University of Southampton

Towards the Development of an Electoral Count System using Formal Methods

Máiréad Meagher

A thesis submitted in fulfilment for the degree of

Master of Philosophy

in the

Faculty of Engineering and Applied Science

Department of Electronics & Computer Science

November, 2001

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS & COMPUTER SCIENCE

Master of Philosophy

TOWARDS THE DEVELOPMENT OF AN ELECTORAL COUNT SYSTEM USING FORMAL METHODS

by Máiréad Meagher

Techniques which use mathematical principles to develop computer systems are collectively known as Formal Methods. Formal Methods are usually applied to computer systems when correctness and soundness are primarily important.

A system to count votes is an example of such a system. This work includes the specification of, and the full development of part of, such an electoral system. When developing the system, a number of interesting issues arose, the examination of which became a significant part of this work.

The development of a system using formal methods entails taking a specification, written using mathematics and, moving, step by step, towards eventual implementation. We call these steps refinement steps. There are two main kinds of refinement - data refinement where we move from using abstract data in our descriptions to using more concrete data and algorithmic refinement where we introduce programming–like constructs. The traditional strategy is to proceed with data refinement and then with algorithmic refinement. In this thesis a strategy of mixing these approaches is examined, e.g. applying algorithmic refinement first. This strategy is found to be useful and to result in elegant solutions.

A fundamental tenet of refinement is that at each point in the developmental cycle (including the starting specification and the eventual implementation), the user should be unaware of any 'behind the scenes' activity. This means that the interface to the user should not change. However, it may happen that part of the specification is written in terms of parameterised abstract data. Then data refinement will change the interface. This issue is examined in this work and a workaround is provided for checking the correctness of this tricky refinement step.

Two paths of development are used in the work. The first is that of using Z and Morgan's Refinement Calculus. The B Method is then used for the main part of the thesis. The specification of the systems are written in Z and B. The development of parts of each are found in the main body of the text.

Contents

1 Introduction **12**

1.1 Brief History of Project 12
1.2 Related Work 13
1.3 Weakest Precondition 14
1.4 Refinement 15
1.4.1 Introduction to Refinement 15
1.4.2 Data Refinement 16
1.5 Introduction to B 17
1.6 Introduction to Z 20
1.6.1 The Z Specification Notation 20
1.6.2 Refinement of Z Specifications 21
1.7 Refinement Calculus and The B Method 22
1.7.1 Morgan's Refinement Calculus 22
1.7.2 Refinement in The B Method 23
1.7.3 Differences Between the Two Techniques 24
1.8 Dot Notation 25

2 The Development of a Z Specification **26**

2.1 Introduction 26
2.2 Pre–processing of Votes 27
2.2.1 Z Specification of Pre–processing 28
2.2.2 Approach Taken to Development 29
2.3 Refinement of make_ballot 30
2.3.1 Data Refinement under Functional Abstraction Invariant 30
2.3.2 From Z to Specification Statement 30
2.3.3 From Specification Statement to Code For MakeBallot 33
2.3.4 Code For Procedure MakeBallot 40
2.4 Refinement of insert 40

2.4.1 Supporting Definitions . . . 42
2.4.2 Definition of insert Using Functional Programing . . . 42
2.4.3 Calculational derivation of pointer algorithms from tree operations . . . 44
2.4.4 Producing Code from Functional Definition of insert . 45
2.5 Refinement of Pre-processing . . . 48
2.5.1 From Z to Specification Statement . . . 49
2.5.2 From Specification Statement to Code for pre-processing 50
2.6 Moving From the Specific to the Generic . . . 51
2.7 Conclusions . . . 52

3 Performing Algorithmic Refinement before Data Refinement in B 54
3.1 Introduction . . . 54
3.2 Laws of Distribution of Data Refinement . . . 55
3.3 Examples to Illustrate Laws . . . 56
3.3.1 Distribution Over Basic Assignment, Using **DatRef 1** 56
3.3.2 Distribution Over Generalised Assignment, Using **DatRef 2** . . . 56
3.3.3 Distribution Over Sequence, Using **DatRef 3** . . . 59
3.3.4 Distribution Over IF statement, Using **DatRef 4** . . . 59
3.3.5 Distribution Over the Introduction of a Local Variable using **DatRef 5** . . . 61
3.3.6 Distribution Over Loop Introduction, Using **DatRef 6** 62
3.4 Current Practice with Loop Introduction . . . 64
3.5 Conclusions . . . 64

4 Interface Refinement in B 67
4.1 Introduction . . . 67
4.2 Operations and Procedures . . . 67
4.3 Refinement of Procedures in B . . . 69
4.4 Examples of Data Refinement of Procedures . . . 70
4.4.1 Example 1 - Data Refinement of a Parameter . . . 70
4.4.2 Example 2 - Data Refinement of a Return Type . . . 71
4.4.3 Example 3 - Data Refinement of Both Return Type and Parameter . . . 71
4.5 Workaround . . . 72
4.5.1 Underlying Theory of Workaround . . . 73
4.5.2 Implementation of Workaround . . . 75
4.6 Conclusions . . . 76

5 Case Studies in B Development **78**
5.1 Introduction 78
5.2 Design Issues - B Method and C++'s S.T.L. 78
5.3 Description of System 80
5.4 Case Study 1 - Pre–Processing 83
5.4.1 Development of Make_Ballot 85
5.4.2 Development of Pre_Process 92
5.5 Case Study 2 - Setting up First Count 96
5.6 Conclusions 101

6 Conclusions **105**
6.1 Conclusions 105
6.2 Future Work 107

A Waterford Institute of Technology Academic Council Election Count Rules **110**
A.1 Election Procedures of Academic Members to the Academic Council 110
A.2 Rules for Academic Council Election (Academic Members) (Abridged Form) 111
A.3 Rule For Election or Exclusion (RuFEE) 112
A.3.1 Election 112
A.3.2 Exclusion 112
A.4 Rules for Academic Council Election (Academic Members) . 113

B Z Specification of an STV electoral system, specifically Waterford Institute of Technology's Academic Council election. **120**
B.1 Introduction 120
B.2 Global Declarations 121
B.3 Pre-Processing of Voting Papers 122
B.3.1 Z Specification of Pre–processing 123
B.4 Counting of Ballots 124
B.5 Count Operation 142

C Laws used in Refinement Calculus Example **143**

D B Specification of Academic Council count **145**
D.1 Introduction 145

D.2 B Specification 147
D.2.1 Overall Election 147
D.2.2 Global Variables 150
D.2.3 Bags - Paper and Ballot 152
D.2.4 Pre-Processing of Votes 157
D.2.5 Counting Functions 159
D.2.6 Ordering Functions 175
D.2.7 Candidate Balance Functions 183

Bibliography **185**

List of Figures

1.1 Machine M and associated Refinement N 18

2.1 Code for procedure MakeBallot 41
2.2 Definition of Insert 44
2.3 Generic Directed Update 45
2.4 Generic Directed Update After First Algorithmic Refinement 46

3.1 Data Refinement Laws 57
3.2 Basic Assignment 58
3.3 How to introduce loops early. This framework will allow early introduction of loops to be checked using the B-Toolkit. . . . 65

4.1 Simple Operation Refinement 68
4.2 Operation Refinement with parameter and return value . . . 68
4.3 Example of refinement of operation with parameter being data refined 71
4.4 Example of refinement of operation with return type being data refined 72
4.5 Example of refinement of operation with both return type and parameter being data refined 73
4.6 Structure of Machines for Workaround for Interface Refinement The refinement of operation $x \longleftarrow first(xseq)$ in MACHINE M1, by the operation $y \longleftarrow cfirst(yseq)$ in MACHINE M2 can be checked according to the framework above. 77

5.1 Mulitset of papers machine 81
5.2 Multiset of ballots machine 82
5.3 Mulitset of ballots machine... contd. 83
5.4 Abstract Specification of pre-processing 84
5.5 First Loop Introduction on *Make_Ballot* 87
5.6 *Make_Ballot* after the introduction of intermediate variable *bb*. 88

5.7 *Make_Ballot* after first data refinement 88
5.8 Calculating intermediate variable *bb*. 89
5.9 Data-refined loop for calculating *bb* 90
5.10 Algorithmically-refined pre-processing 93
5.11 Abstract Specification of *Setup_First_Count* 96
5.12 First Loop Introduction in *Setup_First_Count* 97
5.13 First Data Refinement in *Setup_First_Count* 99
5.14 Second Data Refinement in *Setup_First_Count* 100
5.15 ConcreteVoteMass Machine 102
5.16 ConcreteVoteMass Machine.. contd. 103
5.17 Implementation of *Setup_First_Count* 104

Acknowledgements

I would like to take this opportunity to thank my supervisor, Prof. Michael Butler. His generosity, patience, respect and help to me during this time has made working with him a profoundly developmental, rewarding and inspirational experience. He has been truly exceptional.

Thanks are due to the following from W.I.T.: to Paul Barry, Head of Department of P & Q, for his help in timetabling matters, Eric Martin, Head of School of Science, Tony McFeely, Returning Officer of the Academic Council Election Count, and all in Computer Services, for their help in technical matters. Thanks are also due to the Institute Management Committee for their help in facilitating study leave to pursue this study full-time for a short period.

I also acknowledge the help and hospitality I received from members of the DSSE group in the University of Southampton during my trips over there.

Thanks are due to Ib Sorenson of B–Core(UK) Ltd. for his help with the B–Toolkit.

I very much appreciate the help received during this time from Mícheál Ó Foghlú, Willie Donnelly, Eamon de Leastar, Michael Brennan, Jimmy McGibney, Richard Lacey, Shane Dempsey, and Gary McManus. Outrageous thanks are due to Kieran Murphy for his 'help' with LaTeX. (Or, roughly translated, for saving my life.)

My friends have, to my surprise and relief, stuck by me for the last four years. For that and for the various and innumerable kindnesses and support, I thank you all. A special thanks to (not in order of appearance!): Ann Clancy, P.J. Cregg, Claire Keary, Mary Keating, Mary Lyng, Ann Prendergast, Alice McDermott, Eamon Molloy, Laura Murphy, Louis Nevin, Tom O Toole, Ann Prendergast, Richard Vaughan and Cathy Walsh.

My family, as always, have given me support and understanding (and the occasional chance of a 'break' from work!) and many thanks are due to them.

To
Martin and Philly Meagher,
my parents.

"Mathematics, rightly viewed, possesses not only truth, but supreme beauty – a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show."

Bertrand Russell

Chapter 1

Introduction

This chapter introduces the reader to the main story of the project contained in this work. Some background information is needed for later chapters, some well-known areas and some conventions used throughout. The areas that are fundamental to this work but well known are briefly dealt with. The discussions are biased towards what we need for this work and as such do not serve as comprehensive guides to these areas. The references given provide such comprehensive coverage.

1.1 Brief History of Project

The plan for this work was based around writing the specification and developing the implementation of the counting system for the Waterford Institute of Technology's Academic Council Election of Academic Members using formal methods. The rules for this election are contained in Appendix A. The plan was to write the specification using Z [30] and develop the implementation using Morgan's Refinement Calculus [23].

The Z specification for this system is presented in Appendix B. An examination of the Z specification will show that the specification relies heavily on the axiomatic-definition style of function specification. The ultimate election count 'operation' (which is the only operation schema in the specification) is specified using calls on the axiomatically defined functions. This style was used as the usual state-based approach of completely separate operation schemas did not suit this type of system. As the development progressed, we found that the Z-specification through Morgan's refinement was not yielding neat solutions. This may have been due to the nature of the original problem (not naturally state-based). This part of the project is

discussed in more detail in Chapter 2.

Because of the inherent difficulties involved, it was decided to switch to the B Method. The B Method incorporates the entire suite of development stages. The re–writing of the Z specification using the B Method was not difficult. The translated specification using the B Method is contained in Appendix D.

When we started looking at the refinement process using the B Method, however, interesting challenges arose. Whereas theoretically it seemed possible and often very desirable to proceed with algorithmic refinement before data refinement, this was not directly supported in the tools available to us. This led to work on the examination of the feasibility and soundness of this seemingly unused approach. This work is reported on in Chapter 3 and in [9].

We adopted the approach of using stateless machines. This meant that the only conduit for data between machines was through operations' parameters and return values. As these operations were originally specified using abstract parameters and return values, the data refinement of these operations led to change of interface of operations. Refinement is interface–preserving in the B Method. This led to a examination of this area and the development of workarounds which is reported in Chapter 4.

We have used parts of the system specified using the B Method and developed them to implementation stage using standard B Method techniques and also the techniques developed during the course of this work. These case studies are presented in Chapter 5.

1.2 Related Work

We look at related work under two headings:

- Specification of voting systems
- Case studies involving data refinement early in the life-cycle.

Specification of voting systems

A voting system is a popular choice for case studies in specification as the rules are already well specified, tried and tested. Proportional Representation (PR) is a system where voters cast their vote 'in order of preference' for a list of candidates, usually in a multi-seat constituency. Single Transferable Vote (STV) is a particular (but usual) variant of PR[11]. Mukherjee

& Wichmann[25] present a full specification of a PR system using STV in VDM[18]. The specification is animated using SML[21] as part of the process of validating the specification. No post–specification development takes place.

Poppleton[26] uses the specification of an STV variant of a PR system, written using Z[30] as a case study to examine functional decomposition in Z. Again, no post–specification development takes place.

Case studies involving data refinement early in the life cycle

We have seen how the relative order of data refinement and algorithmic refinement is an important aspect of this work. This notion is not often discussed. It seems that if practitioners wish to use an order other than data refinement and then algorithmic refinement, (which is the only one directly supported by the B tools(i.e. the B–Toolkit and Atelier–B)) they use the workaround (described in Section 3.4), with the 'layered development' approach as described in Section 1.5.

Fraer presents a case study which looks at the classic Minimum Spanning Tree in [13]. In this case study, both the data structures used in implementation and the algorithm used on them are complex. The algorithm is introduced on the abstract data types to successfully simplify this step. It is implemented using 'layered development', incorporating the workaround approach.

In a case study of a Distributed Load Balancing System[31], Waldén proceeds with some algorithmic refinement before data refinement. The author concludes that the inability to introduce loops at anything but the final stage is restrictive. Again, this problem is solved within the available structures of the B-Toolkit.

In case studies on application of the B-Method to CICS[15], and Railway Signalling Systems[10], the authors use the 'layered design' approach.

In his paper on pointer implementation of tree structures[8], Butler introduces algorithmic refinement before data refinement. The B Method is not used. The resultant technique is used in Chapter 2 of this work.

1.3 Weakest Precondition

Much of the underlying theory used in this work is fundamentally based on the use of Dijkstra's *weakest precondition* [12]. We define what we mean by *weakest precondition* [14]:

Definition 1 *wp(S,R) is the set of all states such that execution of S begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying R.*

So, for example

$$wp(\text{`}x := x + 1\text{'},\ x < 1) \quad = \quad x < 0$$

We use the notion of *weakest precondition* to define semantics of commands, specifications and even refinement. For commands, for example, if for all postconditions we know which preconditions will guarantee termination satisfying the postcondition, then we say we know the *meaning* or *semantics* of the command.

For example, the semantics of assignment can be defined as follows:
For any postcondition $\mathcal{A}$

$$wp(w := E, \mathcal{A}) \ \widehat{=}\ \mathcal{A}[w \backslash E]$$

where the formula $\mathcal{A}[w \backslash E]$ can be obtained by replacing in $\mathcal{A}$ all occurrences of w by E.

Semantics of other programming constructs may be found in [23].

1.4 Refinement

In this section, we examine what we mean, intuitively as well as formally, by refinement. As much of the later work deals with data refinement, it is specifically discussed here.

1.4.1 Introduction to Refinement

We deal with systems being developed using Formal Methods. Such systems are originally specified using abstract data types. The eventual implementation will involve concrete data types and algorithms working on these. To get from the abstract specification to the concrete implementation, we repeatedly refine the previous program. Refining a program means making it less abstract whilst preserving the previous refined program's properties. A refined program is observationally indistinguishable from the previous program and is 'at least as good' for the customer. Each successive refinement should move the program towards executability. This is done by

- removing non-determinism

- introducing programming language-like structures (closer to executability)
- replacing abstract data types with concrete data types.

Back & Butler [4] describe refinement as a '...correctness-preserving transformation ... between (possibly abstract, non-executable) programs which is transitive, thus supporting stepwise refinement, and is monotonic with respect to program constructors, thus supporting piecewise refinement'.

We can categorize refinement into two main categories

- Algorithmic refinement
- Data Refinement

Data refinement involves introducing change into the type of data being worked on (introduces concrete data types). Algorithmic refinement involves introducing more concrete programming language-like structures to work on the data, leaving the structure of the data unchanged, e.g. introducing a loop.

More formally, using the theory of weakest pre-condition [12], the following definition of algorithmic refinement holds [23]:
For any commands S and T, we say that S is refined by T, writing $S \sqsubseteq T$, exactly when for all postconditions q we have

$$wp(S, q) \Rrightarrow wp(T, q).$$

From this definition, we can see that both a weakening of a pre-condition and a strengthening of post-condition or making a more non-deterministic transition are refinements. A Refinement Calculus based on [23] has been built around the different refinement rules. These rules show how, for example, to correctly introduce algorithmic structure to a specification statement.

1.4.2 Data Refinement

Using the weakest pre-condition theory, we can define what we mean by data refinement [8, 24]. Data refinement involves replacing abstract program variables with concrete program variables, preserving an abstraction relation between them. Let S be a statement with program variables u, a and let T be a statement with variables u, c (a represents the abstract variables that are replaced by the concrete variables c while u represents variables

that are common to both S and T). S is data refined under abstraction relation R, written $S \sqsubseteq_R T$, if the following holds: for all postconditions q not containing c, we have

$$R \wedge wp(S, q) \Rightarrow wp(T, \exists\, a \bullet R \wedge q) \tag{1.1}$$

We also make use of the *least data-refinement* of a statement [32]. Again, let S be a statement with program variables a, and let R be an abstraction relation relating a and c, then the least-refined statement on program variables c which is also a data refinement of S under R is denoted $\mathcal{D}_R^{a,c}[\![\ S\ ]\!]$. It is the least refined data refinement of S under R. So:

- $S \sqsubseteq_R \mathcal{D}_R^{a,c}[\![\ S\ ]\!]$ and
- $S \sqsubseteq_R T$ (T is a data refinement of S)

  $\Rightarrow \mathcal{D}_R^{a,c}[\![\ S\ ]\!] \sqsubseteq T$.

1.5 Introduction to B

The B Method [1] is a formal method which encompasses the entire lifecycle of the development of a system (theory and tool support). The specification, refinement and implementation phases of the development are represented by sets of Abstract Machines. A machine is an encapsulation of a state (determined by a set of variables) and set of operations. The notation used is Abstract Machine Notation (AMN).

AMN specifies state transitions using generalised substitutions. A *generalised substitution* is an abstract mathematical programming construct, built up from basic substitutions $x := e$ corresponding to assignments to state variables, via e.g. the following operators:

Operators on Generalised Substitutions	**AMN Syntax**
$P \mid S$	PRE P THEN S END
$@.S$	VAR S IN S END
$@v.(P ==> S)$	ANY v WHERE P THEN S
$S_1;\ S_2$	$S_1;\ S_2$ (sequence operator)
$S_1 \mid\mid S_2$	$S_1 \mid\mid S_2$ (parallel operator)
WHILE E DO S INVARIANT I VARIANT e END	WHILE E DO S INVARIANT I VARIANT e END

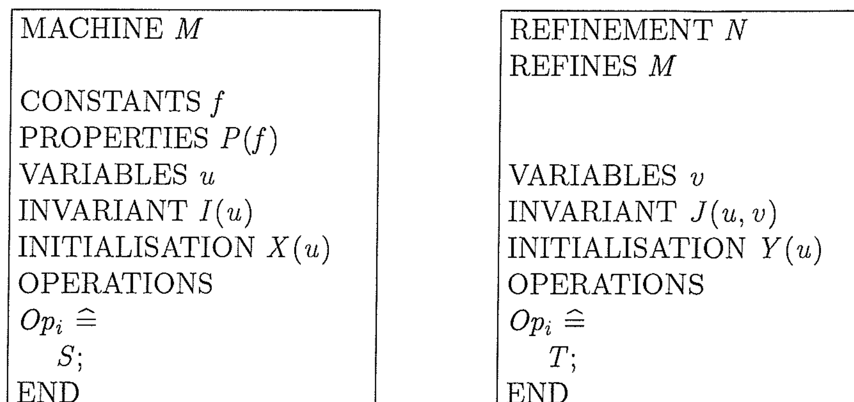


Figure 1.1: Machine M and associated Refinement N

Note that the sequence operator (S_1; S_2) is not currently allowed at the specification stage.

Each machine may have an associated REFINEMENT which contains refinements of the operations of the original machine. Each REFINEMENT may, in turn, have an associated REFINEMENT which contains further operation refinement. Eventually a REFINEMENT will have an associated IMPLEMENTATION. This concludes the development process.

The machine M of Fig. 1.1 introduces the (set of) state variables u. There may be a constant relationship between some of these variables. These relationship(s) are contained in the INVARIANT $I(u)$. Each variable must be initialised in the INITIALISATION. These initialisations must not, of course, violate the invariant. The OPERATIONS section contains the specification of the operations of the system, written using AMN.

In this work, we use the CONSTANTS .. PROPERTIES section extensively, due to the style of our specification. Firstly we declare the constants and then give them appropriate properties, e.g. type and further specification. It is used for mathematical functions. e.g.

CONSTANTS $\quad$ *double*
PROPERTIES $\quad$ $double \in \mathbb{N} \rightarrow \mathbb{N} \;\wedge$
$\forall n.(n \in \mathbb{N} \;\wedge\; double(n) = n * 2)$

Normally, as above, data or state is managed through the manipulation of state variables (u, v). It is, however, possible to have machines with no variables - we call them *stateless* machines. In this case, data is managed through the use of input parameters (*data in*) and return values (*data out*). This is the approach taken in this work.

The (set of) state variables v contained in REFINEMENT N in Fig. 1.1 will be linked formally with MACHINE M's set of variables u through the INVARIANT $J(u, v)$.

Operations are refined by introducing v instead of u (data refinement) and by introducing new algorithmic constructs e.g. sequence, alternation (algorithmic refinement). Note that operation refinement is, fundamentally, interface–preserving. This follows from the principle that the user (who has originally specified the operation) should be unaware of any 'behind–the–scenes' refinement activity. From their point of view, they specify an operation with an interface. They expect an implementation of exactly that operation. We have mentioned that we use stateless machines in this work and that this means that we depend on the parameters of and return values from operations to port data. The interfaces to these operations will be specified using abstract data types and the final operations will have interfaces containing data refined concrete data types. We need to have correct refinements between non-interface preserving refinements. This issue is discussed in Chapter 4.

We have discussed the meaning and use of weakest precondition in Section 1.3. When using the B Method, the previously described $wp(S, R)$ is written in the form $[S]R$.

Refinements must be correct. Using the B Method, when we introduce REFINEMENTs, resulting proof–obligations must be discharged to prove the correctness of the step. The full set of proof–obligations resulting from each introduction is contained in [1]. For instance, one of the main proof obligations for the refinement of operation S from Fig. 1.1 is

$$I \wedge J \wedge [S]true \Rightarrow [T]\neg\,[S]\neg\,J \qquad \textbf{Operation Refinement} \qquad (1.2)$$

The Operation Refinement proof–obligation states that for any concrete step of T there is some abstract step of S that establishes the retrieve relation. It is also called the *Sufficient Condition.*

Abrial [1] shows that (1.2) implies (1.1).

Using the B tools, the proof–obligations resulting from a refinement step can be generated. The more trivial of them can be automatically discharged by the tools.

A number of REFINEMENTs are usually necessary before we make the final step of introducing an IMPLEMENTATION. IMPLEMENTATION is a special case of a REFINEMENT, with a few further constraints. The main constraint is that no further refinements can take place. The other main

constraints are to do with independence from other implementations and ensuring that the implementation is concrete. Whereas the introduction of all algorithmic structures is allowed at REFINEMENT stage in the B Method, in practice, no current B tools will allow the introduction of loops before the implementation stage. This seeming anomaly leads us to the work of Chapter 3.

A set of Abstract Machines can be structured using classical techniques, e.g. top–down, by grouping different related parts of the system into different MACHINEs for instance, in the case of the specification. The entire system can be integrated by the use of the INCLUDES, USES, SEES, and IMPORTS clauses. This ability to share other MACHINEs (for example) means that general MACHINEs can be re–used. When using the B Tools, any changes to any of the machines in such a hierarchy will result in the need for the appropriate machines to be re-analysed. Also, when using the B Tools, it is possible to get an overview of the specification and design as presented in a layered horizontal manner [5].

Larger developments are structured using a technique called 'layered development'. The original specification is decomposed into a number of linked subsystem descriptions. The idea is that each subsystem can be refined separately into code, independent of the design choices made in implementing the other subsystems. A subsystem $SS1$ that makes use of the functionality of another subsystem $SS2$ only accesses the abstract *specification* of $SS2$ and not any of its refinements. Each if the separate refinement sequences are termed *subsystem developments.* They are the 'layers' in the 'layered system development'.

A full description of the B Method is contained in Abrial [1]. Other useful works are available in [20, 27, 34]

1.6 Introduction to Z

In this work, we are interested in the full formal development of a system. Z [30] is a specification notation. We need to use a separate technique to move towards implementation of the Z specification. In this section, we introduce the Z notation. The alternatives available on how to proceed to refinement are mentioned.

1.6.1 The Z Specification Notation

The formal specification notation, Z, is based on Zermelo-Fraenkel (hence Z) set theory and first-order predicate logic. It was initiated by J.R. Abrial

and developed by the Programming Research Group in Oxford since the 1970's. Z is a method of presenting mathematics in a readable framework. The main Z construct is the *schema* where we draw attention to some things of importance in the system and describe the relationships between elements of that system. For example, if we wish to model a (simple) stock control system, we have

$[STOCK]$ The set of all possible stock items

We have a system state

StockSystem

$level : STOCK \rightarrowtail\!\!\!\!\mapsto \mathbb{N}$
$carried : \mathbb{P} STOCK$

$carried = dom\ level$

The declarations of 'state' variables are above the line. The predicates or 'invariants' of the system are below the line. These describe the relationship between variables that must never be violated. We have operation schemas, which, though syntactically the same as system state schemas, describe the effect of an operation on (imported) systems as defined by their system states.

We may need mathematical functions. To specify these, we use *axiomatic definitions*. Once specified, these functions are globally accessible. For instance, if we wish to specify the *double* function which returns the double of any natural number, we would write it as follows:

$double : \mathbb{N} \rightarrow \mathbb{N}$

$\forall n : \mathbb{N} \bullet$
$\quad double(n) = n * 2$

Z is suited to state-based systems. These are systems whose behaviour can be described by the effect of operations on state. If the operations themselves are complex, then the use of mathematical functions, as specified using axiomatic definitions can simplify the operation for the reader. The Z specifications written in Chapter 2 and Appendix B rely heavily on the use of axiomatic definitions.

1.6.2 Refinement of Z Specifications

In his definitive book on Z, [30], Spivey gives an introduction to refinement. A more detailed description of a refinement process applicable to Z specifi-

cations is given by Woodcock & Davies in [33]. Morgan [23] shows how to calculate the program from the specification. His technique (which we refer to as Morgan's Refinement Calculus) is not Z specific but generally useful. King explores the differences between Z and the Refinement Calculus in [19]. In our work (specifically in Chapter 2), we use Morgan's Refinement Calculus to refine systems originally specified using the Z specification notation.

1.7 Refinement Calculus and The B Method

We use two techniques of refinement during this work. These are Morgan's Refinement Calculus and the B Method.

