manual work of transferring ballots and modifying their weights. Such a requirement is not necessary in a computerised count.

This work was completed in concept without awareness of the work of Mukherjee and Wichmann [6, 5]. This simpler specification provides a more easily comprehensible case study for a discussion on specification style.

A timely debate has recently arisen on the structuring of formal specifications in Z [4]. MacDonald and Carrington specify the robot part of the Production Cell Case Study [3] using five different styles, and discuss their utility in various application situations. They conclude that ‘...there is no single correct technique: different techniques suit different situations’.

The styles discussed, and points raised on their applicability are (briefly):

1. *flat* style: the simplest style; global state schema is not decomposed; all operations are defined for the global state schema. Suitable for small specifications with no significant component structure. Duplication of similar constructs and operations. No information hiding or support for reuse.
2. *partitioned* style: global state is decomposed into a separate schema for each component. Duplication. Information hiding. No reuse support.
3. *parameterised* style: a generic template schema is used to instantiate similar components which differ on some parameter. Information hiding. Some reuse support. Local operation framing can be complex. Not suitable for multiple instantiations of the same component type.
4. *library* style: an extension of the Z language providing a schema instantiation mechanism to improve on styles 2 and 3. Information hiding. Provides reuse, avoids instantiation problems of ‘parameterised’ style.
5. *Object-Z* style: an object-oriented extension of the Z language. Not discussed here.

Operation promotion or framing is well understood but counter-intuitive. It is usually introduced in terms of promoting a local operation on a component in an indexed collection, to a global operation [11]. The discussion in [4] concerns promotion of individually defined or instantiated component schemas. Promotion adds complexity in styles 2, 4 and especially 3 above.

A *free* promotion [11] is one where the local update is unconstrained by the global state. The Production Cell robot is an example of constrained promotion. The five styles indicate various ways of describing the global constraint, in either local or global state schema. When the global specification is considered within which the robot is embedded, it is found that

...relationships between components at the production cell level...impose additional constraints on operations defined for a single component... (Promotion) requires read access to state variables of multiple components, but changing these variables is mostly not required. [4]

The STV algorithm presents a specification ‘situation’ not covered by the Production Cell case study. Each stage of the count requires ballot transfer between candidates, that is, simultaneous update of each of a collection of parcels of ballots which is indexed by candidate. This is a more complex situation for which promotion may be definable, but even harder to understand. The structure of this specification exemplifies a ‘functional decomposition’ style, which is intuitively appealing and avoids any use of promotion. Although deterministic, as is required of the specification of an algorithm, it is abstract insofar as specification of the core dynamic behaviour (ballot transfer and counting) is implicit, allowing implementation freedom.

The Single Transferable Voting System: Functional Decomposition in Formal Specification

The specification was produced by experimenting with various state schema decompositions and appropriate dynamic constructs. It was iteratively improved by simplifying schemas and functions (there is clearly scope for work on the psychological/mental modelling ‘method’ involved in formal specification).

A brief narrative algorithm is presented. The state model is presented and followed by operation schema definitions. Axiomatically defined functions are presented in logical sequence in the discussion. The algorithm specification is completed by an elegant description of its time behaviour in the schema calculus. During the discussion, proof obligations are informally stated. Some are necessary, such as proving that an implicit definition is functional. Others are desirable in that, while not required by the rulebook [7], they provide further validation.

An overview of specification structure is then given, demonstrating that this design-oriented specification is structured in a hierarchic, functionally decomposed fashion with suitable levels of abstraction, encapsulation and separation of concerns. Concluding remarks give the context of the work and future plans.

An apology is due to the reader for some syntax errors in the Z, down to deficiencies in the *oz.sty* latex file used to prepare this paper. Specifically, the problems are: if-then-else, injective sequence, bag membership, sub-bag relation.

2 The Single Transferable Vote

The single transferable vote was proposed in the last century by Thomas Hare and John Stuart Mill. It is a method of election providing for preferential voting in multi-member constituencies. Among its key aims are the proportional representation of political views and opinions, and to ensure that as many voters as possible can identify one or more of the elected representatives that they helped to elect.

Each voter completes a single paper on which she expresses an order of preference for one or more candidates. Several seats should be available for the constituency to enable different voter views to be represented. A candidate is elected on achieving the *droop quota*, which is the minimum number of votes which, if attained by as many candidates as there are seats available, leaves at most a quota of votes unused. This quota is given approximately by the total valid vote, divided by one more than the number of seats. For example, if 1000 votes are cast in a contest for 3 seats, then the quota is 251. Proportionality is achieved by the transfer of unused ballots to second and subsequent choice candidates. Ballots are regarded as ‘unused’ either when they are surplus to the quota for election, or are too few to elect a weak candidate - a party may field more than one candidate in a constituency.

Very briefly, the count proceeds as follows. Valid ballots are sorted by first preference, and counted. The quota is computed. Candidates achieving quota are elected.

All the transferable papers of any candidate with a surplus above the quota are transferred to other, continuing, candidates in accordance with next available preferences as expressed by the electors, the transfer value of each paper being determined by sharing the surplus equally between the transferable papers. Candidates with fewest votes are then excluded in turn and their voting papers are transferred to continuing candidates in accordance with next available preferences. [7]

The vote *weighting* mechanism is required to enable *all* transferable ballots to be transferred, avoiding the obvious mechanism of selecting the surplus as a random subset of ballots. This process of transfer continues until all seats are filled.

We give some definitions, followed by a five-step description of the algorithm. Inevitably this description is brief; the reader should refer to the ERS rulebook [7] for the authoritative definition.

The Single Transferable Voting System: Functional Decomposition in Formal Specification

A **first preference** is the figure ‘1’ recorded on the ballot for a candidate. Subsequent preferences are numbered ‘2’, ‘3’ and so on. A **surplus** is the amount by which a candidates’ vote exceeds quota, at any stage of the algorithm. A **stage** refers to any point in execution of the algorithm involving vote counting or transfer. An **invalid** ballot is one which does not have a unique first preference unambiguously stated. A **transferable ballot** is one on which “the next preference in order, passing over earlier preferences for candidates already elected or excluded, ...(is) for a continuing candidate” [7]. A ballot becomes **nontransferable** when it ceases to be transferable, or more than one candidate is marked with the next preference number, or when no candidate is marked with the next preference number (e1).