In this section, we briefly describe the two techniques, highlighting the differences between them. We re–visit the topic of their differences in Section 6.1 where we offer some judgements on the relative merits of each approach.

1.7.1 Morgan's Refinement Calculus

A full description of the Refinement Calculus is contained in [23]. What is presented here is a brief description of the technique, concentrating on the strategies involved in the entire life–cycle.

We treat everything as a program, some programs are executable. We repeat refining the program until we reach an executable program.

A program is specified in the form

$$w : [\ pre, post\ ]$$

where

- w is the set of variables whose values may change during program execution
- *pre* describes the initial state (*precondidion*)
- *post* describes the final state (*postcondition*)

The following is a strategy to (starting with a Z specification) choose a refinement path, and derive an implementation.

1. Rewrite the Z specification of the operation/function in terms of Morgan's '*program*' (pre, post).

2. Introduce more concrete data types. Link abstract and concrete data types using 'retrieve relations'. (This is the data refinement step).

3. Attempt to derive a best–guess at a possible, efficient implementation of the operation/function on the concrete data types. This is usually done by guessing a number of possible implementations, and then costing them using techniques as described in, e.g. [2]. Use this to guide the direction of the remainder of our refinement steps. (The advantage of this step is that we now have a structure to aim for and the implementation will be efficient.)

4. Restructure (e.g.break up) the program derived in step 2 to guide refinement (according to structure derived in step 3).

5. Work on separate '*programs*' until they are executable, using Morgan's Refinement Caclulus [23].

.

We apply the laws of Refinement Calculus, all of which are contained in [23] and some of which are re–written in Appendix C to the various programs. So, using sequential composition, we break down the program into a number of programs, using top–down design techniques. Each of these programs will be further refined using a separate refinement path by repeatedly applying the laws as 'appropriate'. Note the direction of the effort in this case.

1. Choose a law that you think is appropriate.

2. Attempt to prove the *proviso* of the rule.

3. If the *proviso* has been proven, then the refinement law is applicable and the resulting refinement step is correct.

1.7.2 Refinement in The B Method

We have mentioned that the B Method encompasses the entire lifecycle of system development. So, starting with a specification written using the B Method, we can eventually derive a correct implementation, all within the B Method. A typical strategy of a B development would be as follows:

1. Write the original machine which contains the specification, using AMN.
2. Guess at an implementation, in the same manner as in the previous section.
3. Proceed with refinement, repeatedly making the machine more concrete w.r.t. data and/or algorithmic structure.
4. Stop when the machine (now called IMPLEMENTATION) is implementable.

In this case, note the direction of effort. As part of each 'further refinement', the process is :

1. Write the refinement based on the previous available machine.
2. Attempt to discharge the resulting proof–obligations. If this is possible, then the refinement step is correct.

1.7.3 Differences Between the Two Techniques

Apart from the obvious syntactic differences, the main differences in approach are as follows:

- **Direction of Effort:** In the case of Z - Refinement Calculus, the user must 'guess' which law is applicable, prove the *proviso* and then, if successful, applying the law will result in a correct refinement step. In the case of the B Method, the further refinement is guessed, the resulting proof–obligations are mechanically derived. If these proof–obligations can be discharged, then the refinement is correct.
- **Step Sizes:** In the extreme case, using the B Method, we could proceed directly from specification to implementation. The resulting proof–obligations could be difficult, but should be possible if the step is a correct refinement. In practice, the steps are chosen to produce more easily discharged proof–obligations. In the case of Morgan's Refinement Calculus, laws are applied one (or in some cases two) at a time.
- **Readability:** In the case of the B Method, the entire 'program' is always together. (During the process, 'chunks' of code may be renamed as operations, but if the operation is appropriately named, the

overall view is still available in one place.) This makes the process clean and readable. When using Morgan's Refinement Calculus, however, the original program, which is usually broken up using sequential composition, takes many separate refinement paths.

1.8 Dot Notation

When using schemas in Z [30], we can access an element of an instance of that schema using the dot notation, e.g.

```
┌─ Coordinate ──────────────────
│ xpart : ℤ
│ ypart : ℤ
└───────────────────────────────
```

If we have *mypoint* : *Coordinate* then we can access *mypoint.xpart* or *mypoint.ypart.*

There is no corresponding notation in the B Method. Using the B Method, if we have *Bcoordinate* $== (\mathbb{Z} \times \mathbb{Z})$ and an instance *bpoint* : *Bcoordinate*.We need functions to return component parts of pairs, for elements of *bpoint.* We have (for instance) two functions. $f_1(bpoint)$ is the xpart of the co-ordinate and $f_2(bpoint)$ is the ypart of the co-ordinate. The definitions of these functions are usually obvious. We introduce (an overloaded) dot notation as syntactic sugar for these functions, so *bpoint.xpart* returns the xpart (or first component) of bpoint, the same as $f_1(bpoint)$ and *bpoint.ypart* returns the ypart (or second component) of *bpoint*, the same as $f_2(bpoint)$. It should be clear from the name of component part (e.g. *xpart*) which component is being accessed. If it is not obvious, it will be clearly stated. We use this syntactic sugar from now on.

Chapter 2

The Development of a Z Specification

2.1 Introduction

In this chapter, we start with a Z specification and formally develop an implementation using Morgan's Refinement Calculus [23] and techniques described in Butler [8]. We produce Pascal–like code.

The sub-system we examine in this chapter is part of the overall system, the counting of votes in an electoral count system. An implementation of a PR system, known as STV, is used [11]. This particular system is based largely on the rules of election for Seanad *É*ireann [28] and is tailored to count the votes polled to elect the academic elected membership of the Academic Council in Waterford Institute of Technology, Ireland. (Each academic member of staff can vote to elect 13 members from the academic staff). The main modification from the Seanad *É*ireann rules is the need for gender and school balance in the elected members cohort. The full set of rules as used are available in 'Academic Council Election Rules', Appendix A.

The input to this system is a collection of votes, which are then counted according to our set of rules. The main output of the system is the list of elected candidates. The raw input can contain errors, either accidentally or deliberately introduced by the voter. A decision was taken to specify the counting system based on validated votes (which we call ballots). We therefore need to specify (and implement) this preprocessing of the input. The development of this preprocessing is the subject of this chapter.

2.2 Pre–processing of Votes

At the abstract level, the input is modelled as a sequence of papers where a paper is a partial function from candidate to $\mathbb{N}$ and $bag\ T == T \rightarrowtail\!\!\!\!\!\!+ \mathbb{N}_1$, where for $b \in bag\ T$ and $item \in dom(b)$, $b(item)$ returns the number of occurrences of *item* in the bag. This raw input *paper* is processed to become what we term a *ballot*. The operation which deals with this has an abstract specification called *make_ballot*. The preprocessing of the sequence of papers returns a bag of ballots and has an abstract specification of *pre_process*. Not all votes will be valid, so some input will be discarded.

Valid preferences on a paper are that set of preferences that are unique, contiguous and start at one. Duplicate preferences (e.g., two candidates have preference 3 associated with them) are disregarded as are all higher preferences on the paper. A skip in preferences (e.g., the voter expresses preferences 1,2,4,5 but no 3) invalidates all preferences after the skipped preference (in this case, only 1,2 are valid). The ballot holds only valid preferences and uses an injective sequence such that the first element of the sequence is the candidate whose preference was 1, etc. (No other candidate will have been validly assigned preference 1). It may happen that a paper has no valid preferences (e.g., if two candidates are given preference 1), in which case the ballots sequence will be empty. This is termed a spoiled vote and is not added to the (resultant) bag of ballots.

In the specification, the input paper is modelled as a function $Cand \rightarrowtail\!\!\!\!\!\!+ \mathbb{N}$ so as to model the physical voting paper as closely as possible. The validated vote (which we call a *ballot*) is modelled as a sequence of *Candidate*s in order of the voter's preference. A weight is associated with each *ballot* for counting purposes. At the end of pre-processing the votes, the weight of each *ballot* is 1000. There may be duplicate *ballot*s. Obviously, each duplicate is important. What may not be so obvious is that each duplicate *ballot* is dealt with equivalently. It is thus appropriate to model the collection of *ballot*s as bags of *ballot*s at the end of preprocessing.

The refinement process results in an implementation that takes as input an array of voting papers. The output from the process is a binary search tree where each node contains a *ballot* and the number of times the *ballot* occurs. The binary search tree is used because it is an efficient method of grouping the duplicate ballots.

Our first step is to take the Z specification of the preprocessing of the votes and refine this into specification statements as described in Morgan [23]. Next, we manipulate that specification statement to break it into more manageable statements. Finally, we refine each of the statements using

appropriate techniques.

2.2.1 Z Specification of Pre–processing

We have the following type:

[*Candidate*] The set of all possible candidates for election.

We also have three global variables:

$no_cands : \mathbb{N}$ The number of candidates nominated for election (i.e. the number that appear on the voting paper),
$no_votes : \mathbb{N}$ The number of votes cast,
$no_voters : \mathbb{N}$ The number of eligible voters.

We use the following definitions of Ballot and Paper as follows:

Ballot

$$pref : \text{iseq } Candidate$$
$$value : \mathbb{Z}$$
$$-1000 \leq value \leq 1000$$

$$Paper == Candidate \twoheadrightarrow \mathbb{N}_1$$

Note that we are expecting non-zero preferences. Valid preferences on a voting paper start at 1 and are unique, increasing and contiguous. We specify a function which returns the first non-unique, non-contiguous or non-existent preference. This number minus 1 is the number of valid preferences on the voting paper. All preferences between 1 and this number are valid.

$$find_first_hole_or_dup : Paper \rightarrow \mathbb{N}$$
$$find_first_hole_or_dup(paper) = min\{n : \mathbb{N} \mid n : 1 \mathinner{..} no_cands + 1 \wedge \#paper^{\sim}(\!|\, \{n\} \,|\!) \neq 1\}$$

The next function takes the voting paper and returns a (valid) Ballot with invalid preferences stripped. This means that, for instance, a spoiled vote will have no valid preferences.

$$make_ballot : Paper \rightarrow Ballot$$
$$make_ballot(paper) = \lhd\, pref \leadsto 1 \mathinner{..} find_first_hole_or_dup(paper) - 1 \lhd paper^{\sim}, value \leadsto 1000 \,\rhd$$

The following function takes a sequence of voting papers and returns a sequence of Ballots.

$$makeseqBallots : \mathrm{seq}\, Paper \rightarrow \mathrm{seq}\, Ballot$$
$$makeseqBallots(seqpapers) = map\ make_ballot\ seqpapers$$

The following function throws away empty ballots. These are invalid papers (or spoiled votes) that were stripped down to empty ballots.

$$throwawayempties : \mathrm{seq}\, Ballot \rightarrow \mathrm{seq}\, Ballot$$
$$throwawayempties(fullseq) = fullseq \upharpoonright \{b : Ballot \mid b \in \mathrm{ran}\, fullseq \land \#(b.preference) > 0 \bullet b\}$$

The following function takes in the sequence of voting papers and produces a bag of preprocessed ballots. As a sequence is finite, then items returns a finite bag of Ballots. We call this type *finBagBallot*.

$$pre_process : \mathrm{seq}\, Paper \rightarrow finBagBallot$$
$$pre_process(seqpapers) = items(throwawayempties(makeseqBallots(seqpapers)))$$

Note that the following definition of map is assumed:

$$map : (X \rightarrow Y \times seq\ X) \rightarrow seq\ Y$$
$$map\ f\ s = \{n : 1 \mathinner{..} \#s \bullet n \mapsto f(s(n))\}$$

2.2.2 Approach Taken to Development

We examine the problem specification in three parts, using a bottom–up approach.

1. **make_ballot**. The production of a validated *ballot* from an unvalidated *paper*. This is dealt with in Section 2.3.

2. **insert**. Insertion of one *ballot* into the binary search tree. We examine this development in Section 2.4.

3. **pre_process**. This part takes the collection of unvalidated *paper*s, validates them (using **make_ballot**) and inserts them (using **insert**) into the collection of validated *ballot*s under certain conditions. We examine this development in Section 2.5

The development of parts 1 and 3 are relatively straightforward using standard techniques. However, in the case of 2, we find that because of the use of a concrete recursive data structure, i.e. binary search trees, it is more appropriate to appeal to functional programming techniques and specifically techniques developed in Butler [8]. These provide the basis for a mechanical approach to refining trees (as defined using recursive functions) to pointer implementations. We examine the three parts separately. We look at the development of part 1 carefully, as an example of standard Morgan's Refinement Calculus. We look at the development of part 2 showing how the Butler technique [8] works. We finally give an overview of the (standard Morgan-type) development of part 3.

The refinement steps which use Morgan's Refinement Calculus [23] reference laws which are included in Appendix C.

2.3 Refinement of make_ballot

2.3.1 Data Refinement under Functional Abstraction Invariant

In this section, we use a special type of data refinement, i.e. where the abstraction invariant is functional. We use the following from from [22]:

The following is always valid where a is the set of abstract variables, c is the set of concrete variables, x the set of common variables and AI is the abstraction invariant:

$$a, x : [pre,\ post] \sqsubseteq_{AI} \quad c, x : [(\exists\, a \bullet AI \wedge pre), (\exists\, a \bullet AI \wedge post)]$$

Given that AI is functional, this means we can write $AI \equiv a = f(c)$
$=$ 'Using the one point rule'
$a, x : [a = f(c) \wedge pre,\ a = f(c) \wedge post]$
$\sqsubseteq_{AI}$ 'If post does not contain any initial variables'
$c, x : [pre[a \backslash f(c)],\ post[a \backslash f(c)]]$
We use this law for data refinement where the abstraction invariant is functional. It is called **Law C9** and appears in Appendix C

2.3.2 From Z to Specification Statement

Supporting Definitions

We have used *Paper* and *Ballot* as the abstract models. We now begin to move towards a concrete representation and define two new types, *CPaper*

whose instances will be a concrete representation of *Paper* and *CBallot* whose instances will be a concrete representation of *Ballot*. Note that we use the programming-like structure of a record which we call *rec*.

```
Type  CPaper == array [1 .. no_cands] of
                rec
                    cand:Candidate;
                    ppref: ℕ ;
                end;

      CBallot == rec
                    bpref: array[1 .. no_cands] of Candidate;
                    value: ℕ;
                    size: ℕ;
                end;
```

Retrieve Relations

The following retrieve relations define the relationship between the abstract(ap) and concrete(cp) Paper and abstract(ab) and concrete(cb) Ballots. cp is an instance of type CPaper and cb is an instance of type CBallot.

$$abs_p \mathrel{\widehat=} ap \; = \; abs(cp) \; = \\ \{c \mapsto n \mid \exists\, i : 1 \,..\, no_cands \bullet cp[i].cand = c \wedge cp[i].ppref = n\}$$

$$abs_b \mathrel{\widehat=} ab = abs(cb) = \lhd \quad pref \leadsto \{n \mapsto cb.pref[n] \mid n \in 1 \,..\, cb.size\}, \\ value \leadsto cb.value \rhd$$

Calculation of a Specification Statement for make_ballot

We will look at writing a specification statement which refines the *make_ballot* function as specified.
So:

$$\begin{array}{ll} & ab : [ab = make_ballot(ap)] \\ \sqsubseteq_R & \text{‘\textbf{Law C9} data refinement (functional)’} \\ & cb : [abs(cb) = make_ballot(abs(cp))] \end{array}$$

where $R \equiv ap = abs_p(cp) \;\wedge\; ab = abs_b(cb)$
To get to the next step, we wish to strengthen the postcondition, using **Law C8**. If $next_step \Rightarrow abs(cb) = make_ballot(abs(cp))$ then
$cb : [abs(cb) = make_ballot(abs(cp))] \sqsubseteq cb : [next_step]$

The technique we use to find a specification statement is to find such a *next_step*. Look at L.H.S. of the equation $abs(cb) = make_ballot((abs(cp))$

$$
\begin{aligned}
L.H.S. &= \lhd \quad pref \leadsto \{n \mapsto cb.bpref[n] \mid n \in i \mathrel{..} cb.size\}, \\
&\qquad\quad value \leadsto cb.value \quad \rhd \\
&\quad \text{`concentrate on the pref binding '} \\
&= \{n \mapsto c \mid c = cb.bpref[n] \land n \in 1 \mathrel{..} cb.size\} \\
R.H.S. &= \lhd pref \leadsto 1 \mathrel{..} find_first_hole_or_dup(abs(cp) - 1 \lhd (abs(cp))^{\sim}, \\
&\qquad value \leadsto 1000 \rhd \\
&\quad \text{`Similarly, look at the pref binding '} \\
&= 1 \mathrel{..} find_first_hole_or_dup(abs(cp)) - 1 \lhd (abs(cp))^{\sim} \\
&= \text{`Rewrite find_first_hole_or_dup'} \\
&\quad 1 \mathrel{..} (min\{no : \mathbb{N} \mid \#\{abs(cp))^{\sim} (\!| \{no\} |\!) \neq 1\} - 1) \lhd (abs(cp))^{\sim} \\
&= \text{`Rewrite abs(cp)'} \\
&\quad 1 \mathrel{..} (min\{no : \mathbb{N} \mid \#\{c \mapsto n \mid \exists i : 1 \mathrel{..} no_cands \bullet \\
&\qquad\qquad cp[i] = c \land cp[i].ppref = n \land n > 0\}^{\sim} \\
&\qquad\qquad\quad (\!| \{no\} |\!) \neq 1\} - 1) \\
&\qquad \lhd \{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet \\
&\qquad\qquad cp[i].cand = c \land cp[i].ppref[i] = n \land n > 0\}^{\sim} \\
&= \text{`Let HOD} = min\{no : \mathbb{N} \mid \#\{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet \\
&\qquad\qquad\quad cp[i].cand = c \land cp[i].ppref = n \land n > 0\} \\
&\qquad\qquad\qquad (\!| \{no\} |\!) \neq 1\} - 1' \\
&\quad 1 \mathrel{..} HOD \lhd \{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet \\
&\qquad\qquad cp[i].cand = c \land cp[i].pref = n \land n > 0\} \\
&= \text{`From definition of } \lhd\text{'} \\
&\quad \{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet cp[i].cand = c \land cp[i].ppref = n \land \\
&\qquad\qquad n > 0 \land n \in 1 \mathrel{..} HOD\} \\
&= \text{`}n \in 1 \mathrel{..} HOD \land n > 0 \Rightarrow n \in 1 \mathrel{..} HOD\text{'} \\
&\quad \{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet cp[i].cand = c \land cp[i].ppref = n \land \\
&\qquad\qquad n \in 1 \mathrel{..} HOD\}
\end{aligned}
$$

The following implies the equality of L.H.S. and R.H.S.

$$
\begin{array}{l}
cb.size = HOD \land \\
cb.value = 1000 \land \\
\forall n : 1 \mathinner{\ldotp\ldotp} cb.size \bullet \exists i : 1 \mathinner{\ldotp\ldotp} no_cands \bullet \\
\qquad\qquad cp[i].cand = cb.bpref[n] \land cp[i].ppref = n
\end{array}
$$

This leads to the refinement
cb:[abs(cb) = make_ballot(abs(cp))]
$\sqsubseteq$ '**Law C8** strengthen postcondition'

$$
\begin{array}{l}
cb : [cb.size = \\
\quad min\{no : \mathbb{N} \mid \#\{n \mapsto c \mid \exists i : 1 \mathinner{\ldotp\ldotp} no_cands \bullet \\
\qquad\qquad cp[i].cand = c \land cp[i].ppref = n \land n > 0\}(\!|\ \{no\}\ |\!) \neq 1\} \\
\qquad\qquad -1 \land \\
\quad cb.value = 1000 \land \\
\quad \forall n : 1 \mathinner{\ldotp\ldotp} cb.size \bullet \exists i : 1 \mathinner{\ldotp\ldotp} no_cands \bullet \\
\qquad\qquad cp[i].cand = c \land cp[i].ppref = n \land c = cb.bpref[n] \quad]
\end{array}
$$

Note there is no precondition, only postcondition. We introduce a procedure to name this code, called MakeBallot. We will return to this later when all the code has been refined.
ab := make_ballot(ap) $\sqsubseteq$ MakeBallot(cb, cp)
where

$$
\begin{array}{l}
\textbf{procedure}\ MakeBallot(\textbf{ref}\ cb : Ballot, \textbf{val}\ cp : Paper) \widehat{=} \\
cb : [cb.size = \\
\quad min\{no : \mathbb{N} \mid \#\{n \mapsto c \mid \exists i : 1 \mathinner{\ldotp\ldotp} no_cands \bullet \\
\qquad\qquad cp[i].cand = c \land cp[i].ppref = n \land n > 0\}(\!|\ \{no\} \neq 1\} \\
\qquad\qquad -1 \land \\
\quad cb.value = 1000 \land \\
\quad \forall n : 1 \mathinner{\ldotp\ldotp} cb.size \bullet \exists i : 1 \mathinner{\ldotp\ldotp} no_cands \bullet \\
\qquad\qquad cp[i].cand = c \land cp[i].ppref = n \land c = cb.bpref[n] \quad]
\end{array}
$$

2.3.3 From Specification Statement to Code For MakeBallot

Our first task is to examine the specification statement. A number of intermediate steps are needed for an efficient implementation. We break up the overall program into three smaller programs. An intermediate data structure, b, is used during the stripping of the votes (where duplicates, etc. are 'thrown away'). The first two programs (b:[**INIT**] and b:[**INIT**, **MID**])

deal with the calculation and population of b. The final program, (b:[**MID**, **END**]) deals with building up the final ballot, cb, from the intermediate b.

b is an array of records. The record contains two fields, *cand* containing a candidate and *no*, containing a number. b[i].cand contains a candidate that appears at preference i in the voting paper (in error, there may be a number of different candidates at this preference). b[i].no contains the number of candidates who have preference i marked against their names. When b is populated from a voting paper, we can strip down to the valid preference by noting that the first b[i].no not equal to 1 is the first non-valid preference. Everything up to this preference is copied into the ballot as valid.

Breaking Down Specification Statement

In this and subsequent sections, we use the following extra notation for $w : [pre, post]$ for clarity, i.e.

$$w : \left[\!\!\left[\frac{pre}{post} \right]\!\!\right]$$

Also we label the pre and/or post, for later use, e.g.

$$w : \left[\!\!\left[\frac{pre \quad \textbf{(Label1)}}{post \quad \textbf{(Label2)}} \right]\!\!\right]$$

Using sequential composition, the original program given in the preceding section is refined to

$\sqsubseteq$ ‘**Law C7** sequential composition’

$$|[\, var\ b \bullet$$

$$b : \left[\!\!\left[\frac{\textbf{True}}{b[1].no \mathrel{..} b[no_cands].no = 0 \qquad \textbf{(INIT)}} \right]\!\!\right];$$

$$b : \left[\!\!\left[\frac{\textbf{(INIT)}}{\begin{array}{l} \forall p : 1 \mathrel{..} no_cands \bullet \exists\, setindices : \mathbb{P}\,\mathbb{N} \mid \\ \quad setindices = \{n : \mathbb{N} \mid cp[i].ppref = p \land n \in 1 \mathrel{..} no_cands\} \bullet \\ \quad setindices \neq \varnothing \Rightarrow \qquad \textbf{(MID)} \\ \qquad b[p].no = \#setindices \land \\ \qquad\quad \exists\, i : \mathbb{N} \mid i \in setindices \bullet cp[i].cand = b[p].cand \end{array}} \right]\!\!\right];$$

$$cb : \left[\begin{array}{l} \hfill \text{(MID)} \hfill \\ \hline cb.size = min\{no : \mathbb{N} \mid \#\{n \mapsto c \mid \exists i : 1 \mathrel{..} no_cands \bullet \\ \quad cp[i].cand = c \wedge cp[i].ppref = n \wedge n > 0\} \\ \qquad (\!| \{no\} |\!) \neq 1\} - 1 \wedge \\ cb.value = 1000 \wedge \hfill \textbf{(END)} \\ \forall n : 1 \mathrel{..} cb.size \bullet \exists i : 1 \mathrel{..} no_cands \bullet \\ \quad cp[i].cand = c \wedge cp[i].ppref = n \wedge c = cb.bpref[n] \end{array}\right]$$

$]\!|$

We will take each of the programs in turn, i.e. b:[**True**, **INIT**], b:[**INIT**, **MID**] and cb:[**MID**, **END**]. We look at the refinement of b:[**True**, **INIT**] in detail and give the main structure for the work involved in refining the remaining two programs, i.e. b:[**INIT**, **MID**] and cb:[**MID**, **END**].

Refinement of b:[True, INIT]

$$b : \left[\begin{array}{l} \hfill \textbf{True} \hfill \\ \hline b[1].no \mathrel{..} b[no_cands].no = 0 \qquad \textbf{(INIT)} \end{array}\right]$$

$\sqsubseteq$ '**Law C7** sequential composition and **Law C4** introduce local variable'

$|[\ var\ k \bullet$

$\quad b, k : [b[1].no \mathrel{..} b[k-1].no = 0];$ (2.1)

$\quad b, k : [b[1].no \mathrel{..} b[k-1].no = 0,$

$\qquad b[1].no \mathrel{..} b[k-1].no = 0 \wedge k = no_cands + 1]$ (2.2)

$]\!|$

(2.1) $\sqsubseteq$ '**Law C2** assignment'

$k := 1$

(2.2) $\sqsubseteq$ '**Law C5** iteration

Inv = b[1].no .. b[k-1].no = 0,

G = k $\neq$ no_cands +1

Variant - no_cands+1 -k '

do $k \neq no_cands + 1 \rightarrow$

$\quad b : [b[1].no \mathrel{..} b[k-1].no = 0 \wedge k \neq no_cands,$

$\qquad b[1].no \mathrel{..} b[k-1].no = 0 \wedge$ $\triangleleft$

$$0 \leq no_cands + 1 - k < no_cands + 1 - k_0]$$

od

$\sqsubseteq$ '**Law C3** following assignment'

$b : [b[1].no \mathinner{..} b[k-1].no = 0 \wedge k \neq no_cands + 1,$
$\quad b[1].no \mathinner{..} b[k].no = 0 \wedge k < k + 1 \leq no_cands]; \triangleleft$

$k := k + 1$

$\sqsubseteq$ 'As k > k+1 $\Rightarrow$ k > $k_0 \Rightarrow 1 \leq V < V_0$ '

$b : [b[1].no \mathinner{..} b[k-1].no = 0 \wedge k \neq no_cands + 1,$
$b[1] \mathinner{..} b[k].no = 0] \triangleleft$

$\sqsubseteq$ **'Law C1 assignment'**

$b[k].no := 0$

This leads to the following code:

```
| [ var k•
   k:= 1;
   do k ≠ no_cands +1 →
      b[k].no := 0;
      k:= k+1;
   od
] |
```

Refinement of b:[INIT, MID]

For clarity, b is indexed by 1 to max_pref and cp by 1 to no_cands. These two values are equal as the maximum legal preference is exactly equal to the number of candidates (no 'holes' allowed, i.e. preferences must start at 1 and be contiguous). Thus b:[**INIT**, **MID**] becomes:

$$
\begin{array}{l}
b : [b[1].no \mathinner{..} b[max_pref].no = 0, \\
\hline
\forall p : 1 \mathinner{..} max_pref \bullet \\
\quad \exists\, setindices = \{si : \mathbb{N} \mid cp[si].ppref = p \wedge si \in 1 \mathinner{..} no_cands\} \wedge \\
\quad\quad b[p].no = \#setindices \wedge \\
\quad\quad setindices \neq \varnothing \Rightarrow \\
\quad\quad\quad \exists\, i : \mathbb{N} \mid i \in setindices \bullet cp[i].cand = b[p].cand
\end{array}
$$

For clarity, we will name the predicate :

$$
\begin{array}{l}
P(k) = \forall p : 1 \mathrel{..} max_pref \mid k \in 1 \mathrel{..} no_cands + 1 \bullet \\
\qquad \exists\, setindices = \{ si : \mathbb{N} \mid cp[si].ppref = p \land si \in 1 \mathrel{..} k \} \land \\
\qquad\quad b[p].no = \#setindices \land \\
\qquad\quad setindices \neq \varnothing \Rightarrow \\
\qquad\qquad \exists\, i : \mathbb{N} \mid i \in setindices \bullet cp[i].cand = b[p].cand
\end{array}
$$

As we will refine this using iteration, with the following Invariant, Guard and Variant:

$$
\begin{array}{rcl}
Inv(k) & \widehat{=} & P(k-1). \\
Guard & \widehat{=} & k \leq no_cands \\
\neg G & = & k > no_cands \\
 & & \text{(also k} \in 1 \mathrel{..} \text{no_cands+1)} \\
\Rightarrow \neg G & = & k = no_cands + 1 \\
Variant & = & no_cands + 1 - k
\end{array}
$$

$\sqsubseteq$ '**Law C6** leading assignment, followed by **Law C5** iteration, followed by **Law C3** following assignment '

MID $\sqsubseteq$ $|[\ var\ k \bullet$

$$
\begin{array}{l}
\quad k := 1; \\
\quad \textbf{do}\ k \leq no_cands \rightarrow \\
\qquad b : \left[\!\!\left[\dfrac{Inv(k) \land G}{Inv(k+1)} \right]\!\!\right] \lhd \\
\qquad k := k + 1; \\
\quad \textbf{od} \\
]|
\end{array}
$$

Note that there are 2 possibilities for any k

1. cp[k].ppref $\in$ 1 .. max_pref (valid preference)

2. cp[k].ppref $\notin$ 1 .. max_pref (invalid preference)

We apply **Law C1** alternation for the next refinement. This leads to the following:

$\sqsubseteq$ '**Law C1** alternation on 2 possibilities for k as above'

$$
\begin{array}{ll}
\textbf{if} & cp[k].ppref \in 1 \mathinner{..} max_pref \rightarrow \\
& b : \left[\!\left[\dfrac{cp[k].ppref \in 1 \mathinner{..} max_pref \land Inv(k)}{Inv(k+1)} \right]\!\right] \qquad \textbf{(VAL)} \\
\square & cp[k].ppref \notin 1 \mathinner{..} max_pref \rightarrow \\
& b : \left[\!\left[\dfrac{cp[k].ppref \notin 1 \mathinner{..} max_pref \land Inv(k)}{Inv(k+1)} \right]\!\right] \qquad \textbf{(INVAL)} \\
\textbf{fi} &
\end{array}
$$

... $\sqsubseteq$ 'Using standard Refinement Calculus Techniques'

if cp[k].ppref $\in$ 1 ..*max_pref* $\rightarrow$
b[cp[k].ppref].no := b[cp[k].ppref].no + 1;
b[cp[k].ppref].cand := cp[k].cand;
fi

(In the case that cp[k].ppref is invalid, **(INVAL)**, the preference is ignored.)

Refinement of cb:[MID, END]

The specification statement is written out, labelling the component parts for ease of use.

$$
\begin{array}{lll}
\textbf{END} & \equiv & cb : [\forall p : 1 \mathinner{..} no_cands + 1 \bullet \\
& & \quad \exists\, setindices = \\
& & \qquad \{si : \mathbb{N} \mid si \in 1 \mathinner{..} no_cands \land cp[si].ppref = p\} \land \\
& & \quad b[p].no = \#setindices \land \\
\textbf{(Pre)} & & \quad setindices \neq \varnothing \Rightarrow \\
& & \qquad \exists\, i : \mathbb{N} \mid i \in setindices \bullet cp[i].cand = b[p].cand,
\end{array}
$$

$$
\begin{array}{ll}
 & cb.size = min\{no : \mathbb{N} \mid \\
 & \quad \#\{n \mapsto c \mid \exists\, i : 1 \mathinner{..} no_cands \bullet \\
\textbf{(SIZE)} & \qquad cp[i].cand = c \wedge cp[i].ppref = n \wedge \\
 & \qquad\quad n \in 1 \mathinner{..} no_cands\} \\
 & \quad (\!|\, \{no\} \,|\!) \neq 1\} - 1 \\
 & \wedge \\
 & \forall\, n : 1 \mathinner{..} cb.size \bullet \\
\textbf{(CAND)} & \quad \exists_1\, i : 1 \mathinner{..} no_cands \bullet \\
 & \qquad cp[i].cand = c \wedge cp[i].ppref = n \wedge c = cb.bpref[n] \\
 & \wedge \\
\textbf{(VALUE)} & cb.value = 1000]
\end{array}
$$

$\equiv$ 'rewrite in terms of components'

$cb : [Pre, \textbf{SIZE} \wedge \textbf{CAND} \wedge \textbf{VALUE}]$

$\sqsubseteq$ 'sequential composition'

$cb : [Pre, \quad Pre \wedge \textbf{SIZE}];$ (2.3)

$cb : [Pre \wedge \textbf{SIZE}, \quad \textbf{SIZE} \wedge \textbf{CAND}];$ (2.4)

$cb : [\textbf{SIZE} \wedge \textbf{CAND}, \quad \textbf{SIZE} \wedge \textbf{CAND} \wedge \textbf{VALUE}]$ (2.5)

The following refinements are standard but long. We show the results. Take each statement separately:

(2.3) $\sqsubseteq$ .. $\sqsubseteq$ .. $\sqsubseteq$ ..

```
|[ var k •
   k:=0;
   do b[k+1].no = 1 →
      k:=k+1;
   od;
   cb.size := k;
]|
```

(2.4) $\sqsubseteq$ .. $\sqsubseteq$ .. $\sqsubseteq$ ..

```
|[ var k •
```

```
    k:=1;
    do k ≤ cb.size →
            cb.bpref[k] := b[k].cand;
            k:= k+1;
    od
]|
```

(2.5) $\sqsubseteq$

```
    cb.value:= 1000
```

2.3.4 Code For Procedure MakeBallot

Now we can put the component parts of the code in a procedure called MakeBallot as shown in Fig. 2.1. As all instances of k are separate and distinct, we use only one k.

2.4 Refinement of insert

In this section, the operation of inserting an item (a valid *ballot*) into a binary search tree is examined. Each node on the binary search tree needs to hold information of how many duplicate ballots exist, so each node's data part will contain both an information part (containing the ballot information) and a *count* part (holding the number of occurrences of the particular ballot).

In Butler [8], an approach to the derivation of correct algorithms on trees given recursive functions on trees (including insert) is described. This approach is used here. The refinement of insert thus involves two stages. Firstly, the operation insert is defined on a binary search tree (of type *Tree*) using functional programming and recursion techniques, as described in [6] and taking into account the presence of the count (number of copies) part. Importantly, this definition is appropriate for use with [8]. We define the tree structure using recursion. We 'plug-in' the recursive definition according to the technique described in [8], apply the rules and transformations and produce a correct implementation.

```
procedure MakeBallot(ref cb:Ballot; val cp:Paper) ≙
    var k:integer;
    var b: array[1 .. no_cands] of
            rec
                no:integer;
                cand: Candidate;
            end;
   \\ Initialisation of intermediate variable b
   k:= 1;
   do k ≠ no_cands +1 →
       b[k].no := 0;
       k:= k+1;
   od
       k:= 1;
   \\ Calculating size of valid ballot and transferring to cb.
   do k ≤ no_cands →
       if cp[k].ppref ∈ 1 ..max_pref →
           b[cp[k].ppref].no := b[cp[k].ppref].no + 1;
           b[cp[k].ppref].cand := cp.cand;
       fi;
       k := k+1;
   od
   k:=0;
   do b[k+1].no = 1 →
       k:=k+1;
   od;
   cb.size := k;
   k:=1;
   do k ≤ cb.size →
           cb.bpref[k] := b[k].cand;
           k:= k+1;
   od
   \\ Assigning value to weight.
   cb.value := 1000;
```

Figure 2.1: Code for procedure MakeBallot

2.4.1 Supporting Definitions

The type we use for the tree structure in the recursive definition is:

$$\begin{aligned} BTree &\mathrel{\widehat=} \epsilon \mid node(Item \times Tree \times Tree) \\ &\quad \text{and} \\ Item &\mathrel{\widehat=} Info \times Count \end{aligned}$$

Note, we will use the abstraction function bag, bag having the usual meaning. Specifically, when dealing with trees:

$$bag\ \epsilon = \varnothing \tag{2.6}$$

$$bag(bin((a,b),L,R)) = bag\ L \cup \{a \mapsto b\} \cup bag\ R \tag{2.7}$$

$$\text{`doms distinct'}$$

We will move onto an implementation of the tree using the following concrete types Pointer Structures

$$\begin{aligned} \textbf{Type}\, TreePtr &= \text{POINTER TO } Node; \\ Data &= \text{RECORD} \\ &\quad info : CBallot; \\ &\quad count : integer; \\ &\quad \text{END}; \\ Node &= \text{RECORD} \\ &\quad root : Data; \\ &\quad Left, Right : TreePtr; \\ &\quad \text{END}; \end{aligned}$$

2.4.2 Definition of insert Using Functional Programing

This treatment is based on Bird & Wadler [6]. We define an (insert x t) based on a binary search tree with the usual meanings except that the root information part contains the number of occurrences of the ballot as well as the ballot.
The general definition of insert is

$$bag\ (insert\ x\ t)\ =\ bag\ (t)\ \uplus [\![x]\!]$$

and leads to the following:
<u>Case t = ϵ</u>

$$\begin{aligned}
bag(insert\ x\ \epsilon) &= bag\ \epsilon \uplus [\![x]\!] \\
&= \varnothing \uplus [\![x]\!] \\
&= \{x \mapsto 1\} && \text{`from (2.6)'} \\
&= bag(bin(x,1),\epsilon,\epsilon) && \text{`from (2.7)'} \\
\Leftarrow insert\ x\ \epsilon &= bin((x,1),\epsilon,\epsilon)
\end{aligned}$$

<u>Case t $\neq \epsilon$</u>

$$t\ =\ node((a,b),L,R)$$

Lemma on $\uplus$ and bags:
bag (node((a,b), L, R) $\uplus$[x] = "because of ordering on binary search trees"

$$\begin{array}{rl}
(bag\ L \uplus [\![x]\!] \cup \{a \mapsto b\} \cup bag\ R & x < a \\
bag\ L \cup \{a \mapsto b+1\} \cup bag\ R & x = a \\
bag\ L \cup \{a \mapsto b\} \cup (bag\ R \uplus [\![x]\!]) & x > a
\end{array}$$

<u>subcase x < a</u>
bag(insert x node(a,b), L, R))

$$\begin{aligned}
&= bag(\ node((a,b),L,R)\ ) \uplus [\![x]\!] \\
&= \text{`Lemma } \uplus\text{'} \\
&\quad (bag\ L \uplus [\![x]\!]) \cup \{a \mapsto b\} \cup bag\ R \\
&= \text{`Rewrite'} \\
&\quad bag(insert\ x\ L) \cup \{a \mapsto b\} \cup bag\ R \\
&= \text{`From (2.7)'} \\
&\quad bag\ (node((a,b),\ (insert\ x\ L),R) \\
&\quad \text{`Remove bag'} \\
\Leftarrow insert\ x\ node((a,b),L,R) &= node((a,b),(insert\ x\ L),R)
\end{aligned}$$

$$
\begin{array}{rcl}
insert(x\ \epsilon) & = & node((x,1),\epsilon,\epsilon) \\
insert(x, node((a,b),L,R) & = & \textbf{if}\ x = a \rightarrow node((a,b+1),L,R) \\
 & & \Box x < a \rightarrow node((a,b),(insert(x\ L),R) \\
 & & \Box x > a \rightarrow bode((a,b),L,(insert(x\ R)) \\
 & & \textbf{fi}
\end{array}
$$

Figure 2.2: Definition of Insert

<u>subcase x > a</u>
Proof similar and leads to

$$\Leftarrow\ insert\ x\ node((a,b),L,R) = node((a,b),L,(insert\ x\ R))$$

<u>subcase x = a</u>

$$
\begin{array}{rcl}
bag(insert\ x\ node((x,b),L,R)) & = & bag\ (node((x,b),L,R) \uplus [\![x]\!]) \\
 & = & \text{`lemma'} \\
 & & bag\ L\ \cup \{x \mapsto b+1\} \cup bag\ R \\
 & = & \text{`from (2.7)'} \\
 & & bag(node((x,b+1),L,R) \\
 & = & \text{`Remove bag'} \\
\Leftarrow\ insert\ x\ node((x,b),L,R)) & = & node((x,b+1),L,R)
\end{array}
$$

This leads to the definition of insert as shown in Fig. 2.2.

2.4.3 Calculational derivation of pointer algorithms from tree operations

The derivation of correct algorithms involving pointers, especially those originally specified using recursion (e.g. trees) is difficult. A technique has been described in [8] that provides rules that allow recursive functions on trees to be transformed into imperative algorithms on pointers. We call this the 'Butler technique' during this discussion.

Generic Directed Update (*UPD*):

$$
\begin{array}{lcl}
UPD(\epsilon) & = & E1 \\
UPD(\mathsf{node}(b, L, R)) & = & \mathbf{if}\ CT(\mathsf{node}(b, L, R)) \rightarrow E2(\mathsf{node}(b, L, R)) \\
 & & [\!]\ \ CL(\mathsf{node}(b, L, R)) \rightarrow \mathsf{node}(b, UPD(L), R) \\
 & & [\!]\ \ CR(\mathsf{node}(b, L, R)) \rightarrow \mathsf{node}(b, L, UPD(R)) \\
 & & \mathbf{fi}.
\end{array}
$$

Where for any t s.t. $Is_Node(t)$, $CT(t)$, $CL(t)$ and $CR(t)$ are mutually exclusive and exhaustive.

Figure 2.3: Generic Directed Update

The approach taken in the paper is as follows: The specification for a generic update on a tree structure is presented. It is shown here as Fig. 2.3. Users of the technique should 'match' the generic terms with the specific terms in the users specific problem. Algorithmic refinement is immediately applied and the generic refinement is presented for the generic update as specified. This refinement is shown here as Fig. 2.4. This is easily rewritten by the user to match the specific problem using the matched terms. A small amount of further refinement is necessary at this point before the introduction of pointers via data refinement. This data refinement is applied through the use of many rules called *pointer-introduction transformations.* These rules are presented based on the generic components of the tree and matching to the specific is again required. Examples of a some transformations are:

$$
\begin{array}{lcllcl}
\mathcal{S}[\![\ t_i\ ]\!] & \hat{=} & p_i & \mathcal{S}[\![\ t_i := \epsilon\ ]\!] & == & p_i := \mathsf{nil}. \\
\mathcal{S}[\![\ left(t_i)\ ]\!] & \hat{=} & p_i\hat{}.left & \mathcal{S}[\![\ t_i = \epsilon\ ]\!] & == & p_i = \mathsf{nil} \\
\mathcal{S}[\![\ right(t_i)\ ]\!] & \hat{=} & p_i\hat{}.right. & \mathcal{S}[\![\ left(t_i) = \epsilon\ ]\!] & == & p_i\hat{}.left = \mathsf{nil}
\end{array}
$$

As can be seen, application of the transformations is straightforward. From the user's point of view, therefore the only overhead is matching the generic components in the original specification to the specific components of their specific problem. Apart from the extra refinement mentioned directly before the application of data refinement, no further proof is necessary.

2.4.4 Producing Code from Functional Definition of insert

The 'Butler Technique' as described in the previous section is used mechanically in this section to produce code.

Imperative version of Generic Directed Update:

$$
\begin{array}{l}
\quad \{Is_Whole(t)\}\ t := UPD(t) \\
\sqsubseteq \\
\quad \mathbf{if}\ t = \epsilon \rightarrow t := E1 \\
\quad [\!]\ \ t \neq \epsilon\ \wedge\ CT(t) \rightarrow t := E2(t) \\
\quad [\!]\ \ t \neq \epsilon\ \wedge\ \neg\ CT(t) \rightarrow \\
\qquad |[\mathbf{var}\ m \bullet m := \langle\rangle; \\
\qquad \mathbf{do}\ t/m \neq \epsilon\ \wedge\ \neg\ CT(t/m) \rightarrow \\
\qquad\quad \mathbf{if}\ CL(t/m) \rightarrow \{Is_Node(t/m)\}\ m := m \frown \langle \mathsf{left} \rangle \\
\qquad\quad [\!]\ \ CR(t/m) \rightarrow \{Is_Node(t/m)\}\ m := m \frown \langle \mathsf{right} \rangle \\
\qquad\quad \mathbf{fi} \\
\qquad \mathbf{od}; \\
\qquad \{\ m \in paths(t)\ \wedge\ m \neq \langle\rangle\ \wedge \\
\qquad\quad (CL(t/front(m)) \vee CR(t/front(m))\ \wedge \\
\qquad\quad (CL(t/front(m)) \Rightarrow last(m) = \mathsf{left})\ \wedge \\
\qquad\quad (CR(t/front(m)) \Rightarrow last(m) = \mathsf{right})\ \}; \\
\qquad \mathbf{if}\ t/m = \epsilon \rightarrow t := t[m \backslash E1] \\
\qquad [\!]\ \ t/m \neq \epsilon \rightarrow t := t[m \backslash E2(t/m)] \\
\qquad \mathbf{fi}\ ]| \\
\quad \mathbf{fi}.
\end{array}
$$

Figure 2.4: Generic Directed Update After First Algorithmic Refinement

We describe the components of the tree as follows:

$$\begin{aligned}
root(node(b, L, R)) &\ \widehat{=} b \\
left(node(b, L, R)) &\ \widehat{=} L \\
right(node(b, L, R)) &\ \widehat{=} R \\
info(x, y) &\ \widehat{=} x \\
count(x, y) &\ \widehat{=} y
\end{aligned}$$

The recursive definition of insert given in Fig. 2.2 is matched to the generic directed update given in Fig. 2.4 using the following equalities:

$$\begin{aligned}
E1 &= node((a, 1), \epsilon, \epsilon) \\
CT(t) &= a = info(root(t)) \\
CL(t) &= a < info(root(t)) \\
CR(t) &= a > info(root(t)) \\
E2(node((a, b), L, R)) &= node((a, b + 1), L, R)
\end{aligned}$$

Thus this leads to the following refinement of $\{\text{IsWhole}(t_1)\}$ [1] $t_1 := \text{insert}(a)(t_1)$, using the above

```
if t1 = ε → t1:= node((a,1), ε, ε);
[] t1 ≠ ε ∧ a = info(root(t1) → t1 := node((a,b+1), L, R);            (2.8)
[] t1 ≠ ε ∧ a ≠ info(root(t1) →
   |[ var m1 • m1 := <>;
   do t1 \ m ≠ ε ∧ a ≠ info(root(t1\ m1)) →
      if a < info(root(t1 \ m1)) →
         {Is_node(t1 \ m1) } m1 := m1 <left>
      [] a > info(root(t1\ m1)) →
         {Is_Node(t1\ m1) } m1:= m1 <right>
      fi
   od
   {m1 ∈ paths(t) ∧ m ≠ <> ∧
   (a < info(root(t1 \ front(m1)) ∨ a > info (root(t1 \ front(m1))) ∧
```

[1] Normally, $\text{IsWhole}(t_1)$ will be true when applying this rule. We assume that this is the case.

```
    (a < info(root(t1\ front(m1)) ⇒ last(m1) = left) ∧
    (a > info(root)(t1\ front(m1)) ⇒ last(m1) = right) };
    if t1\ m1 = ε → t1 := t1[m1\ node((a,1), ε, ε)]                    (2.9)
    ▯ t1 \ m1 ≠ ε →
          info((x,y), L, R) := info((x,y), L, R)[m \ info((x, y+1), L, R) / m] (2.10)
    fi ]|
fi
```

We need to further examine and refine statements (2.8), (2.9) and (2.10). Then the paths are transformed to pointers, (from [8]) and yields the following code:

```
procedure Insert(val a:CBallot; ref p:TreePtr)
if p = nil → new(p); p1^.root, p1^.right, p1^.left := (a,1), nil, nil;
▯ p ≠ nil ∧ a = p1^.root.info → p1^.root.count := p1^.root.count+1;
▯ p ≠ nil ∧ a ≠ p1^.root.info →
   |[ var q1, r1 • q1 := p;
      do q1 ≠ nil ∧ a ≠ q1^.root.info →
         if a < q1^.root.info → q1, r1 := q1^left, q1;
         ▯ a > q1^.root.info → q1, r1 := q1^.right, q1;
         fi
      od
      if q1 = nil →
         |[var p2 •
         new(p); p2^.root, p2^.left, p2^.right:= (a,1), nil, nil;
         if a < r1^.root.info → r1^.left, q1 := p2, p2;
         ▯ a > r1^.root.info → r1^.right, q1 := p2, p2;
         fi
         ]|
      ▯ q1 ≠ nil → q1^.root.count := q1^.root.count+1;
      fi
   ]|
fi
```

2.5 Refinement of Pre-processing

In this section, we look at the overall specification of *pre_process* and refine it using both the work already done on *make_ballot* (see Section 2.3) and

insert (see Section 2.4). The details of each refinement step are not included as they are standard.

2.5.1 From Z to Specification Statement

Supporting Definitions

The input for the entire *pre_process* (abstract) is *seq Paper* and the concrete version is

Type CSeqPapers == *rec*
    papers: *array*[1 .. no_voters] of CPaper;
    no_votes:integer;
  end

Retrieve Relations

Given that cp is of type CSeqPapers,

$$aseqpapers = abs(cp) = \{i : 1 \mathinner{..} cp.no_votes \bullet \ i \mapsto cp.papers[i]\}$$

Given ctree is of type BTree and abag is of type bag Ballot and where $t_1 \circledcirc t_2$ indicates the tree t_1 is a sub-tree of t_2 or 'is-a-component-of', as described in Morgan [23]:

$$abag = abs(ctree) = \{tl, tr : BTree;\ root : Data \mid (root, tl, tr) \circledcirc ctree \bullet \\ root.info \mapsto root.count\}$$

Calculation of Specification Statement for *pre_process*

We examine the the specification statement for *pre_process*

$$abag : [abag = pre_process(aseqpapers)]$$
$$\sqsubseteq_{abs}$$
$$ctree : [abs(ctree) = pre_process(abs(conarrpapers))]$$

We now work on the two sides of the equation above as before. This gives us (eventually, using standard techniques including strengthening post-conditions which allows us the extra variables of *no_valid* and *no_invalid*)

the specification statement:

$$
\begin{array}{l}
\sqsubseteq \quad \text{'\textbf{Law C8} strengthen postcondition'} \\
ctree, no_valid, no_invalid : \\
\quad [\forall i : 1 \,..\, cp.no_votes \bullet \\
\quad\quad \exists cb : CBallot \mid MakeBallot(cb, cp.papers[i]) \bullet cb.size > 0 \Rightarrow \\
\quad\quad\quad \exists root : Info;\ tl, tr : BTree \mid (root, tl, tr) \circledS ctree \land \\
\quad\quad\quad\quad root.info = cb \land \\
\quad\quad\quad\quad root.count = \#\{j : 1 \,..\, cp.no_votes;\ jb : CBallot \mid \\
\quad\quad\quad\quad\quad\quad MakeBallot(jb, cp.paper[j]) \land jb = cb \bullet j\} \land \\
\quad no_valid = \#\{i : \mathbb{N} \mid i \in 1 \,..\, cp.no_votes;\ cb : CBallot \mid \\
\quad\quad\quad MakeBallot(cb, cp.papers[i]) \land cb.size > 0 \bullet i\} \land \\
\quad no_invalid = \#\{i : \mathbb{N} \mid i \in 1 \,..\, cp.no_votes;\ cb : CBallot \mid \\
\quad\quad\quad MakeBallot(cb, cp.papers[i]) \land cb.size = 0 \bullet i\}\ ]
\end{array}
$$

2.5.2 From Specification Statement to Code for pre-processing

The Invariant for the main loop of this sub-program is:

$$
\begin{array}{l}
\forall i : 1 \,..\, k - 1 \bullet \\
\quad \exists cb : CBallot \mid MakeBallot(cb, cp.papers[i]) \bullet cb.size > 0 \Rightarrow \\
\quad\quad \exists root : Info;\ tl, tr : BTree \mid (root, tl, tr) \circledS ctree \land \\
\quad\quad\quad root.info = cb \land \\
\quad\quad\quad root.count = \#\{j : 1 \,..\, k - 1;\ jb : CBallot \mid \\
\quad\quad\quad\quad\quad MakeBallot(jb, cp.paper[j]) \land jb = cb \bullet j\} \land \\
no_valid = \#\{i : \mathbb{N} \mid i \in 1 \,..\, k - 1;\ cb : CBallot \mid \\
\quad\quad MakeBallot(cb, cp.papers[i]) \land cb.size > 0 \bullet i\} \land \\
no_invalid = \#\{i : \mathbb{N} \mid i \in 1 \,..\, k - 1;\ cb : CBallot \mid \\
\quad\quad MakeBallot(cb, cp.papers[i]) \land cb.size = 0 \bullet i\}
\end{array}
$$

with Guard = k ≤ cp.no_votes. Variant = cp.no_votes - k - 1

So program as specified in specification statement above is refined by

```
var k:int;
    next_b:CBallot;
k:= 0;
no_valid :=0; no_invalid := 0; tree = nil;
do  k ≤ cp.no_votes →
    MakeBallot(next_b, cp.papers[k]);
    if  next_b.size > 0 →
        Insert(next_b, tree);
```

```
        no_valid := no_valid+1;
    [] next_b.size = 0 →
        no_invalid := no_invalid + 1;
    fi
    k:= k+1;
od
```

2.6 Moving From the Specific to the Generic

Our examination has been based on a specific problem from our case study. If we abstract its generic pattern, we find that the problem is that of a selective mapping, where we process each of a collection of items and insert the processed item into another collection, if a certain condition p holds for the processed item. We have a generic specification for the problem, where s is the original sequence, f is the function which processes the elements of that sequence, the resultant b is a bag of the processed elements using the selective mapping based on the condition p. n is the number of successfully mapped elements:
$b := items(map\ f\ s) \upharpoonright p); \quad n := \#b$
This has the generic solution:
$\sqsubseteq .. \sqsubseteq .. \sqsubseteq$

```
|[  var x •
    b:= ∅; n:= 0;
    for i=1.. # s do
        x:= f(s[i]);
        if  p(x) then
            b:= b ⊎ ⟦ x ⟧ ||
            n:=n+1
        fi
    od
]|
```

This is an interesting general problem. Another, possibly more useful approach to this problem would be to prove the general solution and move to the specific. This is seen as further work.

2.7 Conclusions

The work in this Chapter consisted of a thorough development starting with a Z-specification. Not all details are presented as many of the refinements are long but standard. The development was broken down into three parts, as described in Sections 2.3 for processing one vote, 2.4 for inserting one processed vote into the tree and 2.5 calls the above procedures in order to process the entire collection of votes and produce a collection of processed votes. During the development of the insertion into the binary search tree, it was found that Morgan's Refinement Calculus did not fit easily with the insertion. This was at least partly due to the fact that the binary search tree is a recursive data structure. We looked at the use of functional programming techniques [6] and applying the Butler [8] technique to the resultant structures. This was found to be very useful and led to a mechanical type solution.

However, at this relatively early stage in the project, we felt that this paradigm (of Z specification followed by Morgan's Refinement Calculus) had caused us difficulties, for example

- the need to use the non–standard Butler technique as the recursive data type caused difficulties.
- there was no tool support available to help with the refinement route. This is seen as a big disadvantage of this paradigm.
- when proofs were long and complex, it was difficult to keep track of the proofs. The proof illustrated in Section 2.3.3 (that of b:[**INIT**]) is neither long nor complex, but it is not trivial to keep track of each separate path. Neither is it easy to see the overview of the path even when great care is used to present the material.