The algorithm operates as follows:

1. Validate all ballot papers, removing invalid ones. Count them and establish the total vote. Sort the ballots and allocate each to its first-preference candidate. The sum of first preference votes should equal the total vote. A quota of the minimum votes needed for election of a candidate is computed (rounded up to two decimal places; see also formal definition):

$$quota = number\ of\ valid\ votes / (1 + number\ of\ seats)$$

Candidates thus achieving quota are deemed to be elected.

2. Elected candidates’ vote surpluses over quota are computed. Each ballot for each elected candidate is allocated a “transfer weight” as follows:

$$weight = surplus\ ballots / total\ value\ of\ transferable\ ballots\ for\ this\ candidate$$

In the event that the surplus is greater than or equal to the value of transferable ballots, the weight is set to 1. All transferable ballots are then transferred to next preference candidates, with appropriate transfer weight. Any ballot which has become nontransferable is transferred to the nontransferable ballot collection.

3. Terminate if all seats are now taken. If the number of candidates still continuing equals the number of remaining seats, elect such candidates and terminate.
4. Any unelected candidates now achieving quota are elected, any surplus votes being transferred with fractional weight (as per step 2) (e6). Go to step 3.
5. Once such transfer fails to deliver candidates achieving quota, the lowest-vote candidate is identified. If more than one candidate has polled the lowest number of votes, then

amongst this group of low scorers, the one with the lowest score at the earliest stage of the counting process, is selected for exclusion. If backtracking in this fashion fails to yield a single candidate, only producing a reduced set, then from this reduced set one is selected for exclusion at random. [8]

The selected candidate is excluded, and her votes are transferred at full weight (as per step 2). Go to step 3.

3 Real Numbers

The basic mathematical theory of the Z language does not currently contain the real numbers, although this question has been addressed [10]. The specification of the reals in the Z language, and the embedding of the integers into the reals is beyond the scope of this paper. We therefore simply assert the existence of the reals and their required structure, and assume the appropriate embedding of the integers, overloading the arithmetic symbols in the usual way.

$$\begin{array}{l}
 [\mathbb{R}] \\
 \mathbb{Z} \subseteq \mathbb{R} \\
 \hline
 \begin{array}{l}
 - < -, - \leq -: \mathbb{R} \leftrightarrow \mathbb{R} \\
 - + -, - -, - * -: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
 - / -: \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \rightarrow \mathbb{R}
 \end{array} \\
 \hline
 (- \leq -) = (- < -) \cup \text{id } \mathbb{R}
 \end{array}$$

4 Specification: State

We use a given type *CAND* for candidates. Voter identity is irrelevant, hence should not be defined. Use of a given type indicates that internal structure of the type is not of interest in this specification. The number of seats *noSeats* is a given constant, as is the number of candidates *noCands*. For a nontrivial election there must be more candidates than seats.

$$\begin{array}{l}
 [CAND] \\
 \hline
 \begin{array}{l}
 noSeats : \mathbb{N}_1 \\
 noCands : \mathbb{N}_1
 \end{array} \\
 \hline
 \begin{array}{l}
 noCands > noSeats \\
 noCands = \# CAND
 \end{array}
 \end{array}$$

The *Ballot* schema specifies the single transferable, multiple preference ballot. The schema has two components. Voter preferences are expressed as a finite function from *CAND* to the whole numbers. A positive, real-valued ballot *value* is required to record the ballot's "weight" during any ballot transfer stages. The predicate part of the schema constrains values of the components to model meaningful ballot papers only: every stated preference must be for a candidate, and at least a first preference must be stated. The absence of further constraining predicates indicates that, for example, duplicate preferences are allowed, as are discontinuities (gaps) in the given preferences. A ballot may become nontransferable at some stage; this will be detected by function *nextPref*, defined later.

$$\begin{array}{l}
 \text{Ballot} \\
 \hline
 \begin{array}{l}
 preference : CAND \multimap \mathbb{N}_1 \\
 value : \mathbb{R}
 \end{array} \\
 \hline
 \begin{array}{l}
 \text{dom } preference \subseteq CAND \\
 \# preference^{-1}(\{1\}) = 1 \\
 0 \leq value \leq 1
 \end{array}
 \end{array}$$

The collection of ballots cast is a given constant. It is best modelled as a finite bag since identical-preference ballots are possible, making the set an inadequate model. The obvious "key" for a function model for the

The Single Transferable Voting System: Functional Decomposition in Formal Specification

collection, voter identity, is neither available nor relevant, because of the anonymity of the act of voting. During operation of the algorithm, the complete bag of ballots is split up into smaller bags (the regulations refer to “parcels” [7]) for each candidate. The bag is a generic construct enabling a collection to have duplicates. It is a function from the element type to the whole numbers, recording the count of each element in the collection:

$$\text{bag } X == X \rightarrow \mathbb{N}_1$$

We define a generic type synonym for finite bags, and then define the (constant) collection of all valid ballots cast in this election. We assume that ballots in *BallotBag* initially have *value* 1; the initial state schema *InitSystem* below enforces this to avoid the multiplicity of *Ballot* bindings that would arise from an undetermined *value*.

$$\text{finBag } X == \{ B : \text{bag } X \mid (\text{dom } B) \in \mathbb{F} X \}$$

$$\mid \text{BallotBag} : \text{finBag } \text{Ballot}$$

We now require some theory for finite bags. The above definition, and the following theory borrows the style of recursive definition used in Hayes’ original paper on bags [2]. First, a function is needed to give the size (total number of elements) of a finite bag.

$\begin{aligned} & \text{bagSize} : \text{finBag } X \rightarrow \mathbb{N} \\ & \text{bagSize } [] = 0 \\ & \forall x : X; n : \mathbb{N}_1 \bullet \\ & \quad \text{bagSize } \{x \mapsto n\} = n \\ & \forall B1, B2 : \text{finBag } X \bullet \\ & \quad \text{bagSize}(B1 \uplus B2) = (\text{bagSize } B1) + (\text{bagSize } B2) \end{aligned}$	$\begin{aligned} & \text{bagSize} : \text{finBag } X \rightarrow \mathbb{N} \\ & \text{bagSize } [] = 0 \\ & \forall x : X; n : \mathbb{N}_1 \bullet \\ & \quad \text{bagSize } \{x \mapsto n\} = n \\ & \forall B1, B2 : \text{finBag } X \bullet \\ & \quad \text{bagSize}(B1 \uplus B2) = (\text{bagSize } B1) + (\text{bagSize } B2) \end{aligned}$
---	---

In fact we will require a form of bag “size” customised to a bag of ballots: we need to know its cumulative value as the sum of all *value* components, of course taking duplicates into account. The definition of *bagValue* is almost identical to *bagSize*:

$\begin{aligned} & \text{bagValue} : \text{finBag } \text{Ballot} \rightarrow \mathbb{R} \\ & \text{bagValue } [] = 0 \\ & \forall b : \text{Ballot}; n : \mathbb{N}_1 \bullet \\ & \quad \text{bagValue } \{b \mapsto n\} = b.\text{value} * n \\ & \forall B1, B2 : \text{finBag } \text{Ballot} \bullet \\ & \quad \text{bagValue}(B1 \uplus B2) = (\text{bagValue } B1) + (\text{bagValue } B2) \end{aligned}$	$\begin{aligned} & \text{bagValue} : \text{finBag } \text{Ballot} \rightarrow \mathbb{R} \\ & \text{bagValue } [] = 0 \\ & \forall b : \text{Ballot}; n : \mathbb{N}_1 \bullet \\ & \quad \text{bagValue } \{b \mapsto n\} = b.\text{value} * n \\ & \forall B1, B2 : \text{finBag } \text{Ballot} \bullet \\ & \quad \text{bagValue}(B1 \uplus B2) = (\text{bagValue } B1) + (\text{bagValue } B2) \end{aligned}$
--	--

We further require a special “bag union” generic function. The major state component *VoteMass* will represent ballot-to-candidate allocation as a function from candidates to finite bags of ballots. We will need to describe aggregate properties of the range of such a function, i.e. of the combined bag of all candidates’ bags of ballots. *bagRange* collects the range elements of a *finBag*-producing function into a composite bag.

$[X, Y]$
$bagRange : (X \rightarrow finBag\ Y) \rightarrow finBag\ Y$
$bagRange\ \emptyset = []$ $\forall x : X; B : finBag\ Y \bullet$ $\quad bagRange\ \{x \mapsto B\} = B$ $\forall f, g : X \rightarrow finBag\ Y \mid disjoint\ (dom\ f, dom\ g) \bullet$ $\quad bagRange(f \cup g) = (bagRange\ f) \uplus (bagRange\ g)$

Returning to the STV algorithm, we define the *quota* to be the minimum number of votes required for the election of a candidate. This requires definition of a function *trunc* to return the integer part of a real number. We have used ERS' quota definition (e2).

$trunc : \mathbb{R} \rightarrow \mathbb{Z}$
$\forall r : \mathbb{R} \bullet \exists_1 i : \mathbb{Z} \bullet$ $\quad i \leq r < i + 1$ $\quad trunc\ r = i$

$quota : \mathbb{R}$
$($ $\quad (let\ q \hat{=} (bagSize\ BallotBag) / (noSeats + 1) \bullet$ $\quad q \in \mathbb{N} \Rightarrow quota = q \wedge$ $\quad q \notin \mathbb{N} \wedge q > 100 \Rightarrow quota = trunc\ q + 1 \wedge$ $\quad q \notin \mathbb{N} \wedge q < 100 \Rightarrow quota = trunc\ q + 0.01$ $\quad)$ $)$

We now have sufficient structure in place to define the central part of the state schema, i.e. the collection of ballots and their association with the candidates, called *VoteMass*. Starting from an initial state which has all ballots assigned to their first-preference candidates, counting operations will work by structured and weighted transfer of ballots between candidates.

The overall vote mass has two components: a finite function mapping each candidate in *CAND* to her finite bag of ballots, and the bag of accumulating nontransferable ballots. Candidates achieving no votes will be mapped to an empty bag. Two candidates may not simultaneously be allocated identical-preference ballots, i.e. all ballots representing a given assignment of preferences will always reside with precisely one candidate (The expression $dom\ (voteMass\ s)$ denotes the set of distinct ballots in the bag of candidate s). Each candidate must be some preference of each of her ballots.

The existentially qualified predicate below ensures a correct bijective correspondence between ballots in the constant *BallotBag*, and ballots in the dynamic *VoteMass*. The latter set of ballots, i.e. the range of this bijection, is precisely all ballots held by any candidate, together with all nontransferable ballots. Each corresponding pair of ballots is equal down to preference and count in the containing bag (but not in *value*: this will change with ballot transfers).

The Single Transferable Voting System: Functional Decomposition in Formal Specification

<i>VoteMass</i>
$voteMass : CAND \mapsto finBag\ Ballot$ $nonTransferables : finBag\ Ballot$
$dom\ voteMass = CAND$ $\forall s, t : CAND; b : Ballot \bullet$ $s \neq t \Rightarrow dom(voteMass\ s) \cap dom(voteMass\ t) = \emptyset \wedge$ $b\ in\ voteMass\ s \Rightarrow s \in dom\ b.preference$ $\exists corr : Ballot \mapsto Ballot \bullet$ $(let\ ranCorr \hat{=} (bagRange\ voteMass) \uplus nonTransferables \bullet$ $dom\ corr = dom\ BallotBag \wedge ran\ corr = dom\ ranCorr \wedge$ $\forall b : Ballot \mid b \in dom\ corr \bullet$ $b.preference = (corr\ b).preference \wedge$ $count\ BallotBag\ b = count\ ranCorr\ (corr\ b)$ $)$

Next it is necessary to specify the state components for recording candidate scores, or ballot counts, at the stages of the algorithm. State schema *Score* records a candidate's identity and count at a given counting stage. The count is real-valued, representing the sum of the *value* components of the candidate's associated ballots.