It should be noted that an interesting property of the Butler technique is that algorithmic refinement takes place before data refinement. This was seen to be successful and pointed the way for further examination of this strategy in general.

These problems led us to look towards and decide on moving to the B Method for the development of our system. The tool support was a major factor. Also the structure of the B machines means that at any point, the entire operation is viewable, at least at the highest level machine.

The case–studies (on parts of the system) using the B Method and different concrete data structures are presented in Chapter 5.

This chapter, therefore, summarises our work done using Morgan's Refinement Calculus [23] and the Butler technique as described in [8]. It is presented as a valid development.

Chapter 3

Performing Algorithmic Refinement before Data Refinement in B

3.1 Introduction

The standard approach in the development of formal systems is to apply data refinement and then proceed with algorithmic refinement on the concrete data types. In this chapter, we investigate the strategy of introducing algorithmic refinement before data refinement. We present the underlying theory of distribution of data refinement over algorithmic structures. The formal treatment is elucidated by the use of simple examples.

This idea of mixing the relative order of algorithmic refinement and data refinement is not new in formal methods in general. In Chapter 2, we use a technique described in [8] which applies algorithmic refinement before data refinement. Nor is this mixing of relative order new in the B Method in particular. It is indeed part of the theory of the B Method [1]. However, it does not seem to be used (directly) in practice. Neither the B-Toolkit nor Atelier-B at present support this approach directly. (The workaround that is currently used in the B-Toolkit is presented in Section 3.4.)

Our treatment of this approach relies on work [8, 16, 32] which simplifies the data refinement step over algorithmically refined programs by providing rules on the distribution of data refinement.

By implementing the strategy of algorithmic refinement before data refinement in the B Method, we get the benefits of the B Method's tool support coupled with a strategy which, it is felt, makes loop introduction and proving

easier.

Much of the work in this chapter is based on work presented in [9].

3.2 Laws of Distribution of Data Refinement

Our approach is to, immediately from specification, introduce algorithmic structures. These algorithmic structures are based on abstract data types. Ensuing data refinement, therefore, will be on algorithmic structures.

We have discussed the idea of least data refinement in Section 1.4.2. ($\mathcal{D}_R^{a,c}[\![S]\!]$ is the least data refinement of S).

Rules for distributing $\mathcal{D}_R^{a,c}$ through the structure of S may be found in [8, 16, 22, 24, 32]. Some of these rules are repeated in Fig. 3.1, rewritten using the notation of the B Method. The first rule **DatRef 1** deals with data refinement of a basic assignment statement. **DatRef 2** shows the conditions under which nondeterministic assignments may be data refined. **DatRef 3** shows that $\mathcal{D}_R^{a,c}$ distributes through sequential composition. The two rules, **DatRef 4** and **DatRef 6** show that $\mathcal{D}_R^{a,c}$ distributes through if-statements and loops provided the guards are equivalent under the abstraction relation. Note that the abstract B loop will normally have an associated variant and invariant. These will not be explicitly carried forward in later refinements as they are only required when the abstract loop is first introduced in a refinement step. The fifth rule, **DatRef 5** deals with the data refinement of blocks with local variables. Note that a_0, c_0 are global to the statement. a_1, c_1 are local to S. The proof obligation $(\forall\, a_1, c_1 \bullet R) \iff R$ shows that R deals with global variables (a_0, c_0) only. The final proof obligation, $(@a_1 \bullet S) = S$ states that we expect S to initialise a_1. Note that Q may involve a_0, c_0 as well as a_1, c_1, but R does not involve a_1, c_1, only a_0, c_0.

We will look at the affect of data refinement on a *procedure* in the next chapter, Chapter 4.

We calculate a data refinement of a statement S under R by calculating a refinement of $\mathcal{D}_R^{a,c}[\![S]\!]$. For example, we wish to data refine (under R) the sequential composition represented by $S_1;\ S_2$. Given that $\mathcal{D}_R^{a,c}[\![S_1]\!] \sqsubseteq S_1'$ and $\mathcal{D}_R^{a,c}[\![S_2]\!] \sqsubseteq S_2'$, and appealing to the data refinement laws of Fig. 3.1, then:

$$S_1;\ S_2 \ \sqsubseteq_R\ \mathcal{D}_R^{a,c}[\![S_1;\ S_2]\!] \ \sqsubseteq\ \mathcal{D}_R^{a,c}[\![S_1]\!];\ \mathcal{D}_R^{a,c}[\![S_2]\!] \ \sqsubseteq S_1';\ S_2'$$

So,

$$S_1;\ S_2 \ \sqsubseteq_R\ S_1';\ S_2'$$

Some interesting properties of $\sqsubseteq$ and $\sqsubseteq_R$ are:

- $S \sqsubseteq_R T \wedge T \sqsubseteq U \Rightarrow S \sqsubseteq_R U$
- $S \sqsubseteq T \wedge T \sqsubseteq_R U \Rightarrow S \sqsubseteq_R U$

3.3 Examples to Illustrate Laws

In this section, we illustrate the Data Refinement Laws as presented in Fig. 3.1, using simple examples.

In the first example, we present the MACHINE and REFINEMENT as they would appear. For the remaining examples, we use segments rather than the entire MACHINE or REFINEMENT.

An interesting part of this work is to examine how the practice of applying algorithmic refinement before data refinement compares with the more usual approach of *vice versa*. Whereas we do not attempt, in this work, to *quantitatively* compare both approaches, we attempt intuition-based comparisons in some of the non-trivial examples. We concentrate, however, in illustrating the laws in this section and draw from work presented in [9] to make the comparisons.

3.3.1 Distribution Over Basic Assignment, Using DatRef 1

Example
If we take a very simple assignment of (common variable) xx to an abstract variable aa. The concrete version of the variable is cc. this is shown in Fig. 3.2.

This refinement is correct according to the Data Refinement Law **DatRef 1**, because
$aa = 2 * cc \Rightarrow aa \; = \; 2 * cc$

3.3.2 Distribution Over Generalised Assignment, Using DatRef 2

Example
We look at an example where we non-deterministically assign a member of a set to a variable. The (data) refined version of the set is a sequence, whose range is equal to the original set. The refined program will involve non-deterministically assigning an index of the sequence to an integer variable. Given a type *AType*, the following makes up the Abstraction Relation

$$\frac{p \wedge R \Rightarrow E = F}{\mathcal{D}_R^{a,c}[\![\{p\}\ u := E]\!] \sqsubseteq u := F} \quad \textbf{(DatRef 1)}$$

$$\frac{R \wedge Q \wedge p \Rightarrow (\exists\, a' \bullet ([a, c := a', c']R) \wedge P)}{\mathcal{D}_R^{a,c}[\![\textbf{PRE}\ p\ \textbf{THEN}\ @a' \bullet (P \implies a := a')\ ]\!] \sqsubseteq @c' \bullet (Q \implies c := c')} \quad \textbf{(DatRef 2)}$$

$$\mathcal{D}_R^{a,c}[\![S_1;\ S_2]\!] \sqsubseteq \mathcal{D}_R^{a,c}[\![S_1]\!];\ \mathcal{D}_R^{a,c}[\![S_2]\!] \quad \textbf{(DatRef 3)}$$

$$\frac{p \wedge R \Rightarrow G_i \iff H_i,\ \textit{each}\ i}{\begin{array}{c}\mathcal{D}_R^{a,c}[\![\textbf{PRE}\ p\ \textbf{THEN}\ \textbf{IF}\ G_1\ \textbf{THEN}\ S_1 \\ \textbf{ELSE IF}\ G_2\ \textbf{THEN}\ S_2\ \ldots \\ \textbf{ELSE IF}\ G_n\ \textbf{THEN}\ S_n\ \textbf{END}\ ]\!] \\ \sqsubseteq\ \textbf{IF}\ H_1\ \textbf{THEN}\ \mathcal{D}_R^{a,c}[\![S_1]\!] \ldots \\ \textbf{ELSE IF}\ H_n\ \textbf{THEN}\ \mathcal{D}_R^{a,c}[\![S_n]\!]\ \textbf{END}\end{array}} \quad \textbf{(DatRef 4)}$$

$$\frac{\begin{array}{c}R \Rightarrow (\exists\, a_1, c_1 \bullet Q) \\ (\forall\, a_1, c_1 \bullet R) \iff R \\ (@a_1 \bullet S) = S\end{array}}{\mathcal{D}_R^{a_0,c_0}[\![\textbf{VAR}\ a_1\ \textbf{IN}\ S\ \textbf{END}]\!] \sqsubseteq \textbf{VAR}\ c_1\ \textbf{IN}\ \mathcal{D}_{R \wedge Q}^{(a_0,a_1),(c_0,c_1)}[\![S]\!]\ \textbf{END}} \quad \textbf{(DatRef 5)}$$

$$\frac{p \wedge R \Rightarrow G \iff H}{\begin{array}{c}\mathcal{D}_R^{a,c}[\![\textbf{PRE}\ p\ \textbf{THEN}\ \textbf{WHILE}\ G\ \textbf{DO}\ S\ \textbf{END}]\!] \\ \sqsubseteq\ \textbf{WHILE}\ H\ \textbf{DO}\ \mathcal{D}_R^{a,c}[\![S]\!]\ \textbf{END}\end{array}} \quad \textbf{(DatRef 6)}$$

Figure 3.1: Data Refinement Laws

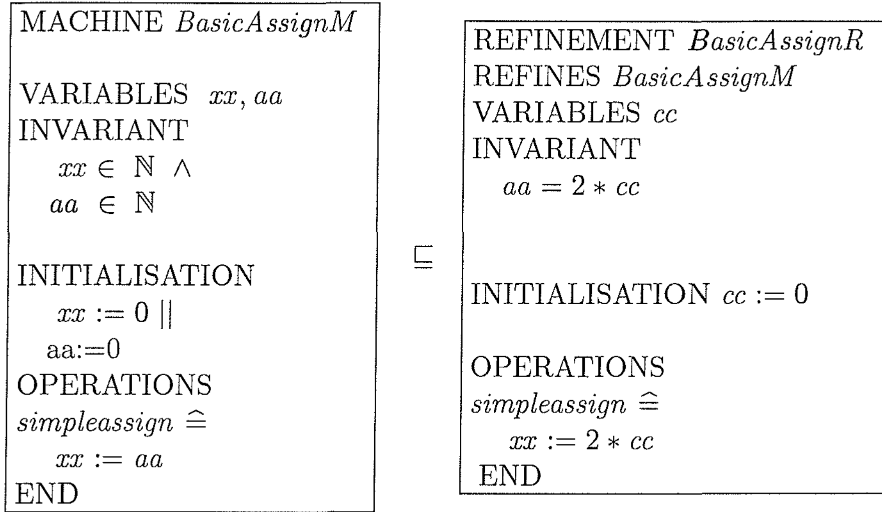


Figure 3.2: Basic Assignment

R,(between abstract xx, $ASet$ and concrete yy, SS):

$$\begin{array}{rl} R \mathrel{\widehat{=}} & ASet \in \mathbb{P}Atype \wedge SS \in seq(Atype) \wedge \\ & xx \in Atype \wedge yy \in 1..card(Aset) \wedge \\ & ran\ SS = ASet \wedge card(ASet) = card(SS) \wedge \\ & xx = SS(yy) \end{array}$$

The original code is

> **PRE** $ASet \neq \varnothing$ **THEN**
> **ANY** xx' **WHERE** $xx' \in ASet$ **THEN**
> $xx := xx'$
> **END**

According to the Data refinement rule, **DatRef 2** if

$$\begin{array}{rl} & R \wedge cc' \in 1 \ldots card(SS) \wedge ASet \neq \varnothing \\ \Rightarrow & \exists\, aa' \bullet [aa, cc := aa', cc'](R \wedge ASet \neq \varnothing) \end{array}$$

then the following is a refinement.

> **ANY** yy' **WHERE** $yy' \in 1..card(SS)$ **THEN**
> $yy := yy'$
> **END**

This is, indeed the case. ($ASet \neq \varnothing \Rightarrow \exists\, aa' \bullet aa' \in ASet$. Also $cc' \in 1..card(SS) \Rightarrow [cc := cc'](cc \in 1..card(SS)$.)

3.3.3 Distribution Over Sequence, Using DatRef 3

We look a refinement over sequence. We look at the program

BEGIN
$yy := aa * bb$
END

($yy, aa, bb \in \mathbb{N}$). aa and bb are abstract variables and are related to concrete variables, as usual via an abstraction relation. This abstraction relation could (and often would) lead to complex computation for aa and bb. We therefore split the evaluation of the abstract variables by introducing sequence (we assume the existence of $t1$ and $t2$ already, $t1, t2 \in \mathbb{N}$). So, the first, algorithmic refinement is:

BEGIN
$t1 := aa;$
$t2 := bb;$
$yy := t1 * t2$
END

We introduce the (very simple) abstraction relation, R, as: $aa = 2 * ca \wedge bb = 3 * cb \wedge ca \in \mathbb{N} \wedge cb \in \mathbb{N}$

We can therefore distribute the data refinement across the sequence, using **DatRef3**. Noting that

$$\begin{aligned} \mathcal{D}_R^{(aa,bb),(ca,cb)}[\![\, t1 := aa \,]\!] &= t1 := 2 * ca \\ \mathcal{D}_R^{(aa,bb),(ca,cb)}[\![\, t2 := bb \,]\!] &= t2 := 3 * cb \end{aligned}$$

and that $t1, t2, yy$ are common variables, then the data refinement of above is:

BEGIN
$t1 := 2 * ca;$
$t2 := 3 * cb;$
$y := t1 * t2$
END

3.3.4 Distribution Over IF statement, Using DatRef 4

We look at a simple program which checks if a word is present in a dictionary as part of a spell check program. The abstract data model of the dictionary

is a set of words. We wish to represent this as a sequence of words, ordered in some pre-defined order. So the program should establish the following:
$(wrd \in dictset \wedge ans = TRUE) \vee (wrd \notin dictset \wedge ans = FALSE)$

The full text of the invariants of both machine and refinement (which makes up R, the abstraction relation) is as follows:

$$R \mathrel{\widehat{=}} wrd \in WORD \wedge dictset \in \mathbb{P}\,WORD \wedge \\ ans \in \mathbb{N} \wedge dictseq \in seq(WORD) \wedge \\ dictseq \in Ordered(WORD) \wedge dictseq = ran\ dictset$$

Whereas the fact that the sequence is ordered is the main advantage of this refinement (and we specify this in the usual way), the exact ordering method is not of interest to us here.

We immediately introduce an IF statement which refines the original specification. (The proof is trivial and not included). We then apply data refinement to the IF statement.

Introduce IF statement

PRE $wrd \in WORD$ **THEN**
IF $wrd \in dictset$ **THEN**
  $ans := TRUE$
ELSE
  $ans := FALSE$
END

Apply Data Refinement

IF $wrd \in ran\ dictseq$ **THEN**
  $ans := TRUE$
ELSE
  $ans := FALSE$
END

The change is correctness-preserving, because, according to the data refinement law **DatRef 4**

$$wrd \in WORD \wedge \\ dictset = ran\ dictseq \wedge \\ dictseq \in Ordered(WORD) \Rightarrow wrd \in dictseq \iff wrd \in ran\ dictseq$$

and
$\mathcal{D}_R^{dictset,dictseq}[\![ans := TRUE]\!] = ans := TRUE$

The alternative approach is to use the data refinement first, i.e. to the post-condition. This gives us a post-condition of:
$(s \in ran\ dictseq \wedge ans = TRUE) \vee (s \notin ran\ dictset \wedge ans = FALSE)$
When we apply algorithmic refinement to this, i.e. introduce the IF..THEN statement, then we get
IF $s \in ran\ dictseq$ **THEN**
  $ans := TRUE$
ELSE
  $ans := FALSE$
END
We have achieved the same result in both cases.

3.3.5 Distribution Over the Introduction of a Local Variable using DatRef 5

We look at a simple example of a basic assignment within a local variable block. The original program is as follows:

$$\textbf{VAR}\ aa\ \textbf{IN}\ aa := xx\ \textbf{END}$$

We choose a concrete variable cc, with the retrieve relation, R, being $cc = 2 * aa$. (Both aa and cc are of type $\mathbb{N}$.) This leads us to a refinement of :

$$\textbf{VAR}\ cc\ \textbf{IN}\ cc := 2 * xx\ \textbf{END}$$

We prove this refinement in two steps, first the outer, **VAR** statement, using Law **DatRef 5** and then the inner statement, using **DatRef 2**.

Using **DatRef 5**, the outer refinement is correct. R is the invariant on global variables. In our case, xx is the only global variable and invariant, R is True. Q is the invariant on the local variables, i.e. aa and cc, and is $cc = 2 * aa \wedge cc \in \mathbb{N} \wedge aa \in \mathbb{N}$. S, the statement to be refined is $aa := xx$
The first proof obligation is therefore:
$True \Rightarrow (\exists\, aa, cc \bullet cc = 2 * aa \wedge aa \in \mathbb{N} \wedge cc \in \mathbb{N})$
This is easily discharged. The second proof obligation is:
$\forall\, aa, xx \bullet R \iff R$
and again is easily discharged. The third proof obligation is written
$(@aa \bullet aa := xx) = aa := xx$.
This again is trivially proven.

Next, we distribute data refinement over the statement S under R using Law **DatRef 2**. In order to do this, we rewrite the statement S as $@aa' \bullet (aa' = xx \Longrightarrow aa := aa')$. This is refined by $@cc' \bullet (cc' = 2 * xx \Longrightarrow cc := cc')$

if the following condition holds (**DatRef 2**):
$cc = 2 * aa \ \wedge \ cc \in \mathbb{N} \ \wedge \ aa \in \mathbb{N} \ \wedge \ cc' = 2 * xx \ \wedge \ True$
$\Rightarrow (\exists\, aa' \bullet ([aa, cc := aa', cc']cc = 2 * aa) \ \wedge \ aa' = xx$
'Using One-Point Rule'
$\Rightarrow [aa, cc := xx, 2 * xx]cc = 2 * aa$
$= True$
This proves that this is a correct refinement.

3.3.6 Distribution Over Loop Introduction, Using DatRef 6

This is one of the most interesting and rewarding examinations. When we mentioned that B-Toolkit or Atelier-B do not fully support algorithmic refinement followed by data refinement, it should be noted that neither support tool allow loop introduction before the implementation stage. (It follows that no further refinement can be applied). There are workarounds possible, and used. An example of such a workaround is described in Section 3.4.

We look at the approach based on the distribution of data refinement over the loop based on abstract data, as specified in **DatRef 6**.

The example we look at has as an abstract variable a finite partial function (which we use as a bag, i.e. $X \twoheadrightarrow \mathbb{N}$). We wish to produce another, new bag where each element of the domain is processed in some way. Our first data refinement is to refine the partial function to a sequence of pairs of $(X \times \mathbb{N})$. As usual, we look at the approach of seeing the original machine operation applying the loop introduction, and then applying data refinement. The original program looks like:

$process \in X \rightarrowtail Y$
...
BEGIN
$newf := \{xx, nn \mid \exists\, pp.(pp \ \in \ dom\ ff \ \wedge \ xx \ = \ process(pp) \ \wedge \ ff(pp) = nn)\}$
END

We immediately introduce a loop, with invariant:

$LI \mathrel{\widehat{=}}$
$newf = \{xx, nn \mid \ \exists\, pp.(pp \ \in \ processed \ \wedge process(pp) = xx \ \wedge$
$ff(pp) \ = \ nn) \ \} \ \wedge$
$processed \ \subseteq \ dom\ ff$
variant $card(dom\ ff - processed)$

$$
\begin{array}{l}
\textbf{BEGIN} \\
\quad processed, newf := \varnothing, \varnothing; \\
\quad \textbf{WHILE}\ dom\ ff\ -\ processed\ \neq\ \varnothing\ \textbf{DO} \\
\quad\quad \textbf{VAR}\ pp\ \textbf{IN} \\
\quad\quad\quad pp\ :\in dom\ ff\ -\ processed; \\
\quad\quad\quad processed := processed \cup \{pp\}; \\
\quad\quad\quad newf := newf \cup \{process(pp) \mapsto ff(pp)\} \\
\quad\quad \textbf{END} \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

We will now introduce the data refinement to the loop, and we wish to refine the partial function to a sequence containing the pairs, as in the abstraction relation,

$$
\begin{array}{ll}
R \mathrel{\widehat{=}} & ff = ran\ ss\ \wedge \\
& card(ff) = card(ss)\ \wedge \\
& ss\ \in\ seq(X \times \mathbb{N})\ \wedge \\
& processed\ =\ \{xx \mid\ \exists\, ii.(ii \in 1..index \wedge\ ss(index).xx = xx)\}\ \wedge \\
& index\ \in\ 0..card(ss)\ \wedge \\
& pp\ =\ rr.xx
\end{array}
$$

This gives us the following loop:

$$
\begin{array}{l}
index, ss := 0, \varnothing; \\
\textbf{WHILE}\ index\ <\ card(ss)\ \textbf{DO} \\
\quad \textbf{VAR}\ rr\ \textbf{IN} \\
\quad\quad rr := ss(index); \\
\quad\quad index := index + 1; \\
\quad\quad ss := ss\hat{}[process(rr.xx) \mapsto rr.nn]; \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

This is a refinement because (using Data Refinement Law)
$p \wedge\ R\ \Rightarrow\ dom\ ff - processed\ \iff\ index\ <\ card(ss)$.
Also

$$
\begin{array}{l}
\mathcal{D}_R^{(ff,processed),(ss,index)} [\![\\
\quad \textbf{VAR}\ pp\ \textbf{IN} \\
\quad\quad pp :\in dom\ ff\ -\ processed; \\
\quad\quad processed := processed \cup \{pp\} \\
\quad \textbf{END}\ ]\!]
\end{array}
$$

$\sqsubseteq$
$index := index + 1$

and
$\mathcal{D}_R^{(ff, processed),(ss, index)} [\![$
VAR pp **IN**
$pp :\in dom\ ff\ -\ processed;$
$newf := newf \cup \{process(pp) \mapsto ff(pp)\}$
END $]\!]$
$\sqsubseteq$
VAR rr **IN**
$rr := ss(index);$
$ss := ss\hat{}[process(rr.xx) \mapsto rr.nn]$
END

3.4 Current Practice with Loop Introduction

We have mentioned that current tools do not allow the early introduction of loops using the above techniques (distributing data refinement through abstract loops). There is however a workaround that is currently used [29]. It involves introducing a loop *early* (as we do throughout this work), as an implementation. This *implementation* is based on abstract types. The loop control and loop body (based on abstract types) are separately (data) refined and eventually implemented. By joining the eventual implementation to the 'original' loop implementation, we have the full, data and algorithmically refined and implemented code. The technique is shown in Fig. 3.3.

Whereas this workaround approach is in line with the 'layered development' strategy used in the B Method, it is felt that it does not result in clear intermediate steps. The loop guard and loop body are so logically associated that to separate them in this way seems excessive.

3.5 Conclusions

The techniques discussed in this chapter allow the introduction of loops early in the refinement cycle using laws of distribution of data refinement as shown in Fig. 3.1 and as presented in [9]. This is a sound and clear approach. Workarounds to this approach are currently used, as discussed in Section 3.4 and such machines can be checked using the B-Toolkit. This is an obvious advantage. Another point to be made about the workaround is

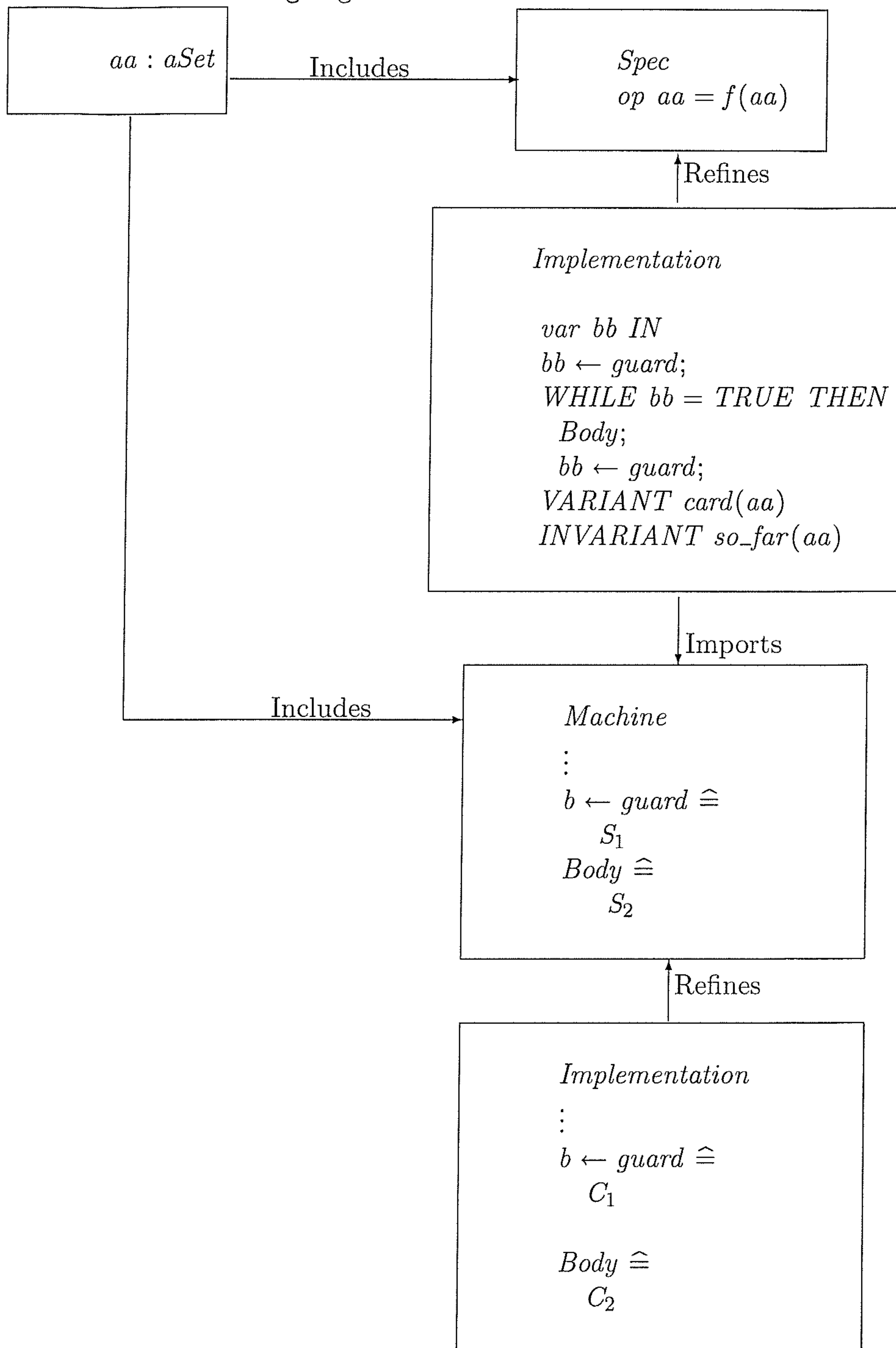


Figure 3.3: How to introduce loops early. This framework will allow early introduction of loops to be checked using the B-Toolkit.

that there is no room in this approach for further algorithmically refining a loop. This seems reasonable but may require further thought and work.

The approach we suggest is that of allowing data refinement at any stage during development, using the laws as described in this chapter. This is the first time that these laws have been used in this way. It is a cleaner, more elegant approach

Chapter 4

Interface Refinement in B

4.1 Introduction

Refinements on operations in the B Method are interface–preserving [1]. This is fundamental and sensible. Refining operations whose operations are based on abstract data types is thus not supported using standard techniques. In this chapter, we categorize these operations which need special care. We call these special operations *procedures*. We provide a rule which tells us under which conditions non interface–preserving refinements are valid. A workaround is also provided so that this refinement step can be checked by current B tools. Much of the work in this chapter is based on, and develops, work originally presented in [9].