The aggregate of scores for all candidates at a given stage is specified by schema *Stage*. This comprises a sequence of *Scores* (one per candidate) and a count of nontransferable ballot values. The sequence is injective since each candidate must be uniquely identified; the sequence is furthermore ordered on decreasing count.

<i>Score</i>
$cand : CAND$ $count : \mathbb{R}$
$count \geq 0$

<i>Stage</i>
$scores : seq\ Score$ $nonTransCount : \mathbb{R}$
$\#scores = noCands$ $nonTransCount \geq 0$ $\forall i, j : \mathbb{N}_1 \mid i, j \in dom\ scores \wedge i < j \bullet$ $(scores\ i).count \geq (scores\ j).count \wedge$ $(scores\ i).cand \neq (scores\ j).cand$

The stage count function is specified axiomatically. The nontransferable ballot count is given by accumulating the *value* components of the nontransferable ballots. The candidate's scores are more indirectly specified by stating the cardinality of the *Score* sequence *scores* , and specifying each range element. A proof obligation is that this relation is in fact functional.

The Single Transferable Voting System: Functional Decomposition in Formal Specification

$stageCount : VoteMass \rightarrow Stage$	
$\forall v : VoteMass \bullet$	$(stageCount\ v).nonTransCount = bagValue\ v.nonTransferables \wedge$
$\forall c : CAND \bullet$	$\exists s : Score \bullet$
	$s.cand = c \wedge$
	$s.count = bagValue(v.voteMass\ c) \wedge$
	$s \in ran(stageCount\ v).scores$

The STV System state can now be specified as comprising the (mutually exclusive and exhaustive) sets of elected, continuing for election, and excluded candidates, the total vote count, the sequence of counting stages, and the *VoteMass*. The partition operation succinctly structures the candidates *CAND*. Only continuing and elected candidates may hold nonempty bags of ballots: excluded candidates always lose all their ballots on transfer; elected candidates retain their ballots only when exactly achieving quota, when transfer would have no effect. The total vote count is given by the size of the *BallotBag* of all valid votes.

A desirable proof obligation is that the total count is equal at all counting stages to $\#elected * quota + total_residual_ballot_value$.

<i>System</i>	
$elected, continuing, excluded : \mathbb{F}\ CAND$	
$totalCount : \mathbb{R}$	
$stages : seq\ Stage$	
$VoteMass$	
	$\langle elected, continuing, excluded \rangle partitions\ CAND$
	$dom(voteMass \triangleright []) \subseteq continuing \cup elected$
	$totalCount = bagSize\ BallotBag$

Before proceeding to initial state, we require a function *sum* to sum the elements of a finite set of numbers.

$sum : \mathbb{F}\ \mathbb{R} \rightarrow \mathbb{R}$	
$sum\ \emptyset = 0$	
$\forall S : \mathbb{F}_1\ \mathbb{R};\ r : \mathbb{R} \bullet$	$sum\ \{r\} = r \wedge$
	$sum((S \setminus \{r\}) \cup \{r\}) = r + sum\ S$

The initial system state constitutes step 1 of the counting algorithm, apart from the election of first-preference candidates. It specifies that all candidates are continuing (available) for election. Implicitly, the sets of elected and excluded candidates are both empty. The bag of nontransferable ballots is empty. We assert that every ballot in *BallotBag* is assigned to its first-preference candidate in *VoteMass*. Furthermore, this predicate asserts that all such ballots have weight 1; this effectively “initialises” the *value* components of ballots in *voteMass* as required. The initial state requires that the total count equals the sum of first preference votes. The first stage count is set up in *stages'*.

<i>InitSystem</i>	
<i>System'</i>	$ \begin{aligned} & \text{continuing}' = CAND \\ & \text{nonTransferables}' = \llbracket \rrbracket \\ & \forall b : Ballot \mid b \text{ in } BallotBag \bullet \\ & \quad \exists_1 c : CAND \bullet \\ & \quad \quad b \text{ in } voteMass' \ c \wedge \\ & \quad \quad b.preference \ c = 1 \wedge \\ & \quad \quad b.value = 1 \\ & \text{totalCount}' = \text{sum}\{c : CAND \bullet \text{bagValue}(\text{voteMass}' \ c)\} \\ & \text{stages}' = \langle \text{stageCount } voteMass' \rangle \end{aligned} $

5 Specification: Operations

Before proceeding to define the operations required by the counting algorithm, further support functions are required. For clarity, placeholders for function arguments in the formal definition will be quoted in the narrative.

Function *nextPref*, given a preferences assignment *pref* from a ballot, a current candidate *c* and a set of candidates *ccs* still continuing for election, returns the next preferred candidate, if one exists. A nontransferable vote at any counting stage can be characterised as one for which *nextPref* is undefined. A vote is transferable if a unique, subsequent preference (in consecutive numerical order) is available for a continuing candidate. Elected or excluded candidates are not considered; such candidates will not appear in *ccs*.

<i>nextPref</i> : (CAND $\rightarrow$ $\mathbb{N}_1$) $\times$ CAND $\times$ $\mathbb{F}_1$ CAND $\rightarrow$ CAND
$ \begin{aligned} & \forall pref : CAND \rightarrow \mathbb{N}_1; \ c : CAND; \ ccs : \mathbb{F}_1 \text{ CAND} \mid c \notin ccs \bullet \\ & \quad \text{nextPref}(pref, c, ccs) = (\mu \ c' \mid \\ & \quad \quad (\exists \ c' : CAND \mid c' \in ccs \setminus \{c\} \wedge c, c' \in \text{dom } pref \wedge pref \ c' = pref \ c + 1)) \end{aligned} $

Function *valTransBallots* gives the cumulative value of transferable ballots held by the current candidate. Arguments are the *VoteMass* *v*, the current candidate *c* and the set of continuing candidates *ccs*. From the candidate's ballot bag, the subbag of those ballots in *VoteMass* for which a next preference is defined is obtained, and its cumulative bag value returned.

<i>valTransBallots</i> : <i>VoteMass</i> $\times$ CAND $\times$ $\mathbb{F}_1$ CAND $\rightarrow$ $\mathbb{R}$
$ \begin{aligned} & \forall v : VoteMass; \ c : CAND; \ ccs : \mathbb{F}_1 \text{ CAND} \mid c \notin ccs \bullet \\ & \quad \text{valTransBallots}(v, c, ccs) = \\ & \quad \quad \text{bagValue}(\{b : Ballot \mid (b.preference, c, ccs) \in \text{dom } \text{nextPref}\} \triangleleft (v.voteMass \ c)) \end{aligned} $

Function *transValue* gives the transfer value of an elected, over-quota candidate's ballots. The transfer value will be multiplied into each such ballot's value on transfer. This is the proportion by which the candidate exceeds quota, i.e. the current vote value less quota, all divided by the total value of the transferable ballots the candidate holds (e3). If the value of transferable votes does not exceed the surplus, then the transfer value is set to 1 (e4). Arguments are the *VoteMass*, the over-quota candidate *c* just elected, and the set of continuing candidates *ccs*.

$$\begin{array}{l}
 \text{transValue} : \text{VoteMass} \times \text{CAND} \times \mathbb{F}_1 \text{CAND} \rightarrow \mathbb{R} \\
 \text{(let } \text{bagVal} \hat{=} \text{bagValue}(v.\text{voteMass } c); \\
 \quad \text{valTrans} \hat{=} \text{valTransBallots}(v.\text{voteMass}, c, \text{ccs}) \bullet \\
 \forall v : \text{VoteMass}; c : \text{CAND}; \text{ccs} : \mathbb{F}_1 \text{CAND} \mid c \notin \text{ccs} \wedge \text{bagVal} > \text{quota} \bullet \\
 \quad \text{if } \text{valTrans} > \text{bagVal} - \text{quota} \\
 \quad \text{then } \text{transValue}(v, c, \text{ccs}) = (\text{bagVal} - \text{quota}) / \text{valTrans} \\
 \quad \text{else } \text{transValue}(v, c, \text{ccs}) = 1 \\
 \text{)}
 \end{array}$$

Function *multValue* multiplies a ballot's *value* component by the given input value. This will be used in ballot transfer.

$$\begin{array}{l}
 \text{multValue} : \text{Ballot} \times \mathbb{R} \rightarrow \text{Ballot} \\
 \forall b : \text{Ballot}; r : \mathbb{R} \mid 0 \leq r \leq 1 \bullet \\
 \quad \text{multValue}(b, r).\text{preference} = b.\text{preference} \\
 \quad \text{multValue}(b, r).\text{value} = b.\text{value} * r
 \end{array}$$

We can now proceed to specification of operational schema. We proceed from an initial state where all candidates have been allocated their first-preference ballots. Algorithm steps 1 and 2 are specified in schema *ElectAllandTransfer*. The tricky part of this operation is the transfer of ballots, which is defined separately as function *transfer*.

transfer takes three arguments: current *VoteMass*, candidates now exceeding quota who are to be elected and whose ballots are to be transferred *tcs*, and continuing candidates *ccs*. An updated *VoteMass* is returned. The relation is a function because every ballot in the input *VoteMass* has a determined destination in the output *VoteMass*. Note that transfer is only defined for over-quota candidates.

Firstly, all ballots apart from those held by transferring candidates explicitly remain “where they are”, i.e. they remain in the same after-state *VoteMass* bag as in the before-state. Such ballots have either already been marked as nontransferable, or are held by candidates not transferring at this stage (i.e. continuing candidates, previously elected candidates, and candidates elected exactly on-quota at the current stage) Secondly, each transferring candidates' bag (in the after-state) is emptied. Finally, transferring candidates' ballots are transferred, either to next-preference candidates or to the nontransferable bag (*e5*). In both cases transfer is subject to adjustment of the ballot weights (*value*) by multiplying by the appropriate transfer weight given by *transValue*.

```

transfer : VoteMass  $\times$   $\mathbb{F}_1$  CAND  $\times$   $\mathbb{F}_1$  CAND  $\rightarrow$  VoteMass

(let bagVal  $\hat{=}$  bagValue(v.voteMass c);
  nextPr  $\hat{=}$  nextPref(b.preference, c, ccs);
  multVal  $\hat{=}$  multValue(b, transValue(v.voteMass, c, ccs));
  transBallotBag  $\hat{=}$  bagRange(tcs  $\triangleleft$  v.voteMass);
  v'  $\hat{=}$  transfer(v, tcs, ccs) •
 $\forall v : \text{VoteMass}; tcs, ccs : \mathbb{F}_1 \text{ CAND}; c : \text{CAND} \mid$ 
  tcs  $\cap$  ccs =  $\emptyset \wedge c \in tcs \wedge \text{bagVal} > \text{quota}$  •
   $\forall b : \text{Ballot} \mid b \text{ in } ((\text{dom transBallotBag}) \triangleleft \text{BallotBag})$  •
    (b in v.nonTransferables  $\wedge$ 
      count v'.nonTransferables b = count v.nonTransferables b)  $\vee$ 
    ( $\exists cc : \text{CAND} \setminus tcs$  • b in v.voteMass cc  $\wedge$ 
      count (v'.voteMass cc) b = count (v.voteMass cc) b
    v'.voteMass c = []  $\wedge$ 
     $\forall b : \text{Ballot} \mid b \text{ in v.voteMass c}$  •
      if (b.preference, ccs, c)  $\in$  dom nextPref
      then count (v'.voteMass nextPr) multVal =
        count (v.voteMass c) b
      else count v'.nonTransferables multVal =
        count (v.voteMass c) b
  )

```

Operation schema *ElectAllandTransfer* now completes the algorithm steps 1 and 2: all candidates achieving quota are elected and removed from the list of continuing candidates; their ballots are transferred, and the list of excluded candidates is not changed (it is still empty). The last predicate specifies the update of *stages* with the count for this stage. The initial constraint on number of nonexcluded candidates is redundant at the first stage of counting but not during later stages.