4.2 Operations and Procedures

In this section, we discuss B Method operations in general, and our special *procedures* in particular.

In the B Method, operations allow us to manipulate the state variables. For each operation, we specify the inputs, outputs and its effect on state variables. For example, we can have a simple operation as shown in Fig. 4.1. In this case, only state variable x is affected by the original operation. The refinement is standard and shown. (The proof is standard, straightforward and not shown.) The interface remains unchanged during refinement. The user can use *Assign* without knowing that concrete y is used instead of abstract x. This is fundamental. The user should never need to be aware of what is happening behind the interface. The user's only concern is that what is delivered is as least as good for the user as what was specified (including

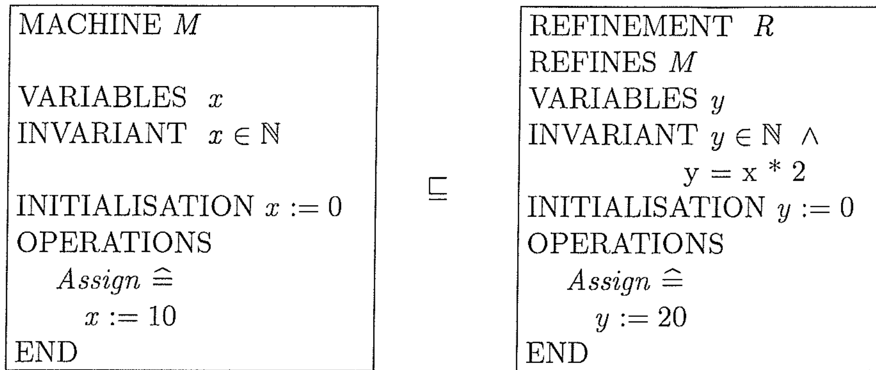


Figure 4.1: Simple Operation Refinement

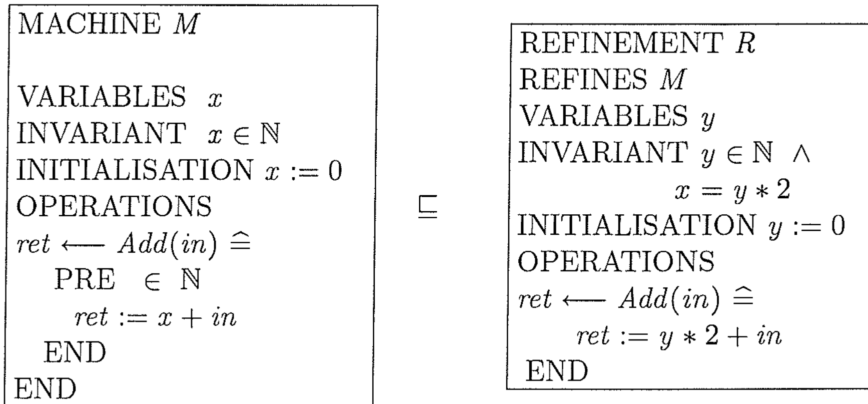


Figure 4.2: Operation Refinement with parameter and return value

Assign).

The B Method supports parameterised operations. We categorise these operations according to nature of the parameters.

The first category of operation is that when both return type and parameters are *common variables*, i.e. they do not need to be data refined during development. A simple example is as shown in Fig. 4.2.

In this case, *ret* and *in* remain unchanged during the refinement step. This refinement is thus fully supported by the B Method. Again, the proof is straightforward and not shown.

The next category of operation is that whose return types and parameters are based on abstract data types. In order to implement these operations, we need to data refine (possibly) both the return types and the parameters. This changes the interface, thus violating a fundamental tenet

of the B Method. The user is now aware of the changes necessary during development. There are many practical cases, however, when it is appropriate to specify such operations. One such case is a system which is specified using stateless machines (machines that have no state variables). Parameters are thus the only available conduit for movement of information. The operations are originally specified using abstract data, but as usual must be implemented using concrete data. A change of interface is thus necessary. This change of interface during development is not currently supported using standard techniques.

For the purposes of this work, we categorise these special operations, call them *procedures* and note that they need special care. We define *procedures* as follows:

Definition 2 *Procedures are B operations any of whose parameters or return types are based on data types which will change during refinement, according to Rule* **ProcRef**.

The rule which tells us under which circumstances a refinement involving a change of interface is correct, stated as Rule **ProcRef** is presented in Section 4.3. It is useful to establish a structure under which such a non–standard refinement can be written and checked using B tools. This workaround and underlying theory are discussed in Section 4.5.

4.3 Refinement of Procedures in B

The following law shows us under which circumstances data refinement of *procedures* is valid.

$$\frac{\begin{array}{c} a_2 \longleftarrow Aproc(a_1) \mathrel{\widehat{=}} A \\ c_2 \longleftarrow Cproc(c_1) \mathrel{\widehat{=}} C \\ \mathcal{D}_R^{(a_1,a_2),(c_1,c_2)}[\![\, A \,]\!] \sqsubseteq C \end{array}}{\mathcal{D}_{R'}^{(a_1',a_2'),(c_1',c_2')}[\![\, a_2' \longleftarrow Aproc(a_1') \,]\!] \sqsubseteq c_2' \longleftarrow Cproc(c_1')} \quad \textbf{(ProcRef)}$$

where a, c are formal parameters, a', c' are actual parameters.

This means that the code of the original *procedure* is data refined according to the laws of distribution of data refinement as detailed in Chapter 3. The interface changes and is specified in terms of the concrete data.

4.4 Examples of Data Refinement of Procedures

We look at three examples, one where only the input parameter is data refined, one where only the output parameter is data refined and finally one where both input and output parameter are data refined. In each case there will be either a gluing invariant between the input parameters, a gluing invariant between the return types, or both, depending on which are being data refined. We show the entire machine when some state is being data refined. In the final example, when we use stateless machine, we simply show the operation and specify the gluing invariants separately for clarity. In all cases, we assume that sets X and Y are available via, e.g. the inclusion of a *Global Data* machine.

4.4.1 Example 1 - Data Refinement of a Parameter

The first example takes an operation which returns the position of an (input) element of a sequence. The abstract sequence, s, is one of X, whereas the concrete sequence, sr, is one of Y. So we need to data refine state variables as well as the parameter. The return type is a common variable. The gluing invariant, I_{IN} is $I_{IN} \in Y \rightarrow X \;\wedge\; s = map\ I_{IN}\ sr$ and relates the parameters xel and yel. Further specification of this partial function is not of interest to us here. It may, for instance, specify a function that *process*es a member of Y to produce a particular X. This is shown in Fig. 4.3.

In order to be able to thus data refine the *procedure* $pos \leftarrow showpos(xel)$, we use the data refinement law as written in Section 4.3. We need to show that $\mathcal{D}_R^{xel,yel}[\![\ pos := s^{\sim}(xel)\ ]\!] \sqsubseteq pos := sr^{\sim}(yel)$ where R is the invariant contained in the machine. According to law **DatRef 1** in Chapter 3, this reduces to showing that $R \Rightarrow\ s^{\sim}(xel) = sr^{\sim}(yel)$, where R is the invariant of the REFINEMENT. The invariant implicitly includes the specification of I_{IN}, but we add that $I_{IN}(yel) = xel$.

$$
\begin{array}{cl}
 & s^{\sim}(xel) = sr^{\sim}(yel) \\
= & \exists\, n \bullet xel \mapsto n\ \in s^{\sim}\ \wedge \\
 & \qquad yel \mapsto n\ \in sr^{\sim} \\
= & \exists\, n \bullet n \mapsto xel\ \in s\ \wedge \\
 & \qquad n \mapsto yel\ \in sr^{\sim} \\
\Leftarrow & xel = I_{IN}(yel)\ \wedge\ sr = map\ I_{IN}\ s
\end{array}
$$

Therefore, the refinement as shown is valid.

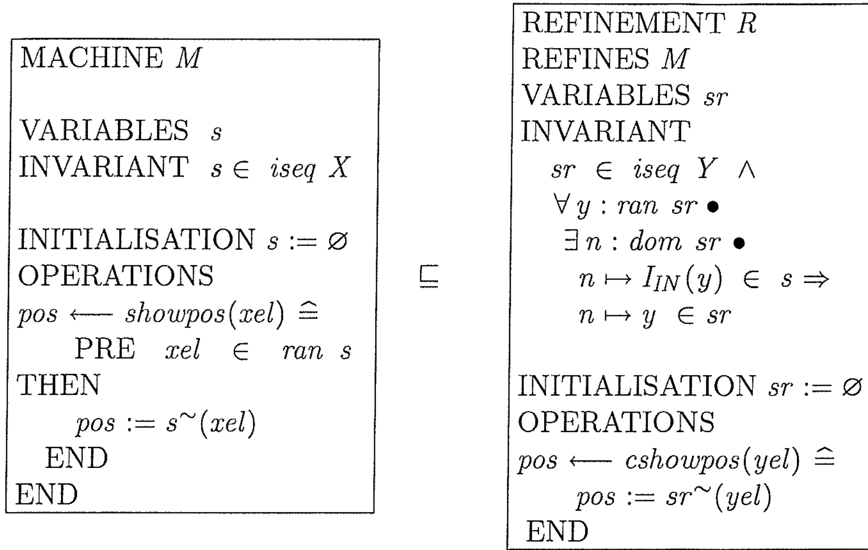


Figure 4.3: Example of refinement of operation with parameter being data refined

4.4.2 Example 2 - Data Refinement of a Return Type

In this example, illustrated in Fig. 4.4, the return type is data refined and the parameter is a common variable. It uses the same data as Example 1. In this case, the input to the operation is a position (of the sequence) and the output of the operation is the element of the sequence at that position. In the abstract operation this will be an (abstract) element of type X, whereas the concrete operation will return a (concrete) element of type Y. Again, the elements of the sequences are related by a gluing invariant (in this case I_{OUT}) which is the same as the I_{IN} of the previous example.

As with the first example, we need to show that the law for data refinement of *procedures* is obeyed. The proof is similar to that in **Example 1** and not shown.

4.4.3 Example 3 - Data Refinement of Both Return Type and Parameter

In this example, both the return type and parameter are data refined. This is a simple operation which returns the first element of a sequence, provided that the sequence is non–empty. The operation is contained in a stateless machine. In the case of the abstract operation, the first element of a sequence

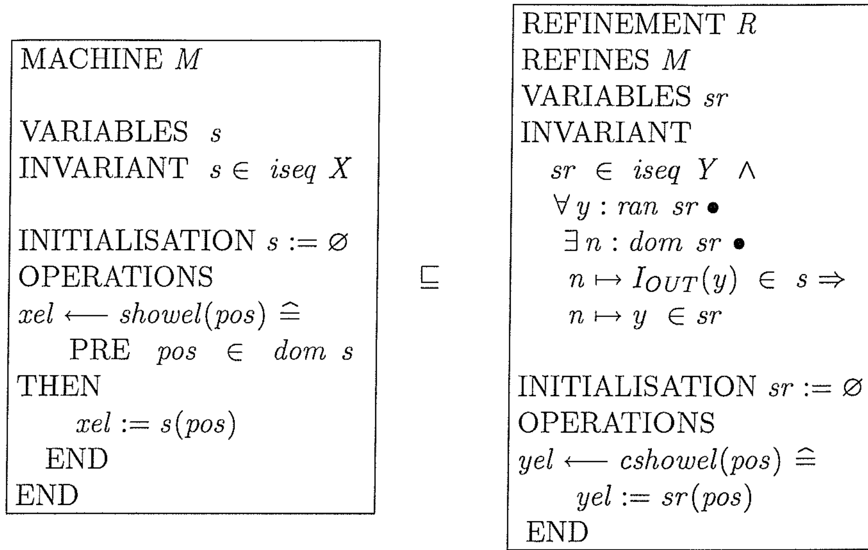


Figure 4.4: Example of refinement of operation with return type being data refined

of X's is returned. In the data refined version the first element of the corresponding sequence of Y's is returned. We show the gluing invariants separately. I_{IN} is the gluing invariant for the parameters, i.e. the sequences. I_{OUT} shows the correspondence between the elements of the sequence (again, possibly a *process*ing type of function).

$$
\begin{aligned}
I_{IN} &\mathrel{\widehat{=}} xseq = map\ abs\ yseq \\
I_{OUT} &\mathrel{\widehat{=}} abs \in Y \rightarrow X \land \\
&\qquad x = abs(y)
\end{aligned}
$$

The operations are shown in Fig. 4.5

The proof obligation (from law in Section 4.3) is easily discharged and not shown.

4.5 Workaround

As we have discussed, when a procedure is refined, its interface changes. The B tools do not at present support this non interface–preserving data refinement. In this section, we look at a technique that allows us to check that the data refined procedure is a correct refinement. The workaround is presented in two sections for clarity. In Section 4.5.1, the underlying

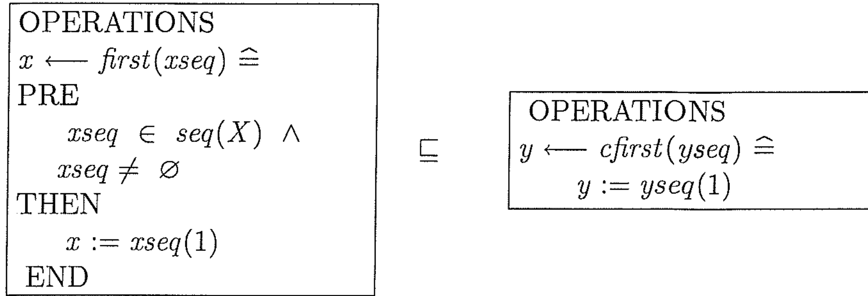


Figure 4.5: Example of refinement of operation with both return type and parameter being data refined

theory of the workaround is shown. It is conceptual and uses the example of one machine's operation being refined by another machine's operation. It is however, not directly usable with current B tools. Section 4.5.2 shows, using a simple example, how to use the ideas presented in 4.5.1 to produce a structure which is usable in current B tools.

4.5.1 Underlying Theory of Workaround

Refinements using the rule for procedures discussed in Section 4.3 can be checked with the B tools by using the following technique. Assume we have

$$a_2 \longleftarrow Aproc(a_1) \mathrel{\widehat{=}} A$$
and
$$c_2 \longleftarrow Cproc(c_1) \mathrel{\widehat{=}} C,$$

where a_1, c_1 are linked by I_{IN} and a_2, c_2 are linked by I_{OUT}. *Aproc* is an operation of machine M_1 and *Cproc* is an operation of machine M_2. We want to check that *Cproc* is a data refinement of *Aproc*. We cannot do this directly. Instead, we construct machines M_1' and M_2' respectively with operations $Proc'$ as follows:

In abstract M_1' :
$c_2 \longleftarrow Proc'(a_1) \mathrel{\widehat{=}}$
$\quad$ **VAR** a_2 **IN** $a_2 \longleftarrow Aproc(a_1)$; $c_2 : I_{OUT}$ **END**

In concrete M_2':
$c_2 \longleftarrow Proc'(a_1) \mathrel{\widehat{=}}$
$\quad$ **VAR** c_1 **IN** $c_1 : I_{IN}$; $c_2 \longleftarrow Cproc(c_1)$ **END**

(Note that we use the new notation $c : I$ whose older form was $c :\in \{c \mid I\}$). Now the two operations have the same interface and the tool can be used to check that the 'concrete' *Proc'* is refined by abstract *Proc'*.

According to the above structure, showing that the abstract *Proc'* is refined by the concrete *Proc'* is equivalent to showing that A is data refined by C under $I_{IN} \wedge I_{OUT}$. To prove this, we need to show that $A \sqsubseteq_{I_{OUT} \wedge I_{IN}} C$ is a 'sufficient condition' for
$\mathbf{var}\ a_2\ in\ A;\ c_2 : I_{OUT}\ \mathbf{end} \quad \sqsubseteq \quad \mathbf{var}\ c_1\ in\ c_1 : I_{IN};\ C\ \mathbf{end}$

Structure of Proof: The proof is shown at two levels. Firstly, the overall structure is shown. One of the steps, which is justified using different techniques (i.e including weakest precondition calculus) is fully developed separately. Note that when quantifying over all q's during the proof, q is independent of a_1 anda_2(a_1 is input only, and a_2 is a local variable).

$\mathbf{var}\ a_2\ in\ A;\ c_2 : I_{OUT}\ \mathbf{end}$
$\sqsubseteq$ 'body of var is independent of c_1'
$\mathbf{var}\ a_2, c_1\ in\ A;\ c_2 : I_{OUT}\ \mathbf{end}$
$\sqsubseteq$ 'strengthen initialisation'
$\mathbf{var}\ a_2, c_1 \mid I_{IN} \wedge I_{OUT}\ in\ A;\ c_2 : I_{OUT}\ \mathbf{end}$
$\sqsubseteq$ 'assertion can be introduced because of initialisation condition'
$\mathbf{var}\ a_2, c_1 \mid I_{IN} \wedge I_{OUT}\ in\ \{I_{IN} \wedge I_{OUT}\};\ A;\ c_2 : I_{OUT}\ \mathbf{end}$
$\sqsubseteq$ 'step ξ'
$\mathbf{var}\ a_2, c_1 \mid I_{IN} \wedge I_{OUT}\ in\ C\ \mathbf{end}$
$\sqsubseteq$ 'C independent of a_2, I_{OUT} independent of c_1'
$\mathbf{var}\ c_1\ in\ c_1 : I_{IN};\ C\ \mathbf{end}$

Next, we look at 'step ξ' by using wp calculus and the usual definition of $S \sqsubseteq T$ precisely when $wp(S, q) \Rightarrow wp(T, q)$ over all q. Below, assume universal quantification over q :

$wp(\mathbf{var}\ a_2, c_1 \mid I_{IN} \wedge I_{OUT}\ in\ \{I_{IN} \wedge I_{OUT}\};\ A;\ c_2 : I_{OUT}\ \mathbf{end}, q)$
$=$ 'wp calculus - local var'
$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet wp(\{I_{IN} \wedge I_{OUT}\};\ A;\ c_2 : I_{OUT}, q)$
$=$ 'wp calculus - sequence'
$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet wp(\{I_{IN} \wedge I_{OUT}\}, wp(A, \forall c_2 : I_{OUT} \Rightarrow q))$
$=$ 'wp calculus - assertion'
$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet (I_{IN} \wedge I_{OUT} \wedge wp(A, \forall c_2 : I_{OUT} \Rightarrow q))$
$\Rightarrow$ 'since $A \sqsubseteq_{I_{IN} \wedge I_{OUT}} C$'

$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet wp(C, \exists a_1, a_2 \bullet I_{IN} \wedge I_{OUT} \wedge \forall c_2 : I_{OUT} \bullet \Rightarrow q)$
$\Rightarrow$ 'pred calculus and monotonicity'
$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet wp(C, \exists a_1, a_2 \bullet q)$
$=$ 'a_1, a_2 not in postcondition'
$\forall a_2, c_1 \mid I_{IN} \wedge I_{OUT} \bullet wp(C, q)$
$=$ 'wp calculus'
$wp(\mathbf{var}\ a_2, c_1 \mid I_{IN} \wedge I_{OUT}\ in\ C\ \mathbf{end}, q)$

This method is therefore sound. The inspiration for this structure was based on properties of data refinement described by von Wright [32]. There, the gluing invariant is represented as a predicate transformer rather than a predicate. Let α be a gluing predicate transformer. It is shown that 'S is data refined by T under α' is equivalent to:

$$S;\ \alpha\ \sqsubseteq\ \alpha;\ T,$$

where $S \sqsubseteq T$ is standard algorithmic refinement. Operationally, α can be viewed as a command that (nondeterministically) transforms an abstract state into a concrete state. For standard data refinement, predicate transformer α may be constructed from a gluing invariant I as $c : I$.

4.5.2 Implementation of Workaround

The description of the workaround above is clear. However, it cannot be implemented directly. This is mainly due to the fact that sequence (;) is not allowed in B machines, at least not presently. To work around this, we specify an original, 'dummy', 'mixed' operation, $c_2 \longleftarrow Proc'(a_1)$ without sequence. It is only necessary that this operation may be refined to the operation as written in M_1' above. This operation is then refined to the operation as written in M_2' above. We illustrate this structure in Fig. 4.6, using the example procedure already used in Section 4.4.3. We re-use the gluing invariants I_{IN} and I_{OUT} from the example and note that they would appear in the CONSTANTS...PROPERTIES section in the appropriate REFINEMENTs.

Having the machines structured as in Fig. 4.6, we can use the workaround described in Section 4.5.1. The two refinements in Fig. 4.6 echo the operations contained in the machines M_1' and M_2' in Section 4.5.1 which are in turn refinements of the original 'dummy' operation contained in $M3$.

The significant property of the 'dummy' operation $y \leftarrow mixedfirst(xseq)$ as originally specified in $M3$ is that it can be refined by the namesake operation contained in $MR3$. The workaround, whose underlying soundness was

described in Section 4.5, can be implemented now by checking that $M3RR$ is indeed a refinement of $M3R$ using the B tools directly.

4.6 Conclusions

In this chapter, the use of parameterised operations in B has been examined. The operations which need special care are categorised and named *procedures*. A law is presented to show when the refinement of these procedures is valid. As this is a non–standard strategy, a workaround is presented which allows the user to check the non interface–preserving refinement using the current B tools. This workaround is shown to be sound.

```
MACHINE M1
:
OPERATIONS
x ⟵ first(xseq) ≙
PRE
   xseq ∈ seq(X) ∧
   xseq ≠ ∅
THEN
     x := xseq(1)
  END
END
```

```
MACHINE M2
:
OPERATIONS
y ⟵ cfirst(yseq) ≙
PRE
   yseq ∈ seq(Y) ∧
   yseq ≠ ∅
THEN
     y := yseq(1)
  END
END
```

```
MACHINE M3
:
OPERATIONS
y ⟵ mixedfirstseq(xseq) ≙
PRE
   FALSE
THEN
     y :∈ YSet
  END
END
```

⊑

```
REFINEMENT M3R
/*c.f. M1' */
REFINES M3
INCLUDES M1
:
OPERATIONS
y ⟵ mixedfirstseq(xseq) ≙
   VAR x IN
    x ⟵ first(xseq);
    y :∈ {y | I_OUT}
  END
END
```

⊑

```
REFINEMENT M3RR
/*c.f. M2' */
REFINES M3R
INCLUDES M2
:
OPERATIONS
y ⟵ mixedfirst(xseq) ≙
   VAR
  yseq IN
   yseq :∈ {yseq | I_IN};
   y ⟵ cfirst(yseq)
  END
END
```

Figure 4.6: Structure of Machines for Workaround for Interface Refinement
The refinement of operation $x \longleftarrow first(xseq)$ in MACHINE M1, by the operation $y \longleftarrow cfirst(yseq)$ in MACHINE M2 can be checked according to the framework above.

Chapter 5

Case Studies in B Development

5.1 Introduction

This chapter describes the developmental process of two parts of the overall W.I.T. Academic Council (A.C.) Election system as described in Appendix A and specified in Z in Appendix B and as specified in the B Method's AMN in Appendix D. The two parts of the system which we develop are:

- the pre-processing of votes
- the setting up of the first count

In Chapter 2, we developed (the pre-processing) part of the W.I.T. A.C. Election system from the Z Specification in Appendix B. This was done mostly using Morgan's Refinement Calculus [23]. As discussed in Section 2.7, the development was not straightforward. We decided to approach the development of the system (starting with the same pre-processing part) using the B Method. We took the opportunity to re-examine our choice of concrete data structure and this resulted in a fundamental change thereof.

5.2 Design Issues - B Method and C++'s S.T.L.

Having decided to move to using the B Method, we decided to take a fresh look at our concrete data structure. The development towards the binary search tree was difficult, though its properties led to an efficient implementation. Our new solution needed to be at least as efficient. Our choice of

new concrete data structure was contained in the C++ Standard Template Library (S.T.L.). We model (some of) the existing S.T.L. libraries and then only need to refine to the level of the library.

C++ programs call on a large number of functions from the Standard C++ Library. These functions provide essential services such as input and output. Of the many libraries, (currently) 13 constitute the Standard Template Library. These define numerous templates that implement useful and efficient algorithms and the containers that these algorithms can work on. A container is a class that *contains* other objects. An example of a container is a *list*. S.T.L. defines a template for a list, i.e. a list which contains generic items, together with algorithms/functions for inserting, deleting items, etc.

For a fuller discussion of C++ S.T.L., see [7]. There are many very useful on-line tutorials, e.g. [17].

There are many containers in S.T.L.. The container that we choose for our system is called the multiset. The multiset is the most appropriate container to store items where duplicates occur. We use a multiset to model the input (abstractly a bag of *papers*) and the output (abstractly a bag of *ballots*).

They are an efficient mechanism for storing multiple occurrences of items, e.g., an insert into a multiset is of order log N. They are modelled here (as a B Machine) as a sequence of items (in our case a *ballot* or *paper*). In implementation, multisets work by using an ordering function (supplied by the user on instantiation of a multiset). The order of the sequence is defined by a constant *X_Order* (e.g., *PaperOrder*) which is local to the machine. There is an invariant on the multisets stating that they are always ordered according to this function.

The implementation is not as simple as modelled as we can see from the (supplied) costings of the different operations. A complex 'behind the scenes' structure delivers an efficient mechanism with a simple interface. We model these multisets, and the 'call' of these operations in the development will simply be rewritten as calls to the actual C++ S.T.L. code.

We need two types of multiset machines, the multiset containing concrete papers and the multiset containing concrete ballots. A desired construct would be a generic multiset machine of the form

MACHINE $\quad Multiset(Datatype, X_Order)$,

where *Datatype* is the data being stored and *X_Order* defines the order in which the data part of *Datatype* is sequenced. We could instantiate this machine in both ways. This, however, is not easily done in B because of

the constraints on machine parameters (does not allow functions or complex data types). Inelegant workarounds are possible, but we present two separate machines, *MultisetPapers* (Fig. 5.1) and *MultisetBallots* (Figs. 5.2 and 5.3). Obviously, all multiset machines should have the same operations specified and implemented. The figures present only those operations used specifically for the case studies in this Chapter. The machines use what we will later describe as concrete data types of

$$CPaper \mathrel{\widehat{=}} iseq(Candidate \times \mathbb{N})$$
$$CBallot \mathrel{\widehat{=}} iseq(Candidate)$$

5.3 Description of System

The first part of the system that we develop is the pre–processing of votes. This sub–system has been described in Section 2.2. In the treatment of this chapter, we make some changes to the specification of Chapter 2. The first is that we ignore the 'weight' of the ballots. It was felt that this omission made the process clearer for the reader without losing any of the main character of the work. The second change is that we model the input, in this instance as a bag of *paper*s rather that as a sequence of *paper*s. The third change is that bags are now total functions.

At the abstract level, the input is modelled as a bag of *paper*s where $bag\ T == T \rightarrow \mathbb{N}_1$, where $b(item)$ returns the number of occurrences of *item* in the bag. This raw input *paper* is processed to become what we term a *ballot*. The operation which deals with this has an abstract specification called *Make_Ballot*. The pre-processing of the bag of papers returns a bag of ballots. Not all votes will be valid, so some input will be discarded. The first case study will deal with this overall operation, whose abstract specification is called *Pre_Process* and includes the operation *Make_Ballot*.

The first step (pre-processing of votes into validated ballots) having been completed, we proceed to the counting of the ballots. This is effected through a series of '*counts*'. The result of the first count is that each candidate will have 'allocated' to him/her the ballots on which they appear as first preference. Note that because of the pre-processing step, each ballot will have (at least) a first preference. The setting up of this first count (*Setup_First_Count*) is the subject of our second case study. Note that in the specification, we use stateless machines. We specify the (parameterised) operations using mathematical functions defined in the CONSTANTS and PROPERTIES clauses. The final operation simply calls these functions.

```
MACHINE  MulitsetPapers
CONSTANTS  PaperOrder
PROPERTIES
     PaperOrder ∈ ℙ(seq(CPaper))      /* Set of all Ordered sequences */
VARIABLES
     msetpapers, /* The sequence containing the papers */
     iter /* The multiset iterator, which indicates the current
          position in the multiset. */
INVARIANT
     msetpapers ∈ seq(CPaper) ∧
     iter ∈ 1..card(msetpapers) + 1 ∧
     msetpapers ∈ PaperOrder
OPERATIONS
     Start ≙ iter := 1;

     number ⟵ Count(paper) ≙
     PRE
          paper ∈ CPaper
     THEN
          number := card({nn | nn ∈ ℕ ∧ msetpapers(nn) = paper}
     END

     paper ⟵ GetNext ≙
     PRE
          iterNotAtEnd
     THEN
          paper := msetpapers(iter) || iter := iter + Count(paper);
     END

DEFINITIONS
     iterNotAtEnd ≙ (iter ≤ card(msetpapers))
END
```

Figure 5.1: Mulitset of papers machine

MACHINE *MultisetBallot*
CONSTANTS *BallotOrder*
PROPERTIES
$\quad BallotOrder \in \mathbb{P}(seq(CBallot))$
VARIABLES *msetballots*
INVARIANTS
$\quad msetballots \in seq(CBallot) \;\wedge$
$\quad msetballots \in BallotOrder$
INITIALISATION
$\quad msetballots := \varnothing$
OPERATIONS
$\quad MultiInsert(item, number) \;\widehat{=}$
$\quad$ **PRE**
$\qquad item \in CBallot \;\wedge\; number \in \mathbb{N}$
$\quad$ **THEN**
$\qquad$ **ANY** $msetballots', aa, bb$ **WHERE**
$\qquad\quad msetballots = aa\hat{\ }bb \;\wedge$
$\qquad\quad msetballots' = aa\hat{\ }\{nn \mapsto item \mid nn \in 1 \mathinner{..} number\}\hat{\ }bb \;\wedge$
$\qquad\quad msetballots' \in BallotOrder$
$\qquad$ **THEN**
$\qquad\quad msetballots := msetballots'$
$\qquad$ **END**
$\quad$ **END**;

$\quad MakeEmpty \;\widehat{=}\; msetballots := \varnothing;$

$\quad number \longleftarrow Count(ballot) \;\widehat{=}$
$\quad$ **PRE**
$\qquad ballot \in CBallot$
$\quad$ **THEN**
$\qquad number := card(\{nn \mid nn \in \mathbb{N} \wedge msetballots(nn) = ballot\})$
$\quad$ **END**

$\quad ballot \longleftarrow GetNext \;\widehat{=}$
$\quad$ **PRE**
$\qquad iterNotAtEnd$
$\quad$ **THEN**
$\qquad ballot := msetballots(iter) \parallel iter := iter + Count(ballot);$
$\quad$ **END**

$\quad bb \longleftarrow CurrentBallot \;\widehat{=}$
$\qquad bb := msetballots(iter);$

Figure 5.2: Multiset of ballots machine

$Start \widehat{=} iter := 1;$

$MoveToNextBallot \widehat{=} iter := iter + Count(msetballots(iter))$

END

Figure 5.3: Mulitset of ballots machine... contd.

Part of this case study has been presented as part of [9]. The specification of the *Pre_Process* part has been changed (from the paper) to use bags as total functions and to use a neater specification.

5.4 Case Study 1 - Pre–Processing

The specification of the pre-processing using bags as total functions is as written in Fig. 5.4. The specification use two global functions. The first is a special map function that works on bags, which we call map_b.

$$\begin{array}{l} map_b \in ((Paper \rightarrow Ballot) \times (bag\ Paper)) \rightarrow bag\ Ballot \quad \wedge \\ \forall (ff, bb).(ff \in Paper \rightarrow Ballot \wedge bb \in bag\ Paper \Rightarrow \\ \quad map_b(ff, bb) = \\ \quad\quad \lambda\, yy.(yy : Ballot \mid \Sigma xx.(xx \in Paper \wedge ff(xx) = yy \mid bb(xx)))) \end{array}$$

The second is a standard *restrict* function, written for bags (as total functions):

$$\begin{array}{l} restrict \in (bag\ Ballot \times \mathbb{P}Ballot) \rightarrow bag\ Ballot \quad \wedge \\ \forall (bb, ss).(bb \in bagBallots \wedge ss \in \mathbb{P}Ballot \Rightarrow \\ \quad restrict(bb, ss) = \lambda\, yy.(yy \in Ballot \mid 0) \mathbin{<\!\!+} (ss \lhd bb)) \end{array}$$

Before we specify the parameterised operations, we introduce a few types. The raw input is modeled as simply a partial function from *Candidate* to $\mathbb{N}$. The type is called *Paper.* The validated form of the paper is modelled as an injective sequence of *Candidates* in order of preference. This type is called *Ballot.* In summary, the types are as follows:

$$\begin{array}{c} Paper \widehat{=} Candidate \mathrel{+\!\!\!\rightarrow} \mathbb{N} \\ Ballot \widehat{=} iseq(Candidate). \end{array}$$

We have a system-wide constant called *no_cands* and stands for the number of registered candidates. It is an important number as it limits the size of both the *paper* and the *ballot*.

CONSTANTS
make_ballot,
pre_process

PROPERTIES

$$
\begin{array}{l}
make_ballot \in Paper \rightarrow Ballot \quad \wedge \\
\forall\ paper.(paper \in Paper \Rightarrow \\
\quad make_ballot(paper) = \\
\qquad 1..(min(\{nn \mid nn \in 1..no_cands + 1 \ \wedge\ card(paper^{\sim}[\{nn\}]) \neq 1\}) - 1) \\
\qquad\qquad \lhd\ paper^{\sim}) \\
\wedge \\
pre_process \in bag\ Paper \rightarrow bag\ Ballot \quad \wedge \\
\forall\ bagpapers.(bagpapers \in bag\ Paper \Rightarrow \\
\quad pre_process(bagpapers) = \\
\qquad restrict\ (\ (map_b\ (make_ballot, bagpapers)\ ), \\
\qquad\qquad \{ballot \mid ballot \in Ballot \ \wedge\ card(ballot) > 0\}\ )
\end{array}
$$

OPERATIONS
$bagballots \longleftarrow Pre_Process(bagpapers) \ \widehat{=}$
PRE
  $bagpapers \in bag\ Paper$
THEN
  $bagballots := pre_process(bagpapers)$
END

Figure 5.4: Abstract Specification of pre-processing

The first part of the first case study illustrates the development of the *Make_Ballot* operation. The abstract *paper* is a function from *Candidate* to $\mathbb{N}$ and, if we invert the function, we have a relation from preferences to candidates at that preference. If we call the lowest non-unique preference $\geq$ 1 *first_skip_or_dup*(licate), i.e., the lowest preference either to appear more that once (duplicate) or not at all (causing a skip in the order of preferences), then it follows that all preferences between 1 and $first_skip_or_dup - 1$ appear exactly once. Thus, if we domain restrict the inverted abstract *paper*'s function between 1 and $first_skip_or_dup - 1$, we have a sequence. This sequence contains the candidates in order of preference and is the abstract *ballot*. This sequence is injective as it is formed from an inverted function. It may happen that all preferences are used 'correctly', i.e., the size of the abstract *ballot* is *no_cands*. This case is dealt with by the use of $nn \in 1..no_cands + 1$ in the definition of *make_ballot*(*paper*).

The rest of the first case study illustrates the development of the *Pre_Process* operation which takes a collection of *ballot*s, each returned by the *Make_Ballot* operation, and inserts them into an output bag under certain conditions.

5.4.1 Development of Make_Ballot

The abstract specification of *Make_Ballot* is as follows:

$$
\begin{array}{l}
aballot \longleftarrow Make_Ballot(apaper) \mathrel{\widehat{=}} \\
\textbf{PRE} \\
\quad apaper \in Paper \\
\textbf{THEN} \\
\quad aballot := make_ballot(apaper) \\
\textbf{END}.
\end{array}
$$

When we substitute for *make_ballot* in the assignment, we get

$$
\begin{array}{l}
aballot := (\ 1..min(\{nn \mid nn \in 1..no_cands + 1 \wedge card(apaper^{\sim}[\{nn\}]) \neq 1\}) - 1\ ) \\
\qquad \lhd\ apaper^{\sim}.
\end{array}
$$

If we let

$$first_skip_or_dup = min(\{nn \mid \in 1..no_cands{+}1 \wedge card(apaper^{\sim}[\{nn\}]) \neq 1\}),$$

then the above can be rewritten as

$$aballot := (\ 1..first_skip_or_dup - 1\ ) \ \lhd apaper^{\sim}.$$

The R.H.S. of the assignment statement can be simplified, using the definition of $\lhd$ to

$$\{\ nn \mapsto cc \mid nn \in 1..first_skip_or_dup - 1 \wedge cc \mapsto nn \in apaper\ \}.$$

We explore the two possible paths of development, the style of algorithmic refinement first. We take the following approach on deciding on the shape of our implementation: We guess at a possible implementations. Having this possible implementation means that we have something to aim for which helps us to make decisions during development. Using this approach means that for our case studies, we start with the same specification and expect to arrive at a similar implementation using both styles of development.

Make_Ballot - Algorithmic Refinement Followed by Data Refinement

We look directly for a loop invariant based closely on the structure of the specification, as follows:

$$LI_1 \mathrel{\widehat{=}} \quad aballot = \{\ nn \mapsto cc \mid nn \in 1..so_far - 1 \wedge cc \mapsto nn \in apaper\ \}.$$

The guard of the loop is $so_far < first_skip_or_dup$. We introduce the loop shown in Fig. 5.5. Note that $apaper^{\sim}(so_far)$ is well defined since:

$$\begin{array}{l} apaper \in Candidate \twoheadrightarrow \mathbb{N} \wedge so_far < first_skip_or_dup \\ \Rightarrow\ \forall\, ii.(ii \in 1.so_far \Rightarrow card(apaper^{\sim}[\{ii\}]) = 1 \\ \Rightarrow\ 1..so_far \lhd apaper^{\sim} \in \mathbb{N} \twoheadrightarrow Candidate. \end{array}$$

Most of the proof obligations generated from the introduction of this loop are easily discharged. We have a close look at the P-Rule [34], i.e., $LI_1 \wedge G \Rightarrow [Body]LI_1$.

$$\begin{array}{ll} & [aballot(so_far) := apaper^{\sim}(so_far)] \\ & [so_far := so_far + 1] \\ & (aballot = \{\ nn \mapsto cc \mid nn \in 1..so_far - 1 \wedge cc \mapsto nn \in apaper\ \}\ ) \\ & \\ = & [aballot(so_far) := apaper^{\sim}(so_far)] \\ & (aballot = \{\ nn \mapsto cc \mid nn \in 1..so_far \wedge cc \mapsto nn \in apaper\ \}\ ) \\ & \\ = & (aballot \mathbin{<\!\!+} \{so_far \mapsto apaper^{\sim}(so_far)\}) \\ & \quad = \{\ nn \mapsto cc \mid nn \in 1..so_far \wedge cc \mapsto nn \in apaper\ \} \\ & \\ \Leftarrow & aballot = \{\ nn \mapsto cc \mid nn \in 1..so_far - 1 \wedge cc \mapsto nn \in apaper\ \} \\ = & LI_1. \end{array}$$

```
VAR so_far IN
    so_far := 1;
    WHILE so_far < first_skip_or_dup DO
        aballot(so_far) := apaper~(so_far);
        so_far := so_far + 1;
    INVARIANT  LI_1
    VARIANT    first_skip_or_dup - so_far
    END
END
```

Figure 5.5: First Loop Introduction on *Make_Ballot*

The loop requires further refinement as the calculation of *first_skip_or_dup* and of $apaper^{\sim}(so_far)$ are nontrivial. We construct a local variable, *bb* which has the following value:

$$\begin{array}{ll} BB \mathrel{\widehat=} & \{1..no_cands \times \{(\bot, 0)\} \;\lessdot\!\!+ \\ & \{\; ii \mapsto (cand, no) \;\mid\; ii \in ran(apaper) \;\wedge \\ & \quad cand \in apaper^{\sim}[\{ii\}] \;\wedge \\ & \quad no = card(apaper^{\sim}[\{ii\}]) \;\}. \end{array}$$

Here $\bot$ represents a special null candidate.

Each pair at position *ii* contains a candidate with preference *ii* associated in *apaper* and the number of candidates at this preference. When this number is 1, the candidate of the pair is the unique candidate at this preference. A pre-condition of the former loop is that $bb = BB$. We introduce the local variable, *bb* with its property in Fig. 5.6.

We proceed with data refinement on the above loop with the assertion that $bb = BB$ holds. We also take advantage of this refinement step to replace *aballot* by *cballot* using the following simple gluing invariant

$$GI_1 \;\mathrel{\widehat=}\; aballot = cballot.$$

We proceed as follows: The outermost **VAR** statement is refined using Law **DatRef 6** from Fig. 3.1. This does not result in any change in the introduced variables, but distributes the data refinement inside the statement. We accordingly apply Law **DatRef 3** from Fig. 3.1 to the sequentially composed statements in the **VAR** statement. This means that each component statement will be data refined using GI_1 as R. There is no change in the first two statements. The third is more interesting. It is a loop and we apply

```
VAR so_far, bb IN
    so_far, bb := 1, ∅;
    bb := BB ;
    PRE bb = BB THEN
        WHILE so_far < first_skip_or_dup DO
            aballot(so_far) := apaper~(so_far);
            so_far := so_far + 1;
        END
    END
END
```

Figure 5.6: *Make_Ballot* after the introduction of intermediate variable *bb*.

```
VAR so_far, bb IN
    so_far, bb := 1, ∅;
    bb := BB ;
    WHILE bb(so_far).no = 1 DO
        cballot(so_far) := bb(so_far).cand;
        so_far := so_far + 1;
    END
END
```

Figure 5.7: *Make_Ballot* after first data refinement

Law **DatRef 5** from Fig. 3.1. We can show that

$$bb = BB \Rightarrow so_far < first_skip_or_dup = bb(so_far).no = 1 \land$$
$$bb = BB \Rightarrow apaper^{\sim}(so_far) = bb(so_far).cand.$$

The data refined version of Fig. 5.6 is shown in Fig. 5.7.

Next, we refine the calculation of the local variable bb using the loop shown in Fig. 5.8. The loop invariant is:

$$\begin{array}{ll} LI_2 \mathrel{\widehat{=}} & bb = \{1..no_cands \times \{(\bot, 0)\} \Leftarrow\!\!+ \\ & \{ii \mapsto (cand, no) \mid ii \in ran(processed \lhd apaper) \land \\ & \quad cand \in (processed \lhd apaper)^{\sim}[\{ii\}] \land \\ & \quad no = card((processed \lhd apaper)^{\sim}[\{ii\}])\} \\ & \land \\ & processed \subseteq dom(apaper). \end{array}$$

Again, the main proof is easily discharged.

VAR *processed* **IN**
  $bb, processed := 1..no_cands \times \{(\perp, 0)\}, \varnothing;$
  WHILE $dom(apaper) - processed \neq \varnothing$ **DO**
    VAR *currcand*, *index* **IN**
      $currcand :\in dom(apaper) - processed$;
      $index := apaper(currcand)$;
      $bb(index).cand := currcand$;
      $bb(index).no := bb(index).no + 1$;
      $processed := processed \cup \{currcand\}$
    END
  INVARIANT LI_2
  VARIANT $card(dom(apaper) - processed)$
  END
END

Figure 5.8: Calculating intermediate variable *bb*.

We now apply data refinement to the code in Fig. 5.8. We replace an abstract paper, represented as a partial function, by a sequence of *Candidate*, $\mathbb{N}$ pairs.

$$CPaper \in iseq(Candidate \times \mathbb{N}).$$

We use GI_2 to relate these:

$$GI_2 \ \widehat{=}\ apaper = ran(cpaper).$$

Furthermore, we wish to reduce non-determinism. We introduce a new (indexing) variable, *so_far*. We replace the non-deterministic choice of the next candidate to be processed by the next in sequence of *cpaper*, i.e., $cpaper(so_far)$. The new invariant is:

$$GI_3 \ \widehat{=}\ processed = \{c \mid \exists\, ii \bullet (ii \in 1..so_far - 1 \wedge cpaper(ii).cand = c)\} \wedge so_far \in 1..no_cands + 1.$$

We can show that

$$GI_2 \wedge GI_3 \ \Rightarrow\ dom(apaper) - processed \neq \varnothing \ = \ so_far - 1 \leq card(cpaper)$$
$$\mathcal{D}[\![\ currcand :\in dom(apaper) - processed\ ]\!] \sqsubseteq currcand := cpaper(so_far).cand$$
$$\mathcal{D}[\![\ index := apaper(currcand)\ ]\!] \ \sqsubseteq\ index := cpaper(so_far).pref$$
$$\mathcal{D}[\![\ processed := processed \cup \{currcand\}\ ]\!] \ \sqsubseteq\ so_far := so_far + 1.$$

```
VAR so_far IN
    so_far := 1;
    WHILE so_far ≠ card(cpaper) DO
        VAR currcand, index IN
            currcand := cpaper(so_far).cand;
            index := cpaper(so_far).pref;
            bb(index).cand := currcand;
            bb(index).no := bb(index).no + 1;
            so_far := so_far + 1;
        END
    END
END
```

Figure 5.9: Data-refined loop for calculating *bb*

Using the data refinement rules, the loop of Fig. 5.8 (excluding the initialisation of *bb*) is data-refined by the loop of Fig. 5.9. The initialisation of *bb* to $1..no_cands \times \{(\perp, 0)\}$ is easily refined by a loop that interates i through $1..no_cands$ setting each $bb(i)$ to $(\perp, 0)$. We omit this for reasons of space.

This concludes the development of the concrete version of *Make_Ballot*, which we refer to as *C_Make_Ballot*, using algorithmic refinement before data refinement.

Make_Ballot - Data Refinement Followed by Algorithmic Refinement

We now visit the more usual approach taken in B developments. We immediately apply data refinement to the abstract specification. We then proceed with algorithmic refinement. We start at the same point and end with the same code using both styles, as discussed in Section 5.4.1. For reasons of space, we simply state the loop invariants without showing the resultant code. The development of the code should be obvious from the statement of the invariants.

We start by showing the result of data refining the operation *Make_Ballot* using gluing invariants GI_2 and GI_1. We introduce a name for the data refined *first_skip_or_dup*, called *c_first_skip_or_dup*.

$$c_first_skip_or_dup = \\ min(\{\ nn \mid nn \in 1..no_cands + 1 \wedge card(ran(cpaper)^{\sim}[\{nn\}]) \neq 1\ \}.$$

$$\begin{array}{l} cballot \longleftarrow C_Make_Ballot(cpaper) \widehat{=} \\ \textbf{PRE} \\ \quad cpaper \in iseq(Candidate \times \mathbb{N}) \\ \textbf{THEN} \\ \quad cballot := \{nn \mapsto cc \mid nn \in 1..c_first_skip_or_dup - 1) \land \\ \qquad\qquad cc \mapsto nn \in ran(cpaper)\} \\ \textbf{END}. \end{array}$$

We use the same structure as in the previous section, but with concrete variables. Similarly, we use an intermediate variable (cb) to help us build up $cballot$. The property for cb is

$$\begin{array}{l} cb = 1..no_cands \times \{\bot, 0\} \ \lhd\!\!\!+ \\ \quad \{ \ ii \mapsto (cand, no) \mid ii \in ran(ran(cpaper)) \land \\ \qquad cand \in (ran(cpaper))^{\sim}[\{ii\}] \land \\ \qquad no = card((ran(cpaper))^{\sim}[\{ii\}]) \ \}, \end{array}$$

and we introduce a loop to calculate cb using the following loop invariant:

$$\begin{array}{l} LI_4 \ \widehat{=} \ cb = 1..no_cands \times \{\bot, 0\} \ \lhd\!\!\!+ \\ \quad \{ \ ii \mapsto (cand, no) \mid ii \in ran(ran(1..so_far - 1 \lhd cpaper)) \land \\ \qquad cand \in ran(1..so_far - 1 \lhd cpaper)^{\sim}[\{ii\}] \land \\ \qquad no = card(ran(1..so_far - 1 \lhd cpaper)^{\sim}[\{ii\}] \ \} \land \\ so_far \in 1..no_cands + 1. \end{array}$$

Note the similarity with LI_2. The main difference is the use of an index so_far rather than a set $processed$. This increases the complexity of the reasoning slightly at the concrete level.

We need to calculate the initial value of cb (as with bb before) and so present the following loop invariant to calculate the initial value for cb. Note the similarity to I_3 from the previous section.

$$\begin{array}{l} LI_5 \ \widehat{=} \ 1..so_far \lhd cb = 1..so_far - 1 \times \{(\bot, 0)\} \land \\ \quad so_far \in 1.no_cands + 1. \end{array}$$

Next, we present the 'main' loop invariant, i.e., to calculate $cballot$.

$$\begin{array}{l} LI_6 \ \widehat{=} \ cballot = \{nn \mapsto cc \mid nn \in 1..so_far - 1 \land cc \mapsto nn \in ran(cpaper)\} \land \\ \quad so_far \in 1..no_cands + 1. \end{array}$$

This is very similar to LI_1. Using these loop invariants, we derive the same refined code as resulted from the previous section.

The invariants are similar, with the concrete version being slightly more complex. However, we refined to a more deterministic version in one step with the loop introduction step involving LI_4. When we applied algorithmic refinement first, (see Fig. 5.8), the first loop had more non-determinism.

5.4.2 Development of Pre_Process

Having developed the code for *C_Make_Ballot*, we move on to the higher level, i.e., the container and how the (abstract) bag of papers gets transformed into the (abstract) bag of ballots. The *Pre_Process* operation is specified in Fig. 5.4. In this section, we look at the concrete data types used for the implementation of this operation. We look at both styles of development as in Section 5.4.1.

Pre_Process - Algorithmic Refinement Followed by Data Refinement

In this section, we apply algorithmic refinement immediately and then proceed with data refinement. We are required to refine:

$$bagballots := pre_process(bagpapers).$$

The main loop invariant is:

$$\begin{array}{l} LI_7 \;\widehat{=} \\ \quad bagballots = \\ \qquad restrict\ \ (\ (map\ (\ make_ballot, restrict(bagpapers, processed)\ )\ ) \\ \qquad\qquad \{ballot \mid ballot \in Ballot \wedge card(ballot) > 0\}) \\ \qquad \wedge \\ \quad processed \subseteq dom(bagpapers). \end{array}$$

Looking at the post-condition of *Pre_Process*, we immediately introduce a loop as shown in Fig. 5.10. It can be shown that all the proof obligations can be discharged.

We now apply data refinement techniques as described in Chapter 3 Section 3.2. *bagballots* is replaced by *msetballots* of the MultisetBallots machine of Figs. 5.2 and 5.3. *bagpapers* is replaced by *msetpapers* of the Multiset-Papers machine of Fig. 5.1. We use the transformations and discharge the proof obligations. The gluing invariant between *bagpapers*, *bagballots* and

```
bagballots ⟵ Pre_Process(bagpapers) ≙
PRE bagpapers ∈ bag Paper
THEN
    VAR processed IN
        bagballots, processed := ∅, ∅;
        WHILE (dom(bagpapers) − processed) ≠ ∅ DO
            VAR pp, bb IN
                pp :∈ (dom(bagpapers) − processed) ;
                bb ⟵ Make_Ballot(pp);
                IF card(bb) > 0
                THEN
                    bagballots :=
                      bagballots <+ {bb ↦ bagballots(bb) + bagpapers(pp)}
                 END ;
                processed := processed ∪ {pp}
            END
        INVARIANT  LI_7
        VARIANT  card(dom(bagpapers) − processed)
        END
    END
END
```

Figure 5.10: Algorithmically-refined pre-processing

msetpapers, *msetballots* is

$$
\begin{array}{lrcl}
GI_4 \mathrel{\widehat{=}} & bagpaper & = & items\{nn \mapsto ap \mid ran(msetpapers(nn)) = ap\} \quad \wedge \\
 & bagballots & = & items(msetballots)
\end{array}
$$

One of the main transformations is that of the loop guard. Given

$$GI_5 \mathrel{\widehat{=}} processed = \{ \ ap \mid \exists\, cp \bullet (cp \in mprocessed \wedge ap = ran(cp)) \ \},$$

we can show that

$$
\begin{array}{l}
GI_4 \wedge GI_5 \quad \Rightarrow \\
\quad dom(Papers) - processed \neq \varnothing \ = \ ran(msetpapers) - mprocessed \neq \varnothing.
\end{array}
$$

Within the loop body we use:

$$
\begin{array}{lcl}
GI_6 & \mathrel{\widehat{=}} & pp = ran\ mp \ \wedge \\
 & & bagpapers(pp) \ = \ card\{nn \mid msetpapers(nn) = pp\} \ \wedge \\
 & & bb = cb.
\end{array}
$$

The transformations on the statements and expressions using gluing invariant $GI_4 \wedge GI_5 \wedge GI_6$ are then as follows:

$$
\begin{array}{l}
\mathcal{D}[\![\ bagballots := \varnothing\ ]\!] \ \sqsubseteq \ msetballots := \varnothing \\
\mathcal{D}[\![\ pp :\in dom(bagpapers) - processed\ ]\!] \ \sqsubseteq \\
\qquad mp :\in ran(msetpapers) - mprocessed \\
\mathcal{D}[\![\ bb \longleftarrow Make_Ballot(pp)\ ]\!] \ \sqsubseteq \ cb \longleftarrow C_Make_Ballot(mp) \\
\mathcal{D}[\![\ bagballots := bagballots \mathbin{<\!+} \{bb \mapsto bagballots(bb) + bagpapers(pp)\}\ ]\!] \ \sqsubseteq \\
\qquad MultiInsert(cb, card(\{nn \mid msetpapers(nn) = mp\})) \\
\mathcal{D}[\![\ processed \cup \{pp\}\ ]\!] \ \sqsubseteq \ mprocessed \cup \{mp\}.
\end{array}
$$

This leads to the following program:

```
msetballots ⟵ C_Pre_Process(msetpapers) ≙
PRE msetpapers ∈ multisetPapers THEN
    VAR mprocessed IN
        msetballots, mprocessed := ∅, ∅;
        WHILE ran(msetpapers) − mprocessed ≠ ∅ DO
            VAR mp, cb IN
                mp :∈ ran(msetpapers) − mprocessed ;
                cb ⟵ C_Make_Ballot(mp) ;
                IF card(cb) > 0 THEN
                    MultiInsert(cb, card{nn | msetpapers(nn) = mp})
                END ;
                mprocessed := mprocessed ∪ {mp};
            END
        END
    END
END.
```

Next, we refine the nondeterministic selection of *mp* in the loop body and become more specific about which member we choose. The refinement step replaces *mprocessed* by *iter* using the following gluing invariant:

$$GI_7 \;\widehat{=}\; mprocessed = ran(1..iter - 1 \lhd msetpapers).$$

We can show that

$$GI_7 \;\Rightarrow\; ran(msetpapers) - mprocessed \neq \varnothing \;=\; iter \leq card(msetpapers).$$

The refined program is defined in terms of the multiset machines (Fig. 5.1 and Figs. 5.2 and 5.3):

```
MakeEmpty;
Start;
WHILE iterNotAtEnd DO
    mp ⟵ GetNext;
    cb ⟵ C_Make_Ballot(mp);
    IF card(cb) > 0 THEN
        MultiInsert(cb, Count(mp))
    END
END.
```

This concludes the first path for *Pre_Process*.