Note that only over-quota candidates' (*overqcs*) ballots are transferred; any on-quota candidates (*onqcs*) retain their ballots and are elected.

```

ElectAllandTransfer
 $\Delta$ System
(
  let bagVal  $\hat{=}$  bagValue(voteMass c);
  overqcs = {c : CAND  $\mid c \in \text{CAND} \wedge \text{bagVal} > \text{quota}$ };
  onqcs = {c : CAND  $\mid c \in \text{CAND} \wedge \text{bagVal} = \text{quota}$ } •
  #elected + #continuing > noSeats
  excluded' = excluded
  elected' = elected  $\cup$  overqcs  $\cup$  onqcs
  continuing' = continuing  $\setminus$  overqcs  $\setminus$  onqcs
  if #overqcs = 0
  then voteMass' = voteMass
  else voteMass' = transfer(voteMass, overqcs, continuing')
)
stages' = stages  $\cap$  (stageCount voteMass')

```

We define an operation *Terminate* for algorithm step 3, to test whether election of candidates has been or can now be completed. This is a disjunction of *AllSeatsTaken*, where all seats have been won by elected candidates, and *AvailSuffice*, which elects all remaining continuing candidates where there are as many of them as seats left. Although such operation schemas would usually contain an output component to report the outcome, we omit this, and will explain the reason presently.

The Single Transferable Voting System: Functional Decomposition in Formal Specification

$AllSeatsTaken$
$\Xi System$
$\#elected = noSeats$

$AvailSuffice$
$\Delta System$
$\#elected + \#continuing = noSeats$
$elected' = elected \cup continuing$
$continuing' = \emptyset$
$excluded' = excluded$
$nonTransferables' = nonTransferables$
$voteMass' = voteMass$

$$Terminate \hat{=} AllSeatsTaken \vee AvailSuffice$$

Step 4 of the algorithm is specified by operation *ElectAllandTransfer* above.

We require further support functions in order to specify the exclusion of a candidate. Function *candScores*, given a stage *st* and set of candidates *cs*, maps the scores at that stage for those candidates to the score counts.

$candScores : (Stage \times \mathbb{F}_1 CAND) \rightarrow Score \rightarrow \mathbb{R}$
$\forall st : Stage, cs : \mathbb{F}_1 CAND \bullet candScores(st, cs) =$ $\{s : Score \mid s \in \text{ran } st.scores \wedge s.cand \in cs \bullet s \mapsto s.count\}$

We now require a function *minStages* which, given a sequence of stages *sseq* and a set of candidates *cs*, returns the set of numbers of those stages having a unique minimum-scoring candidate in the input set *cs*. This is done by selecting those stage numbers *i* for which *candScores* (*sseq* *i*, *cs*), restricted to the minimum of its range, has only one element. The minimum of the range is of course the score count of the lowest scoring candidate(s) in *cs* at stage *i*.

$minStages : (\text{seq } Stage \times \mathbb{F}_1 CAND) \rightarrow \mathbb{F} \mathbb{N}_1$
$\forall sseq : \text{seq } Stage, cs : \mathbb{F}_1 CAND \bullet$ $minStages(sseq, cs) = \{i \mid i \in 1 \dots \#sseq \wedge$ $\#(candScores(sseq\ i, cs) \triangleright \min(\text{ran}(candScores(sseq\ i, cs)))) = 1\}$

Finally: function *exclCand*, given the sequence of counting stages *sseq*, and the set of continuing candidates *cs*, returns the candidate to exclude. We establish the set of stage numbers *st* with unique minimum scoring candidate, and for the earliest such stage number *min st*, the map of scores to score counts *sc*. The inverse relational image (under map *sc*) of the minimum score count returns the singleton set of the score required. *select*, a generic function to “de-bracket” a singleton set, helps to extract the required candidate from this score.

$[X]$
$select : \mathbb{F}_1 X \twoheadrightarrow X$
$\text{dom } select = \{z : \mathbb{F}_1 X \mid \#z = 1\}$
$\forall x : X \bullet select\{x\} = x$

$$\begin{array}{|l}
 \text{exclCand} : (\text{seq Stage} \times \mathbb{F}_1 \text{ CAND}) \rightarrow \text{CAND} \\
 \hline
 (\text{let } st \hat{=} \text{minStages}(sseq, cs); \\
 \quad sc \hat{=} \text{candScores}(sseq(\text{min } st), cs) \bullet \\
 \forall (sseq, cs) : (\text{seq Stage} \times \mathbb{F}_1 \text{ CAND}) \mid st \neq \emptyset \bullet \\
 \quad \text{exclCand}(sseq, cs) = (\text{select}(sc^{-1} \mid \text{min}(\text{ran } sc) \mid)).\text{cand} \\
)
 \end{array}$$

Step 5 of the algorithm (operation *ExcludeandTransfer*) is invoked when ballot transfer yields no new continuing candidate achieving quota. This candidate is excluded, and her votes are transferred at full weight. In order to do this, a slightly modified version *xtransfer* of the *transfer* function must be specified. The only differences are that there is no constraint on the transferring candidates' ballot bag value, and that ballot transfer is at full weight. An elegant abstraction of the two functions has not presented itself!