CONSTANTS
setup_first_count

PROPERTIES
$setup_first_count \in bag\ Ballot \rightarrow VoteMass$
$\wedge$
$\forall (bagballots, cand).$
$(bagballots \in bag\ Ballot \wedge$
$cand \in \{cc \mid bal \in dom\ bagballots \wedge bal(1) = cc\} \Rightarrow$
$setup_first_count(bagballots)(cand)(1) =$
$restrict(bagballots, \{bb \mid bb(1) = cand\})$

OPERATIONS
$Setup_First_Count \mathrel{\widehat{=}}$
BEGIN
$vm := setup_first_count(bagballots)$
END

Figure 5.11: Abstract Specification of *Setup_First_Count*

5.5 Case Study 2 - Setting up First Count

The second case study looks at setting up the first count. For this case study, we use state variables and the main operation is parameterless. The specification is written in Fig. 5.11 and is part of a machine that has as state the variables *vm* and *bagballots*. The setting up of the first count entails associating each (validated) ballot with that ballot's first preference candidate at count one.

The abstract *vm* is a *VoteMass*. *VoteMass* is the (abstract) structure type to hold the details of the counts. It is defined as:

$$VoteMass \mathrel{\widehat{=}} Candidate \mathrel{+\!\!\!\rightarrow} (\mathbb{N} \mathrel{+\!\!\!\rightarrow} bag\ Ballot)$$

bagballots is of type *bag Ballot* and contains the validated ballots from the previously described pre-processing.

The first step is to immediately introduce a loop with the following Loop

$$
\begin{array}{l}
\mathit{Setup_First_Count} \mathrel{\widehat{=}} \\
\textbf{VAR}\ \ \mathit{processed}\ \textbf{IN} \\
\quad \mathit{vm}, \mathit{processed} := \varnothing, \varnothing; \\
\quad \textbf{WHILE}\ \ (\mathit{dom}(\mathit{bagballots})\ -\ \mathit{processed}) \neq \varnothing\ \textbf{DO} \\
\quad\quad \textbf{VAR}\ \mathit{bb}\ , \mathit{cand}\ \textbf{IN} \\
\quad\quad\quad \mathit{bb} :\in\ (\mathit{dom}(\mathit{bagballots}) - \mathit{processed})\ ; \\
\quad\quad\quad \mathit{cand} := \mathit{bb}(1); \\
\quad\quad\quad \mathit{vm}(\mathit{cand})(1) := \mathit{vm}(\mathit{cand})(1)\ \cup \\
\quad\quad\quad\quad\quad\quad \{\mathit{bb} \mapsto \mathit{bagballots}(\mathit{bb})\} \\
\quad\quad\quad \mathit{processed} := \mathit{processed} \cup \{\mathit{bb}\} \\
\quad\quad \textbf{END} \\
\quad \textbf{INVARIANT}\quad \mathit{LI}_8 \\
\quad \textbf{VARIANT}\quad \mathit{card}(\mathit{dom}(\mathit{bagballots}) - \mathit{processed}) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figure 5.12: First Loop Introduction in *Setup_First_Count*

Invariant:

$$
\begin{array}{l}
\mathit{LI}_8 \ \mathrel{\widehat{=}} \\
\forall\ \mathit{cand} : \mathit{allcandidates}\ \bullet \\
\quad \mathit{vm}(\mathit{cand})(1)\ =\ \mathit{restrict}(\mathit{bagballots}, \{\mathit{bb}\ \mid\ \mathit{bb}(1) = \mathit{cand}\ \wedge\ \mathit{bb} \in \mathit{processed}\})\ \wedge \\
\mathit{processed} \subseteq \mathit{dom}(\mathit{bagballots}).
\end{array}
$$

This leads to the program in Fig. 5.12. The proof-obligations are discharged but not shown as they are standard.

Next, we proceed with data refinement. We proceed with data refinement in two stages, for clarity. Firstly, we data refine our *vm* to a less abstract *vmpos*. The second data refinement step is to data refine *vmpos* to a more concrete *cvm*.

We proceed with the first data refinement. *vmpos* has the following type:

$$\mathit{VoteMassPos} \mathrel{\widehat{=}} \mathit{Candidate} \twoheadrightarrow\!\!\!\!\!\!+\ (\mathbb{N} \twoheadrightarrow\!\!\!\!\!\!+\ \mathbb{PN})$$

vmpos holds the positions of (or pointers to) the *ballots* in the *msetballots* instead of the actual *ballots* from before. So the gluing invariant is as follows:

$$
\begin{array}{l}
\mathit{GI}_8 \ \mathrel{\widehat{=}} \\
\forall(\mathit{cand}, \mathit{count}).(\mathit{cand}\ \in\ \mathit{dom}\ \mathit{vm} \wedge \mathit{count}\ \in\ \mathit{dom}(\mathit{vm}(\mathit{cand}))\ \bullet \\
\quad \mathit{vmpos}(\mathit{cand})(\mathit{count}) = \\
\quad\quad \{\mathit{place}\ \mid\ \mathit{msetballot}(\mathit{place}) \in \mathit{dom}\ \mathit{vm}(\mathit{cand})(\mathit{count})\}
\end{array}
$$

The next gluing invariant GI_9 specifies a new *cprocessed*. Whereas *processed* is the set of *ballot*s that have been dealt with, *cprocessed* is corresponding set of positions (in *msetballots*) of the *ballot*s that have been dealt with. So the gluing invariant is:

$$GI_9 \;\widehat{=}\; cprocessed = \{nn \mid msetballots(nn) \in processed\}$$

We distribute this set of data refinements across the code of the program of Fig. 5.12.

$$\begin{array}{l}
\mathcal{D}[\![\ processed := \varnothing\ ]\!] \quad \sqsubseteq \quad cprocessed := \varnothing \\
\mathcal{D}[\![\ vm := \varnothing\ ]\!] \quad \sqsubseteq \quad vmpos := \varnothing \\
\mathcal{D}[\![\ dom\ bagballots\ -\ processed \neq \varnothing\ ]\!] \quad \sqsubseteq \\
\qquad\qquad\qquad 1\,..\,card(msetballots)\ -\ cprocessed\ \neq\ 0 \\
\mathcal{D}[\![\ bb\ :\in\ (dom\ bagballots\ -\ processed)\ ]\!] \quad \sqsubseteq \\
\qquad\qquad\qquad newpos\ :\in\ (1\,..\,card(msetballots)\ -\ cprocessed); \\
\qquad\qquad\qquad bb := msetballots(newpos) \\
\mathcal{D}[\![\ vm(cand)(1)\ \cup\ \{bb \mapsto bagballots(bb)\}\ ]\!] \quad \sqsubseteq \\
\qquad\qquad\qquad vmpos(cand)(1) := vmpos(cand)(1)\ \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{nn\ \mid\ msetballots(nn) = bb\} \\
\mathcal{D}[\![\ processed := processed\ \cup\ \{bb\}\ ]\!] \quad \sqsubseteq \\
\qquad\qquad\qquad cprocessed := cprocessed \cup \{nn \mid msetballots(nn) = bb\}
\end{array}$$

The resultant program is given in Fig. 5.13 and is part of a machine which includes *vmpos* and *msetballots* as state. This program is less abstract, but still contains a non-deterministic choice of the next ballot to be processed. This non-determinism can be removed by the use of a further data refinement, introducing a *counter* called *iter*, so the next ballot to be chosen is the next available in *msetballots*.

$$\begin{array}{rl}
GI_{10} \;\widehat{=}\; & cprocessed = 1\,..\,iter - 1 \wedge \\
& iter \in 1..card(msetballots) + 1
\end{array}$$

We distribute this data refinement across the code of the program of Fig. 5.13

$Setup_First_Count \mathrel{\widehat=}$
VAR *cprocessed* **IN**
  $vm, cprocessed := \varnothing, \varnothing;$
  WHILE $(1 \,..\, card(msetballots) \;-\; cprocessed) \neq \varnothing$ **DO**
    VAR *bb* , *cand* , *newpos* **IN**
      $newpos :\in (1 \,..\, card(msetballots) - cprocessed);$
      $bb := msetballots(newpos)\;;$
      $cand := bb(1);$
      $vmpos(cand)(1) := vm(cand)(1) \;\cup$
        $\{nn \mid msetballots(nn) = bb\};$
      $cprocessed := cprocessed \cup \{nn \mid msetballots(nn) = bb\}$
    END
  END
END

Figure 5.13: First Data Refinement in *Setup_First_Count*

as follows:

$$\mathcal{D}[\![\; cprocessed := \varnothing \;]\!] \quad \sqsubseteq \quad iter := 1$$
$$\mathcal{D}[\![\; 1 \,..\, card(msetballots) - cprocessed \;\neq\; \varnothing \;]\!] \quad \sqsubseteq$$
$$iter \;\leq\; card(msetballots)$$
$$\mathcal{D}[\![\; newpos :\in (1 \,..\, card(msetballots) - cprocessed);$$
$$bb \;:=\; msetballots(newpos) \;]\!] \quad \sqsubseteq$$
$$newpos := iter;$$
$$bb \;:=\; msetballots(newpos) \quad \sqsubseteq$$
$$bb \;:=\; msetballots(iter)$$
$$\mathcal{D}[\![\; cprocessed := cprocessed \cup \{nn \mid msetballots(nn) = bb\} \;]\!] \quad \sqsubseteq$$
$$iter := iter + card(\{nn \mid msetballots(nn) = bb\})$$

This leads to the less non-deterministic, data refined program as written in Fig. 5.14.

Our next data refinement introduces our final concrete data structure. It uses sequences. These sequences will be implemented using linked lists. The concrete data structure *cvm* is of the type *C_VoteMass*, defined as follows:

$$C_VoteMass \mathrel{\widehat=} seq(Candidiate \;\times\; seq(\mathbb{N} \;\times\; seq\,\mathbb{N})\;)$$

Associated with each candidate is a sequence, each element of which contains a count number and the sequence of positions of ballots associated

```
Setup_First_Count ≙
VAR iter IN
    vmpos, iter := ∅, 1;
    WHILE (iter ≤ card(msetballots)) DO
        VAR bb, cand IN
            bb := msetballots(iter);
            cand := bb(1);
            vmpos(cand)(1) := vmpos(cand)(1) ∪
                         {nn | msetballots(nn) = bb}
            iter := iter + card({nn | msetballots(nn) = bb})
        END
    END
END
```

Figure 5.14: Second Data Refinement in *Setup_First_Count*

with this candidate at this count. As it may not be obvious how to access each of the elements of *cvm*, we use the following (where i is the position in the sequence of our candidate of interest):
$cvm(i).cand$ yields the candidate of interest.
$cvm(i).countlist$ yields the sequence or 'list' of counts for our candidate of interest.
$cvm(i).countlist(j).countnumber$ yields the count number which is at position j of the sequence associated with our candidate of interest.
$cvm(i).countlist(j).countballots$ yields the sequence of positions of the ballots associated with the candidate of interest at position j of the sequence.
The gluing invariant between *vmpos* and *cvm* is as follows:

$$
\begin{aligned}
GI_{11} \;\widehat{=}\; & \forall(i, j, cand, count).(i \in 1\,..\,no_cands \;\wedge \\
& \quad j \in 1\,..\,no_cands \;\wedge \\
& \quad cand = cvm(i).cand \;\wedge \\
& \quad count \in dom(ran\ cvm(i).countlist(j)) \wedge \\
& vmpos(cand)(count) = ran(ran\ cvm(i).countlist(j)\ )\ (countnumber)\ )
\end{aligned}
$$

We introduce the machine ConcreteVoteMass in Figs. 5.15 and 5.16 and use a strategy similar to that used in the Multiset machines from before. We define the machine with operations which will be useful to us for this problem. The next data refinement is then to rewrite the operation *Setup_First_Count* in terms of the ConcreteVoteMass machine. This machine, which is based

on sequences, would be implemented using linked lists, using techniques as described in [20]. This refinement path is not shown.

In the ConcreteVoteMass machine, we use the mathematical function *makeseq* which returns an injective sequence from a set as specified in the following way:

$$\forall\, ss.(ss \in iseq\ \mathbb{N} \wedge\ makeseq(ran\ ss) = ss\ )$$

This leads to an implementation defined in terms of the MultisetBallot machine (Figs. 5.2 and 5.3) and the ConcreteVoteMass Machine (Figs. 5.15 and 5.16). This implementation is shown in Fig. 5.17.

5.6 Conclusions

The two case studies completed above have illustrated techniques discussed in earlier chapters. Specifically, we have employed the technique of applying algorithmic refinement as a first step throughout. This has led to clear solutions and straightforward proofs in most circumstances. We also applied the technique of data refining interfaces as needed because of some of the operations' interfaces being based on abstract data types.

Interface refinement was used on both of the high-level operations in the case studies, i.e. *Pre_Process* and *Setup_First_Count*. We applied a further interface refinement in the first case study - *Pre_Process*. A clue as to when we need interface refinement within the operations can be found in the abstract specification. Well structured specifications are clear and hide complex parts by appropriately naming small parts of the specifications. This can be implemented by the use of mathematical functions (in the CONSTANTS.. PROPERTIES part of the MACHINE), e.g. the *make_ballot* function led to the *Make_Ballot* procedure. When this non-flat structure is used, this is our hint that the implementation of overall specification may have an internal call to an operation. Because the mathematical function has been defined in terms of data that will have to change, we deal with them as *procedures* as defined in Section 4.2.

Note the structure of both our (high level) abstract specifications in the case studies, in Figs. 5.4 and 5.11. In the case of *Pre_Process*, the function *pre_process* calls another function, *make_ballot*. An operation *Make_Ballot* (a *procedure*) uses the function *make_ballot*. The operation is refined, using interface refinement.

```
MACHINE ConcreteVoteMass
VARIABLES cvm
INVARIANTS cvm ∈ C_VoteMass
INITIALISATION Initcvm
OPERATIONS
Initcvm ≙
    ANY ss WHERE
        dom (ran(s) ) = allcandidates ∧
        card(s) = card(allcandidates)
    THEN
        cvm := ss
    END;

bb ⟵ CountExists(count, cand) ≙
    PRE
        count ∈ ℕ ∧ cand ∈ allcandidates
    THEN
        bb := BOOL(count ∈ dom(ran (ran cvm)(cand))))
    END;

pos ⟵ FindCandIndex(cand) ≙
    PRE
        cand ∈ allcandidates
    THEN
        ANY ii WHERE
            cvm(ii).cand = cand
        THEN
            pos := ii
        END
    END;

pos ⟵ FindCountIndex(count, cand) ≙
PRE
    count ∈ ℕ ∧ cand ∈ allcandidates
THEN
    ANY ii WHERE
        cvm(FindCandIndex(cand)).countlist(ii).count = count
    THEN
        pos := ii;
    END
END;
```

Figure 5.15: ConcreteVoteMass Machine

```
InsertNewCount(count, cand) ≙
PRE
    count ∈ ℕ ∧ cand ∈ allcandidates ∧
    ¬CountExists(count, cand)
THEN
    ANY candindex WHERE
        candindex = FindCandIndex(cand)
    THEN
        cvm(candindex).countlist :=
            cvm(candindex).countlist^[count ↦ [ ] ]
    END
END

InsertNewBallots(count, cand, startpos, number) ≙
PRE
    count ∈ ℕ ∧ cand ∈ allcandidates ∧
    startpos ∈ 1 .. card(msetballots) ∧
    startpos + number ≤ card(msetballots) ∧
    CountExists(count, cand)
THEN
    ANY candindex, countindex, ii WHERE
        candindex = FindCandIndex(cand) ∧
        countindex = FindCountIndex(cand, count)
    THEN
        cvm(candindex).countlist(countindex).countballots :=
        cvm(candindex).countlist(countindex).countballots
        ^makeseq(startpos .. startpos + number − 1)
    END
END;

DEFINITIONS
    C_VoteMass ≙ seq(Candidate × seq(ℕ × seq ℕ))
```

Figure 5.16: ConcreteVoteMass Machine.. contd.

```
SetupFirstCount ≙
    Initcvm;
    Start;   /* Go to start of Ballots */
    WHILE iterNotAtEnd DO
        VAR bb, cand IN
            bb ⟵ CurrentBallot;
            cand := bb(1);
            IF CountExists(1, cand)
            THEN
                InsertNewBallot(1, cand, iter, num);
            ELSE
                InsertNewCount(1, cand);
                InsertNewBallot(1, cand, iter, num)
            END
            MoveToNextBallot;
        END
    END
```

Figure 5.17: Implementation of *Setup_First_Count*

In the case of *Setup_First_Count*, the specification is flat. No further operations are used, so no further interface refinement is required.

It may be noted that the overall structure of the process is cleaner and neater in this chapter than in the earlier Chapter 2 where a case study was examined. This may be due to the more appropriate technique having been used. It is probably more due to the fact that the work done in this chapter was attempted later than that of the earlier chapter. Therefore, more experience was brought into the work of this chapter.

Chapter 6

Conclusions

6.1 Conclusions

We have described, using examples, how to apply data refinement after algorithmic refinement in B. We have used laws on distribution of data refinement to implement the former approach based on [8, 16, 22, 24, 32].

We make some comparisons, albeit intuitive, between developments using both the standard approach of data refinement first and *vice versa*. Some tentative conclusions on these comparisons are:

- Invariants tend to be simpler when the loop is introduced immediately, i.e. before data refinement. This leads to slightly easier reasoning when proving the loop.

- In some cases, the removal of non-determinism in the 'choice of next element to be processed' in the loop was more elegant in the case of algorithmic refinement first. This may have been due to that fact that data refinement was introduced in two stages, (after loop introduction) firstly maintaining, and then eliminating non-determinism. The 'two–stage proof' was not as complex as the 'all-in-one' version of the data refinement first.

We suggest that this developmental style of algorithmic refinement before data refinement is (at least) worthy of examination. We suggest that this approach be seen as an alternative to the more standard approach. It is sometimes (but not always) appropriate. It may not be appropriate, for example, when the algorithmic structure is determined more by the concrete data structures than the abstract data structures. Also, it may often be useful to mix approaches, i.e., perform some algorithmic refinement then some

data refinement, more algorithmic refinement etc. More work needs to be done to formulate a set of heuristics on when either approach is appropriate.

There are many ways to data refine an abstract data type, depending on style and priorities of implementation. In general there are fewer abstract data types used in developments than concrete data types. Developers should find that it is easier and quicker to gain proficiency in the introduction of algorithmic structures on the fewer number of abstract data types than in the introduction of same with concrete data types. Loop introduction, the proof obligations of which are difficult to discharge, is an example of where the developer can build up a useful set of solution patterns more quickly on loops involving abstract data types than on loops working on concrete data types.

At present, neither the B-Toolkit nor Atelier-B fully support algorithmic refinement before implementation stage. The extensions suggested are to support this activity by incorporating the Data Refinement Laws (e.g. those mentioned in Fig. 3.1) and allowing algorithmic refinement during the refinement stage.

Presently, refinement (sensibly) must be interface-preserving. As discussed in Chapter 4 and illustrated in the case studies, however, in some cases it is necessary. A rule for allowing us to introduce a correct refinement step involving interface refinement is presented in Section 4.2. This allows us to progress soundly. However, as this rule is not presently implemented in the B-Toolkit, we present a workaround for checking non interface-preserving refinements in the B-Toolkit in Section 4.5.

We used C++'s S.T.L. during development of the case studies. The motivation for its use is presented in Section 5.2. Whereas the limited nature of the case studies meant that we did not fully develop the idea, we found that the resulting efficiency from using S.T.L. and its data types was a good enough reason to explore the area.

We used the idea of overloaded dot notation in Section 5.4.1. This allows us to access elements of a pair conveniently and is syntatic sugar for the Z-like *fst* and *snd* functions. We found this to be useful.

During both case studies, we used the approach of early in the development guessing a possible implementation. We use this to guide us through the development path as discussed in Section 1.7.

We discussed the main technical differences between Morgan's Refinement Calculus and the B Method in Section 1.7. We now draw some comparisons between each technique.

- Firstly, let us look at the B Method. The developer can see the en-

tire operation together, albeit with possibly some operation/function calls, no matter at what stage the development is. The context is always clear. However, when using Morgan's Refinement Calculus, the program is broken up (using sequential composition) and each piece is separately refined. When developing a program of any reasonable complexity, this leads to many different paths. It becomes quite difficult to keep track of the paths. It is also tricky to have an overview on the development process throughout. The development of Section 2.3.3 is a short one but it is not easy to see the development path at a glance. This is particularly problematic when novice users are using this approach.

- In the B Method, it is possible (albeit not necessary or usual) to go directly, in one step from specification to implementation. The proof-obligations would then have to be discharged (assuming automatic generation of proof obligations by the B-Toolkit). This is not a practical approach for anything but the simplest of systems as usually such proof obligations would be difficult to discharge. However it does nicely illustrate the difference between the two approaches. It is completely up to the user to drive the entire process in the case of Morgan's Refinement Calculus.

- We have mentioned tool support in our presentation of motivations for moving towards using the B Method. There is no commercially available tool support for Morgan's Refinement Calculus. The availability of tool support is a significant advantage for the B Method. It is hard to envisage any real–world systems being developed without such significant tool support.

6.2 Future Work

The work as presented leads us to suggest a number of different areas for future work.

In this work, we have presented a new approach to refinement, i.e. algorithmic refinement before data refinement. We have not thoroughly examined when this approach is better or more appropriate. (We discuss it at an intuitive level). We have not explored under what criteria 'better' or 'more appropriate' could be decided. It would be useful to examine this area more closely. Perhaps an examination of the types and numbers of proof-obligations generated in each case would be a starting point. This

could lead to a quantifiable way of judging the relative merits of different developments which would be very useful.

We have used S.T.L. as the target code and environment in the Case Studies. We discussed why we decided on S.T.L. in Chapter 5. The main benefit of using S.T.L. is that the algorithms and data structures work well to give us efficient implementations. We make the (sensible) assumption that whereas C++ including S.T.L. are not formally proven, we assume them to be correct as they are widely used and thus exhaustively tested.

Future work in this area would be to specify all of S.T.L. libraries in the B Method, so that all the S.T.L. is available to the formal practitioner easily. The development of 'design patterns' of problems which lead to S.T.L. implementations could lead to re-use of some specification - to - code, bringing with it with the usual benefits of re-use of proofs, etc.

In this work, we started using Z specification and developing the implementation using Morgan's Refinement Calculus [23]. As discussed, this did not work well. We then moved on to work with the B Method [1] which worked better and which we used for the remainder of the work. It would be interesting to compile a comprehensive description of the differences between these two approaches to refinement. An examination of when each is more appropriate would also be interesting.

Our strategy when developing sub-systems was to look at the specific problem and develop that. Another approach would be to solve generic problems or subsystems and re–use these generic solutions to particular problems. We have looked at one generic problem, that of 'selective mapping' and shown the generic solution, in Section 2.6. Further work could be done on abstracting the generic patterns of more of the problems that we have dealt with in this work. This generic problems could then be solved, resulting in useful problem patterns with available solutions.

The original plan of this work was to implement the counting of votes in an electoral system, the rules of which are presented in Appendix A. The specification is presented, both in the Z notation (Appendix B) and using the B Method (Appendix D).

As the work progressed, we moved from our original paradigm of Z specification followed by Morgan's Refinement Calculus to the B Method. Although we were satisfied that the B Method was the appropriate choice for development, its use led us to examine a number of areas, the results of which are presented in Chapter 3 and Chapter 4.

This meant that there was a shift in emphasis away from 'pure' implementation. Consequently, the full system as specified has not been fully implemented. An obvious area of future work is to fully implement the

voting system. This would be interesting as:

- a number of very interesting challenges arose from a careful examination of the issues arising out of the development of what was a small part of this real–life system. The development of the remainder of the system and similar careful examination could, we believe, raise more interesting points.

- as a system for counting votes is a mission–critical system, Formal Methods is a very appropriate developmental technique. It would act as a useful real-life example. It may have commercial value as currently there are plans to computerise part of the Irish General Election.

Appendix A

Waterford Institute of Technology Academic Council Election Count Rules

A.1 Election Procedures of Academic Members to the Academic Council

1. The number of members to be elected shall be thirteen.
2. There shall be a minimum of 40% or 6 candidates from each gender. (We call this the 'gender constraint'). This is a legal requirement.
3. There shall be a minimum of two persons from each of the four schools elected subject to there being sufficient candidates. (We call this the 'school constraint')
4. There shall be a minimum of one person from the Department of Adult Education elected subject to there being sufficient candidates. (We call this the 'department constraint')
5. In considering the counting of votes, the rules as set down in Rules for Academic Council Election (Academic Members) will be used. An abridged form is seen below.

A.2 Rules for Academic Council Election (Academic Members) (Abridged Form)

1. All valid papers are grouped by first preference votes and candidates are ordered in descending order of first preferences. Each vote is given a weight of 1000.

2. The quota is calculated
   $quota =$
   $(number\ of\ valid\ votes * 1000)/(number\ of\ vacancies + 1) + 1$

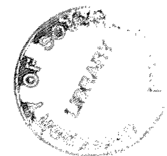
3. If, at the end of any count, a candidate has a total weight of votes greater than the quota, check should that candidate be elected (see Section A.3). If this is not allowed, then exclude the candidate according to rule 7. If it is allowed to elect the candidate, deem the candidate to be elected and distribute the candidate's surplus in the following manner

4. If the candidate's votes come from first preferences only, then transfer all votes according to rule 6.

5. If the candidates votes come from a mixture of first preferences and transfers, then transfer the last bundle to be transferred according the rule 6.

6. Calculate the total weight of transferable votes. If the total weight of transferable votes is greater then surplus, transfer each vote in the bundle (to the continuing candidate indicated as the next available preference) with a decreased weight,
   $newweight =$
   $old\ weight * (surplus/total\ weight\ of\ transferable\ papers)$
   If the total weight of transferable votes is less than or equal to the surplus, transfer all votes in bundle with same weight as before.

7. If, at the end of any count, no candidate has reached the quota and there are more continuing candidates than vacancies, a candidate must be excluded. Working backwards from the candidates with the lowest vote weight, check can the candidate be excluded (see Section A.3). If this candidate's exclusion is allowed by RuFFE, then distribute the transferable votes, leaving the weights unchanged. If the candidate cannot be excluded (see Section A.3), check the next lowest candidate and so on. For candidates that cannot be excluded, do not exclude,



do not elect. These candidates will eventually be elected, without necessarily reaching the quota.

8. STOP when either the number of elected candidates = 13 or (more likely) the number of elected candidates + number of continuing candidates = 13. At this point all continuing candidates may be deemed to be elected.

A.3 Rule For Election or Exclusion (RuFEE)

For this rule, a candidate is seen as belonging to a subset of candidates, either elected, continuing or excluded.

A.3.1 Election

When a candidate reaches or exceeds the surplus, before deeming that candidate to be elected, it must be checked that the following condition holds:

If the candidate is elected and this brings the number elected to n, there is a subset of the continuing candidates, of size 13 - n which, if elected will ensure that the set of 13 elected candidates will obey the gender, school and department constraints.

If this is the case, then the candidate is deemed to be elected.

If this is not the case, the candidate will not be elected as his/her election will disallow the possibility of the 13 candidates including him/her ever obeying the gender, school, and department constraints. Furthermore, the candidate is excluded and his/her votes will be transferred as per rule 7 (abridged rules).

A.3.2 Exclusion

If a candidate needs to be excluded in order to proceed (there being no surplus available), starting at the lowest candidate, check that the following condition holds:

If the candidate is removed from the set of continuing candidates and placed in the set of excluded candidates, there is a subset of continuing candidates, of size 13 - n (n = the number if elected candidates) that, if elected, will ensure that the set of 13 elected will obey the gender, school and department constraints.

If this condition holds, then exclude the candidate and proceed to transfer the candidate's votes as per rule 7 (abridged rules).