$$\begin{array}{|l}
 \text{xtransfer} : \text{VoteMass} \times \mathbb{F}_1 \text{ CAND} \times \mathbb{F}_1 \text{ CAND} \rightarrow \text{VoteMass} \\
 \hline
 (\text{let } \text{bagVal} \hat{=} \text{bagValue}(v.\text{voteMass } c); \\
 \quad \text{nextPr} \hat{=} \text{nextPref}(b.\text{preference}, c, \text{ccs}); \\
 \quad \text{transBallotBag} \hat{=} \text{bagRange}(\text{tcs} \triangleleft v.\text{voteMass}); \\
 \quad v' \hat{=} \text{xtransfer}(v, \text{tcs}, \text{ccs}) \bullet \\
 \forall v : \text{VoteMass}; \text{tcs}, \text{ccs} : \mathbb{F}_1 \text{ CAND}; c : \text{CAND} \mid \\
 \text{tcs} \cap \text{ccs} = \emptyset \wedge c \in \text{tcs} \bullet \\
 \quad \forall b : \text{Ballot} \mid b \text{ in } ((\text{dom transBallotBag}) \triangleleft \text{BallotBag}) \bullet \\
 \quad \quad (b \text{ in } v.\text{nonTransferables} \wedge \\
 \quad \quad \quad \text{count } v'.\text{nonTransferables } b = \text{count } v.\text{nonTransferables } b) \vee \\
 \quad \quad (\exists cc : \text{CAND} \setminus \text{tcs} \bullet b \text{ in } v.\text{voteMass } cc \wedge \\
 \quad \quad \quad \text{count } (v'.\text{voteMass } cc) b = \text{count } (v.\text{voteMass } cc) b \\
 \quad \quad v'.\text{voteMass } c = [] \wedge \\
 \quad \quad \forall b : \text{Ballot} \mid b \text{ in } v.\text{voteMass } c \bullet \\
 \quad \quad \quad \text{if } (b.\text{preference}, \text{ccs}, c) \in \text{dom nextPref} \\
 \quad \quad \quad \text{then count } (v'.\text{voteMass nextPr}) b = \\
 \quad \quad \quad \quad \text{count } (v.\text{voteMass } c) b \\
 \quad \quad \quad \text{else count } v'.\text{nonTransferables } b = \\
 \quad \quad \quad \quad \text{count } (v.\text{voteMass } c) b \\
)
 \end{array}$$

If a uniquely lowest-scoring continuing candidate *c* exists at this stage, that candidate is excluded, and her ballots are transferred to next preferences at full weight. If not, there will be a set of at least two candidates *mincs* with equal minimum vote count. *mincs* is characterised by two predicate components:

- each candidate in *mincs* has a lower vote value than every continuing candidate not in *mincs*, and
- every element of *mincs* has the same vote value

A form of “backtracking” is used to identify, from this group of lowest-scoring candidates, the one with the uniquely lowest vote value at the earliest counting stage: function *minStages* will return a nonempty set of numbers of those stages with unique minimum-scoring candidates. If this process fails to yield a single candidate (*minStages* returns an empty set), then a candidate is selected at random from that set.

ExcludeandTransfer

$\Delta System$

```
(let bagVal  $\hat{=}$  bagValue (voteMass c);
    nextPr  $\hat{=}$  nextPref (b.preference, c, continuing') •
    #elected + #continuing > noSeats
     $\exists c : CAND \mid c \in continuing$  •
    if  $\forall c' : CAND \mid c' \in continuing \wedge c' \neq c$  •
        bagVal < bagValue (voteMass c')
    then true
    else  $\exists_1 mincs : \mathbb{F}_1 CAND \bullet \forall minc, c' : CAND \mid$ 
        minc  $\in mincs \subseteq continuing \wedge \#mincs > 1 \wedge c' \in continuing \setminus mincs$  •
        bagValue(voteMass minc) < bagValue(voteMass c')  $\wedge$ 
         $\#\{mc : CAND \mid mc \in mincs \bullet bagValue(voteMass mc)\} = 1 \wedge$ 
        if minStages(stages, mincs) =  $\emptyset$ 
        then  $\exists sc : CAND \mid sc \in mincs \wedge c = sc$ 
        else c = exclCand(stages, mincs)
    elected' = elected
    continuing' = continuing  $\setminus \{c\}$ 
    excluded' = excluded  $\cup \{c\}$ 
    voteMass' = xtransfer(voteMass, {c}, continuing')
)
stages' = stages  $\cap \langle stageCount voteMass' \rangle$ 
```

6 Putting it together

The full algorithm can now be specified in terms of its component operations using the Z schema calculus. Steps 1 and 2 (setup and first-preference election) are performed by the initial state and *ElectAllandTransfer*. The algorithm is then an infinite iteration round (in order of priority expressed by schema override) the step 3 termination test *Terminate*, step 4 *ElectAllandTransfer*, and step 5 exclusion *ExcludeandTransfer*.

$$STVAlgorithm == ElectAllandTransfer^{\circ} \\ (ExcludeandTransfer \oplus ElectAllandTransfer \oplus Terminate)^+$$

Of course, “schema override” here is precisely the relational override operator. We present the equivalent, schema-typed definition of this operator, along with a definition of transitive closure:

$$A \oplus B \hat{=} (\neg \text{pre } B \wedge A) \vee B$$

$$R^+ \hat{=} \bigcup \{n : \mathbb{N} \mid n \geq 1 \bullet R^n\}$$

The reason for omitting reporting (output) variables from the operations is now apparent. Although outputs from component operations are not a problem for constructing a new schema by overriding, the transitive closure operator is only defined for homogeneous relations. The operator is applied to an operation which is a homogeneous relation on *System*.

This is a succinct mechanism for describing the iteration of the algorithm. Termination (election of candidates to all vacant seats) results in the predominance of schema *AllSeatsTaken* within *Terminate*; this contains a $\Xi System$ component, denoting no subsequent state change. The only limitation of this model is the absence of an explicit output indicator of termination.

7 Overview

The following schematic of the specification shows the state schema, hierarchically decomposed through four levels of components. Each schema is associated with functions operating principally on that schema type. We distinguish between ‘read’ and ‘update’ functions (annotated ‘r’ and ‘u’): an ‘update’ function returns a value of the same type as an input argument, which will be used to update some variable in a state schema. Most of the functions are ‘read’, from various variables in the state. Apart from *multValue*, which updates a ballot’s *value* variable, the only update functions are *transfer* and its clone *xtransfer*, which perform the ballot transfer at a given election/ exclusion stage. ‘Local’ functions, i.e. ones not reading state variables elsewhere in the diagram, are annotated ‘*’. The diagram does not include operation schemas, all of which are defined at the global (*System*) level.

Decomposition of the specification

The levels can be described as follows:

- Level 4: basic low-level components and support functions. Functions are local and read, except for *multValue* and *nextPref*.
- Level 3: the first level of aggregation: i.e. ballot-bag, indexed collection of ballot-bags, sequence of scores. The three functions at this level are local and read, and provide information on bags.
- Level 2 comprises the major system components: sets of candidates, sequence of counting stages, the *VoteMass*. Functions are in two groups: those for *VoteMass* and those for *seq Stage*. Most of the functional coupling into other state variables is seen at this level. For the four *VoteMass* functions concerned with ballot transfer (similarly to *nextPref* at level 4), coupling is by reading current candidate and set of continuing candidates. Similarly, the *seq Stage* functions, which are concerned with finding a candidate for exclusion, couple by reading (subsets of) the set of continuing candidates.
- Level 1 is the global system level.

The overview demonstrates, for the most part, suitable levels of abstraction, encapsulation and separation of concerns. The unique aspect of this application ‘situation’ is, for each counting stage, the simultaneous update of (ballot transfer between) each of a candidate-indexed collection of ballot-bags. The Production

Cell case study does not cover this situation. The indexed nature of this state component suggests some form of promotion after the style of [11]. Recall that, for local and global state schemas *Local* and *Global*, a local operation schema *LocalOperation*, and a promotion schema *Promote* containing before- and after-state copies of both local and global variables, the promoted local operation is defined:

$$\exists \Delta Local \bullet Promote \wedge LocalOperation$$

The local operation could be the transfer of one ballot from the current to the next-preference candidate. This suggests three local parameters, with the next-preference candidate a function of the ballot selected. The full operation for one counting stage involves the local operation repeated for all the current candidate's ballots, over all candidates elected at this stage. These modelling requirements add considerable complexity to the specification of the promotion. The complexity of the global constraints to ballot transfer expressed in the functions at level 2 is a further obstacle. Apart from such considerations, why take the trouble to promote when such a local transfer would never be invoked on its own?

8 Conclusion

A formal specification of the STV algorithm has been presented which exemplifies a 'functional decomposition' style. This modelling 'situation' has been shown to be outside those discussed in the literature reviewed. The style is intuitively appealing because of the familiarity of the approach from software engineering tradition, and also avoids the complexity of promotion.

This work arose from the realisation by the UCT SRC that "first-past-the-post" was not an equitable form of representative election. A traditionally developed computer program for performing the STV count is in place, so the formal specification activity is "post-hoc". The work is relevant in the wider context of national constitution-making which is in progress in South Africa at present. This presentation demonstrates, by reference to errors in the original UCT specification, that construction of a formal model of a system, using formal specification, is an important quality validation technique.

The next stage of this work is to define test cases for the UCT program, and to justify redevelopment on the basis of confirming the presence of (at least) the errors identified here. The specification can subsequently be used for thorough test case generation and execution.

A necessary project is the generation of proof obligations to further validate the specification. Precondition calculation is unlikely to be straightforward. Experience of real elections has shown parties' proportion of first preference votes differing by more than a few percentage points from the final proportion of seats gained. This raises the question of how big such anomalies could become under STV, and it should be possible to formalise such a question, for formal reasoning and proof. This might answer an important open question, as well as casting some light on the tractability of reasoning in specifications such as this.

A useful exercise would be a comparison of this work with the specification given in VDM by Mukherjee & Wichmann [6, 5]. Further work is possible on specification of other systems of election. It may be that a reusable set of generic definitions emerges in an elegant fashion.

References

- [1] DAWES, J. *The VDM-SL Reference Guide*. UCL Press/ Pitman Publishing, London, 1991.
- [2] HAYES, I. A Generalisation of Bags in Z. In *Z User Workshop, Oxford 1989* (1989), J. Nicholls, Ed., Springer-Verlag.

The Single Transferable Voting System: Functional Decomposition in Formal Specification

- [3] LEWERENTZ, C., AND LINDNER, T. *Case Study Production Cell: A Comparative Study in Formal Software Development*. Lecture Notes in Computer Science. Springer-Verlag, 1994.
- [4] MACDONALD, A., AND CARRINGTON, D. Structuring Z Specifications: Some Choices. In *ZUM'95: The Z Formal Specification Notation* (1995), J. Bowen and M. Hinchey, Eds., Springer-Verlag.
- [5] MUKHERJEE, P., AND WICHMANN, B. *STV: A Case Study in VDM*. National Physical Laboratory, Teddington, UK., 1993.
- [6] MUKHERJEE, P., AND WICHMANN, B. *Formal Specification of the STV Algorithm*. Prentice-Hall, 1995.
- [7] NEWLAND, R., AND BRITTON, F. *How to conduct an election by the Single Transferable Vote*. The Electoral Reform Society of Great Britain and Ireland, 1976.
- [8] OF CAPE TOWN STUDENTS' REPRESENTATIVE COUNCIL, U. *The Single Transferable Vote*. 1994.
- [9] SPIVEY, J. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
- [10] VALENTINE, S. Putting Numbers into the Mathematical Toolkit. In *Z User Workshop, London 1992* (1992), J. Bowen and J. Nicholls, Eds., Springer-Verlag.
- [11] WOODCOCK, J., AND DAVIES, J. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, 1996.