If this condition does not hold this means that it will be necessary to elect this candidate (eventually) to ensure that the final set of elected candidates obey the gender and school constraints. Do not exclude the candidate. Do not elect the candidate. The candidate will be elected without necessarily reaching the quota.

If because of this rule, the lowest candidate cannot be excluded, proceed to find the next lowest candidate and apply the rule to check if this candidate can be excluded. If not go to the next lowest candidate and so on.

A.4 Rules for Academic Council Election (Academic Members)

1. The Academic Council election returning officer shall reject any ballot papers that are invalid.

2. The Academic Council election returning officer shall then ascertain the number of first preferences recorded on the ballot papers for each candidate, and shall then arrange the candidates on a list (hereinafter called the order of preferences) in the order of the number of first preferences recorded for each candidate, beginning with the candidate for whom the greatest number of first preferences is recorded. If the number of first preferences recorded for any two or more candidates (hereinafter called 'equal candidates') is equal, the Academic Council election returning officer shall ascertain the number of second preferences recorded on all the ballot papers for each of the equal candidates, and shall arrange the equal candidates as amongst themselves on the order of preferences in the order of the second preferences recorded for each such candidate, beginning with the candidate for whom the greatest number of second preferences is recorded. If the number of first and second preferences recorded for any two or more equal candidates is equal, the Academic Council election returning officer shall, in like manner, ascertain the number of third preferences recorded on all the ballot papers for each of such last-mentioned equal candidates, and arrange such candidates on the order of preferences accordingly, and so on until all the candidates are arranged in order on the order of preferences. If the number of first, second, third, and all other preferences recorded for any two or more equal candidates is equal the Academic Council election returning officer shall determine by lot the order in which such candidates are to be arranged on the order of

preferences.

3. The Academic Council election returning officer shall then arrange the valid ballot papers in parcels, according to the order of preferences.

4. For the purpose of facilitating the processes prescribed by these Rules, each valid ballot paper shall be deemed to be of the value of one thousand.

5. The Academic Council election returning officer shall then count the number of ballot papers in each parcel, and in accordance with the preceding Rule credit each candidate with the value of the valid ballot papers on which a first preference has been recorded for such candidate.

6. The Academic Council election returning officer shall then add together the values in all the parcels and divide the full total value by a number exceeding by one the number of vacancies to be filled. The result increased by one, any fractional remainder being disregarded, shall be the value sufficient to secure the return of a candidate. This value is in this Schedule called the 'quota'.

7. If, at the end of any count or at the end of the transfer of any parcel, or sub-parcel of an excluded candidate or of a candidate deemed not to be a continuing candidate, the value credited to a candidate is equal to or greater than the quota, that candidate shall, subject to the provisions of the subsequent Rules and RuFEE(Rule for Election or Exclusion), be deemed to be elected.

8. If at the end of any count the value credited to a candidate is greater than the quota and the election of the candidate obeys RuFEE, the surplus of the candidate(in this Rule referred to as the elected candidate) shall be transferred to the continuing candidate or candidates indicated on the voting papers in the parcel or sub-parcel of the elected candidate according to the next available preferences recorded thereon, and the following provisions shall apply to the making of such transfer:

9. If the value credited to the elected candidate arises out of original votes only, the Academic Council election returning officer shall examine all the ballot papers in the parcel of the elected candidate and shall arrange the transferable papers therein in sub-parcels according to the next available preferences recorded thereon and shall make a separate sub-parcel of the non-transferable papers;

(a) If the value credited to the elected candidate arises partly out of original and partly out of transferred votes or out of transferred votes only, the Academic Council election returning officer shall examine the ballot papers contained in the sub-parcel last received by the elected candidate and shall arrange the transferable papers therein in further sub-parcels according to the next available preferences recorded thereon and shall make a separate sub-parcel of the non-transferable papers:

(b) In either of the cases referred to in the foregoing sub-paragraphs (a) and (b) the Academic Council election returning officer shall ascertain the number of ballot papers and their total value in each sub-parcel of transferable papers and in the sub-parcel of non-transferable papers;

(c) If the total value of the papers in all the sub-parcels of transferable papers is equal to or less than the said surplus, the Academic Council election returning officer shall transfer each sub-parcel of transferable papers to the continuing candidate indicated thereon as the voter's next available preference, each paper being transferred at the value at which it was received by the elected candidate, and where the said total value is less than the said surplus) the non-transferable papers shall be set aside as not effective. at a value which is equal to the difference between the said surplus and the said total value;

(d) If the total value of the papers in all the sub-parcels of transferable papers is greater than the said surplus, the Academic Council election returning officer shall transfer each paper in such sub-parcel of transferable papers to the continuing candidate indicated thereon as the voter's next available preference, and the value at which each paper shall be transferred shall be ascertained by dividing the surplus by the total number of transferable papers, fractional remainders being disregarded except that the consequential loss of value shall be noted on the result sheet;

(e) A surplus which arises on the completion of any count shall be dealt with before a surplus which arises at a subsequent count;

(f) When two or more surpluses arise out of the same count, the largest shall be first dealt with and the others shall be dealt with in the order of their magnitude;

(g) If two or more candidates have an equal surplus arising out of the same count, the surplus of the candidate credited with the

greatest value at the earliest count at which the values credited to those candidates were unequal shall be first dealt with, and where the values credited to such candidates were equal at all counts, the Academic Council election returning officer shall deal first with the surplus of the candidate who is highest in the order of preferences, subject to RuFEE.

10. (a) If at the end of any count no candidate has a surplus and one or more vacancies remain unfilled, the Academic Council election returning officer shall exclude the candidate (in this Rule referred to as the excluded candidate) then credited with the lowest value subject to RuFEE and shall transfer his/her papers to the continuing candidates respectively indicated on the ballot papers in the parcel or sub-parcels of the excluded candidate as the voter's next available preference, and shall credit such continuing candidates with the value of the papers so transferred, and the following provisions shall apply to the making of such transfer:

    (b) The parcel containing original votes shall first be transferred, the transfer value of each paper being one thousand;

    (c) The sub-parcels containing transferred votes shall then be transferred in the order in which and at the value of which the excluded candidate obtained them;

    (d) For the purpose of determining whether a candidate is a continuing candidate the transfer of each parcel or sub-parcel shall be regarded as a separate count;

    (e) In the transfer of each parcel or sub-parcel, a separate sub-parcel shall be made of the non-transferable papers which shall be set aside at the value at which the excluded candidate obtained them

    (f) If, when a candidate has to be excluded under this Rule, two or more candidates are each then credited with the same value and are lowest regard shall be had to the total value of original votes credited to each of those candidates and the candidate with the smallest such total value shall be excluded, and where such total values are equal regard shall be had to the total value of votes credited to each of those candidates at the earliest count at which they had unequal values, and the candidates with the smallest such total value at that count shall be excluded, and if those candidates were each credited with the same total value of

votes at all counts that one of those candidates who is lowest in the order of preferences shall be excluded, subject to RuFFE.

11. On every transfer made under these Rules, each sub-parcel of papers transferred shall be placed on top of the parcel or sub-parcel (if any) of papers of the candidate to whom the transfer is made and that candidate shall be credited with the value ascertained in accordance with these Rules of the papers so transferred to him/her.

12. (a) If at the end of any count the number of candidates deemed to be elected is equal to the number of vacancies to be filled, no further transfer shall be made.

    (b) When at the end of any count the number of continuing candidates is equal to the number of vacancies remaining unfilled, the continuing candidates shall thereupon be deemed to be elected.

    (c) When the last vacancies can be filled under this Rule, no further transfer shall be made.

13. At the end of every count the Academic Council election returning officer shall record on a result sheet in the prescribed form the total of the values credited to each candidate at the end of that count and also the value of the non-transferable papers not effective on that count and the loss of value on that count owing to disregard of fractions.

14. While the votes are being counted the ballot papers shall so far as it is practicable be kept face upwards and all proper precautions shall be taken by the Academic Council election returning officer for preventing the numbers on the backs of the ballot papers being seen.

15. (a) Any candidate or his/her agent may, at the conclusion of any count, request the Academic Council election returning officer to re-examine and recount all or any of the ballot papers dealt with during that count, and the Academic Council election returning officer shall forthwith re-examine and recount accordingly the ballot papers indicated.

    (b) The Academic Council election returning officer may at his/her discretion recount ballot papers either once or more often in any case in which he/she is not satisfied as to the accuracy of any count.

(c) Nothing in this Rule shall make it obligatory on the Academic Council election returning officer to recount the same parcel of ballot papers more than once.

16. In these Rules:

(a) The expression 'continuing candidate' means any candidate not deemed to be elected and not excluded;

(b) The expression 'first preference' means the figure '1' standing alone, the expression 'second preference' means the figure '2' standing alone in succession to the figure '1', and the expression 'third preference' means the figure '3' standing alone in succession to the figures '1' and '2' set opposite the name of any candidate, and so on;

(c) The expression 'next available preference' means a second or subsequent preference recorded in unique consecutive numerical order for a continuing candidate, the preference next in order on the ballot paper for candidates already deemed to be elected or excluded being ignored;

(d) The expression 'transferable paper' means a ballot paper on which, following a first preference, a second or subsequent preference is recorded in numerical order for a continuing candidate;

(e) The expression 'non-transferable paper' means a ballot paper

i. on which no second or subsequent preference is recorded for a continuing candidate; or

ii. on which the names of two or more candidates (whether continuing or not) are marked with the same number, and are next in order of preference; or

iii. on which the name of the candidate next in order of preference (whether continuing or not) is marked by a number not following consecutively after some other number on the voting paper or by two or more numbers; or

iv. which is void for uncertainty;

(f) the expression 'original vote' in regard to any candidate means a vote derived from a ballot paper on which a first preference is recorded for that candidate;

(g) The expression 'transferred vote' in regard to any candidate means a vote derived from a ballot paper on which a second or subsequent preference is recorded for that candidate;

(h) The expression 'surplus' means the number by which the total value of the votes, original and transferred, credited to any candidate exceeds the quota;

(i) The expression 'count' means (as the context may require) either

   i. All the operations involved in the counting of the first preferences recorded for candidates; or

   ii. All the operations involved in the transfer of the surplus of an elected candidate; or

   iii. All the operations involved in the transfer of the votes of an excluded candidate; or

   iv. The transfer in pursuance of these Rules of the papers of a candidate deemed not to be a continuing candidate;

(j) The expression 'deemed to be elected' means deemed to be elected for the purpose of counting, but without prejudice to the declaration of the result of the election;

(k) The expression 'determine by lot' means determine in accordance with the following directions, that is to say:

   the names of the candidates concerned, having been written on similar slips of paper, and the slips having been folded so as to prevent identification and mixed and drawn at random, the candidates concerned shall as amongst themselves be arranged on the order of preferences in the order in which the slips containing their names are drawn, beginning with the candidate whose name is on the slip drawn first.

Appendix B

Z Specification of an STV electoral system, specifically Waterford Institute of Technology's Academic Council election.

B.1 Introduction

This documents contains a specification for the process of counting the votes polled in a STV electoral system. The particular system used is based largely on the rules of election for Seanad Eireann, the upper house of parliament in Ireland. These rules have been modified for use in the election of academics for the Waterford Institute of Technology's Academic Council Election. The main modification needed is to take into account the need for gender and school balance in the elected members cohort. The full set of rules are available in the document 'Academic Council Rules'.

This document breaks up the specification into a number of parts

- Global declarations
- Operations needed to pre-process voting papers - this gets rid of spoiled votes, strips off any duplicates at the 'end' of the vote and produces a neater 'Ballot' which is processed.

- Operations needed to count the votes, transfer votes from elected members, eliminated, choosing next count's activities, etc. .
- System state and overall operation of count.

B.2 Global Declarations

The first basic type is that of CAND which contains the information for each candidate. This will be used as part of a more used Candidate type.

$[CAND]$

There must be a minimum 40% from each gender in the elected cohort. There must also be an (aspirational) minimum from each school. There must be as part of candidates information, values indicating to which gender and school candidate belongs. The following free types are declared:

$GENDER ::= male \mid female$

$SCHOOL ::= science \mid engineering \mid business \mid humanities$

The schema Candidate will be used throughout the specification.

Candidate

$cand : CAND$
$gender : GENDER$
$school : SCHOOL$

The process of changing what is known as 'voting papers' (input as a sequence of papers, each of which is a function from Candidate to that Candidates preference) to what are known as Ballots is described in the next section. The Ballot schema includes the preference sequence and the Ballot's value or weight.

Ballot

$preference : \mathrm{iseq}\, Candidate$
$value : \mathbb{Z}$

$-1000 \leq value \leq 1000$

As it is necessary to record duplicate ballots, bags of ballots are used throughout the specification. For some operations, the finiteness of these bags are necessary, so the following is used throughout to indicate a bag of ballots:

$$finBagBallot == \{B : \text{bag}\ Ballot \mid \text{dom}\ B \in \mathbb{F}\ B\}$$

A number of shorthand types are defined that will be used throughout the specification. The main function throughout the specification is declared as VoteMass. This is a function which links a Candidate with Bags of Ballots for each count. It resembles the physical model, where a pigeonhole structure is used and a Candidate's ballots from different counts are separated with labelled sheets. FunctBag describes each candidate's pile of ballots. The model to hold nontransferable votes, NonTransfers, is simply a pile of ballots, separated by count.

$$VoteMass == Candidate \twoheadrightarrow (\mathbb{N} \twoheadrightarrow finBagBallot)$$
$$FunctBag == \mathbb{N} \twoheadrightarrow finBagBallot$$
$$NonTransfers == \mathbb{N} \twoheadrightarrow finBagBallots$$
$$Paper == Candidate \twoheadrightarrow \mathbb{N}$$

B.3 Pre-Processing of Voting Papers

It is assumed that voting papers which are the main input to the count process can be described as a function from Candidate to a natural number. The input of voting papers will be in a sequence (this is a straighforward way of modelling the physical input - if this seems that this could lead to a breach of the secretness of the poll, then this can be further examined)

These papers are pre-processed, which involves

1. getting rid of spoiled votes
2. stripping away any preferences that are non-contigous or duplicated, leaving the leading preferences intact.

For valid papers, the preference part is changed to a sequence (this is possible, because now preferences are contigious and non-duplicated). The Ballot type emcompasses this. A value (sometimes known as weight) is also associated with a Ballot.

B.3.1 Z Specification of Pre–processing

Note that we are expecting non-zero preferences. Valid preferences on a voting paper start at 1 and are unique, increasing and contiguous. We specify a function which returns the first non-unique, non-contiguous or non-existent preference. This number minus 1 is the number of valid preferences on the voting paper. All preferences between 1 and this number are valid.

$$find_first_hole_or_dup : Paper \rightarrow \mathbb{N}$$

$$find_first_hole_or_dup(paper) = \\ \quad min\{n : \mathbb{N} \mid n : 1 \mathinner{\ldotp\ldotp} no_cands + 1 \ \wedge \ \#paper^{\sim}(\!|\ \{n\}\ |\!) \neq 1\}$$

The next function takes the voting paper and returns a (valid) Ballot with invalid preferences stripped. This means that, for instance, a spoiled vote will have no valid preferences.

$$make_ballot : Paper \rightarrow Ballot$$

$$make_ballot(paper) = \\ \quad \lhd pref \leadsto 1 \mathinner{\ldotp\ldotp} find_first_hole_or_dup(paper) - 1 \lhd paper^{\sim}, \\ \quad\quad value \leadsto 1000 \rhd$$

The following function takes a sequence of voting papers and returns a sequence of Ballots.

$$makeseqBallots : \mathrm{seq}\, Paper \rightarrow \mathrm{seq}\, Ballot$$

$$makeseqBallots(seqpapers) = map\ make_ballot\ seqpapers$$

This function throws away empty ballots. These are invalid papers(or spoiled votes) that were stripped down to empty ballots.

$$throwawayempties : \mathrm{seq}\, Ballot \rightarrow \mathrm{seq}\, Ballot$$

$$throwawayempties(fullseq) = \\ \quad fullseq \upharpoonright \{b : Ballot \mid b \in \mathrm{ran}\, fullseq \wedge \#(b.preference) > 0 \bullet b\}$$

The following function takes in the sequence of voting papers and produces a bag of preprocessed ballots. As a sequence is finite, then items returns a finite bag of Ballots. We call this type *finBagBallot*.

$$pre_process : \mathrm{seq}\, Paper \rightarrow finBagBallot$$

$$pre_process(seqpapers) = \\ \quad items(throwawayempties(makeseqBallots(seqpapers)))$$

Note that the following definition of map is assumed:

$$map : (X \rightarrow Y \;\times\; seq\ X) \rightarrow seq\ Y$$

$$map\ f\ s = \{n : 1 \mathrel{..} \#s \bullet n \mapsto f(s(n))\}$$

Back to old stuff

This can be called as Ballots = pre_process(Allpapers?) where allpapers is an input which is a sequence of the unprocessed papers.

It may be necessary to record the spoiled votes in certain circumstances.

$$spoiled_votes : \mathrm{seq}\, Paper \rightarrow seqPaper$$

$$\begin{array}{l} \forall\, allpapers : \mathrm{seq}\, Paper \bullet \\ spoiled_votes(allpapers) = \\ \quad allpapers \upharpoonright \\ \quad \{paper : Paper \mid find_first_hole_or_dup(paper) = 1\ \} \end{array}$$

B.4 Counting of Ballots

This section deals with the operations to transfer votes, choose next candidate to deal with etc. These operations will be used in the final section in the schema operation count.

There are some standard bag functions that will be needed to deal with bags of ballots.

bagvalue will used to calculate the total value of a bag of ballots.

$$bagvalue : finBagBallot \rightarrow \mathbb{N}$$

$$\begin{array}{l} bagvalue[\![\,]\!] = 0 \\ \forall\, b : Ballot;\ n : \mathbb{N}_1 \bullet \\ \quad bagvalue\{b \mapsto n\} = b.value * n \\ \forall\, B1, B2 : finBagBallot \bullet \\ \quad bagvalue(B1 \uplus B2) = bagvalueB1 + bagvalueB2 \end{array}$$

bagrange is a bag union generic function that takes a function from some generic type to a bag of ballots and returns a bagunion of the the range of the function.

$$
\begin{array}{l}
[X] \\
\hline
bagrange : (X \mathrel{+\!\!\!\!\twoheadrightarrow} finBagBallot) \mathrel{+\!\!\!\rightarrow} finBagBallot \\
\hline
bagrange \ = [\![\,]\!] \\
\forall x : X;\ B : finBagBallot \bullet \\
\qquad bagrange\{x \mapsto B\} = B \\
\forall f, g : X \mathrel{+\!\!\!\!\twoheadrightarrow} finBagBallot \mid \mathsf{disjoint}\ \langle \operatorname{dom} f, \operatorname{dom} g\rangle \bullet \\
\qquad bagrange(f \cup g) = (bagrange\ f) \uplus (bagrange\ \ g)
\end{array}
$$

The next stage is building up the operations for counting the ballots. To count the total number of valid votes, totalcount is used.

$$
\begin{array}{l}
totalCount : finBagBallots \rightarrow \mathbb{N} \\
\hline
\forall Ballots : finBagBallots \bullet \\
\qquad totalCount(Ballots) = bagvalue\ \ Ballots
\end{array}
$$

When counting votes, we need to total a candidates Ballots:

$$
\begin{array}{l}
totalvaloffunct : (FunctBag \times \mathbb{N}) \rightarrow \mathbb{N} \\
\hline
\forall countno : \mathbb{N};\ candfunct : FunctBag \bullet \\
\qquad totalvaloffunct(candfunct, countno) = \\
\qquad\qquad bagvalue(bagrange(1 \mathinner{..} countno) \lhd candfunct)
\end{array}
$$

A function is required to calculate the total weight of candidates at a given count. This value, when added to non-transferables and loss of weight is constant for each count because votes and values of votes are travelling around within the 'count mass' and do not leak.

$$
\begin{array}{l}
totalweightofCandidates : (\mathbb{N} \times VoteMass \times \mathbb{P}\, Candidate) \rightarrow \mathbb{Z} \\
\hline
\forall count : \mathbb{N};\ vm : VoteMass;\ ;\ c : Candidate \bullet \\
\qquad totalweightofCandidates(count, vm,) = 0 \\
\forall count : \mathbb{N};\ vm : VoteMass;\ c : Candidate \bullet \\
\qquad totalweightofCandidates(count, vm, \{c\}) = \\
\qquad totalvaloffunct(vm\ \ c, count) \\
\forall count : \mathbb{N};\ vm : VoteMass;\ S1, S2 : \mathbb{P}\, Candidate \bullet \\
\qquad totalweightofCandidates(count, vm, S1 \cup S2) = \\
\qquad\qquad totalweightofCandidates(count, vm, S1) + \\
\qquad\qquad totalweightofCandidates(count, vm, S2)
\end{array}
$$

weightatcount totals the weight of transferred ballots and non-transferables at this count, i.e. the traffic at a particular count. Note that this does not

include 'loss of value' (value lost due to remainders). This will be calculated in an invariant by stating that weight at each count + loss of value = 0.

$$
\begin{array}{l}
\mathit{weightatcount} : (\mathbb{N} \times \mathit{VoteMass} \times \mathit{FunctBag}) \rightarrow \mathbb{Z} \\
\hline
\forall\, \mathit{count} : \mathbb{N};\ \mathit{vm} : \mathit{VoteMass};\ \mathit{nontrans} : \mathit{FunctBag} \bullet \\
\quad \mathit{weightatcount}(\mathit{count}, \mathit{vm}, \mathit{nontrans}) = \\
\quad \mathit{bagvalue}\ \mathit{bagrange}\{c : \mathit{Candidate} \mid c \in \mathrm{dom}\, \mathit{vm} \land \\
\qquad \mathit{count} \in \mathit{dom}(\mathit{vm}\ c) \bullet \mathit{vm}\ c\ \mathit{count}\} \\
\quad + \mathit{bagvalue}(\mathit{nontrans}\ \mathit{count})
\end{array}
$$

findquota calculates the quota which is the minimum number of votes a candidate must have (normally) to be elected. (A candidate may in certain circumstances be elected without reaching the quota).

$$
\begin{array}{l}
\mathit{findquota} : \mathit{finBagBallot} \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, \mathit{Ballots} : \mathit{finBagBallot};\ \mathit{no_seats} : \mathbb{N} \bullet \\
\quad \mathit{findquota}(\mathit{Ballots}, \mathit{no_Seats}) = \\
\qquad (\mathit{totalcount}(\mathit{Ballots})\ \mathrm{div}\ (\mathit{no_seats} + 1)) + 1
\end{array}
$$

In the W.I.T. Academic Council election, there are gender and school constraints (see Rules of Election). Briefly, this means that there must be at least 40% of each gender (this currently translates to 5 out of 12) and a minimum (currently 2 out of 12) from each of the four schools. The gender minimums are statutory and it is assumed that on embarking on a count that the condition holds that there are at least the minimum number of candidates from each gender running for election. (In practice, it is the duty of the returning officer to ensure that this is the case and the election process cannot take place until the matter is resolved).

Whereas the gender minimums are statutory, the school balance are more aspirational. Under the condition that there are not sufficient candidates to satisfy the school minimum as laid down in minSchool?, the mimimum becomes the number of candidates available. The following functions calculate the minimum number from each school to be used for checking the school balance later.

$$
\begin{array}{l}
\mathit{findsciencemin} : \mathbb{P}\, \mathit{Candidate} \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, \mathit{AllCandidates} : \mathbb{P}\, \mathit{Candidate};\ \mathit{minschool} : \mathbb{N} \bullet \\
\mathit{findsciencemin}(\mathit{AllCandidates}, \mathit{minschool}) = \\
\mathit{min}(\#\{c : \mathit{Candidate} \mid c \in \mathit{AllCandidates} \land c.\mathit{school} = \mathit{science}\}, \\
\qquad \mathit{minschool})
\end{array}
$$

$$
\begin{array}{|l}
findbusinessmin : \mathbb{P}\, Candidate \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall AllCandidates : \mathbb{P}\, Candidate;\ minschool : \mathbb{N} \bullet \\
findbusinessmin(AllCandidates, minschool) = \\
min(\#\{c : Candidate \mid c \in AllCandidates \wedge c.school = business\} \\
\qquad , minschool)
\end{array}
$$

$$
\begin{array}{|l}
findengineeringmin : \mathbb{P}\, Candidate \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall AllCandidates : \mathbb{P}\, Candidate;\ minschool : \mathbb{N} \bullet \\
findengineeringmin(AllCandidates, minschool) = \\
min(\#\{c : Candidate \mid c \in AllCandidates \wedge c.school = engineering\} \\
\qquad , minschool)
\end{array}
$$

$$
\begin{array}{|l}
findhumanitiesmin : \mathbb{P}\, Candidate \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall AllCandidates : \mathbb{P}\, Candidate;\ minschool : \mathbb{N} \bullet \\
findhumanitiesmin(AllCandidates, minschool) = \\
min(\#\{c : Candidate \mid c \in AllCandidates \wedge c.school = humanities\}, \\
\qquad minschool)
\end{array}
$$

We now define the function to check the gender balance.This will be called e.g.

$$
(continuing, elected) \in GenandSchoolBalanced(mingen, minschool, \\ no_seats, AllCandidates)
$$

The gender and school constraints are ensured as follows: Before a candidate is elected or eliminated the following check is made: Is there a subset of continuing candidates, when added to the already elected candidates will make up a cohort of size no_seats so that the cohort obeys the gender and school constraints. When eliminating a candidate, the candidate could be needed, e.g. the Candidate could be one of only two candidates from a particular school (where 2 is the minimum form each school). Election of a particular candidate could make it impossible to achieve this balance (e.g. if the minimum from each gender is 5 out of 12 candidates, if the system elects the eight candidate from either gender this will destroy any possiblilty of being able to achieve the balance.)

$$
\begin{array}{l}
GenandSchoolBalanced : (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{P}\, Candidate) \rightarrow\!\!\!\!\!\!+\ \ \mathbb{P}(\mathbb{P}\, Candidate \times \mathbb{P}\, Candidate) \\
\hline
\forall mingen, minschool, no_seats : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidate \bullet \\
GenandSchoolBalanced(mingen, minschool, no_seats, Candidates) = \\
\{continuing, elected : \mathbb{P}\, Candidate \mid \\
continuing \subseteq AllCandidates \land \\
elected \subseteq AllCandidates \land \mathsf{disjoint}\ \langle elected, continuing\rangle \land \\
\#elected + \#continuing = no_seats \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.gender = male \bullet c\} \geq \\
\quad mingen \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.gender = female \bullet c\} \geq \\
\quad mingen \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.school = science \bullet c\} \geq \\
\quad findsciencemin(AllCandidates, minschool) \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.school = business \bullet c\} \geq \\
\quad findbusinessmin(AllCandidates, minschool) \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.school = engineering \bullet c\} \geq \\
\quad findengineeringmin(AllCandidates, minschool) \land \\
\#\{c : Candidate \mid c \in AllCandidates \land c.school = humanities \bullet c\} \geq \\
\quad findhumanitiesmin(AllCandidates, minschool) \bullet \\
\qquad (continuing, elected)\}
\end{array}
$$

To order continuing candidates, we follow the rules as specified in the rules for ordering(see Election Rules). If two candidates are tied when all tests are carried out, i.e. they have equal number of first preferences, second preferences etc., then the order is imposed randomly or, as described, by 'drawing by lot'. This is specified by defining a sequence whose range is the set of elements who need to be ordered.

$$
\begin{array}{l}
draw_by_lot : \mathbb{P}\,\mathbb{N} \rightarrow\!\!\!\!\!\!+\ \ \mathrm{seq}\,\mathbb{N} \\
\hline
\forall setofcands : \mathbb{P}\,\mathbb{N} \mid \\
\quad \exists sx : \mathrm{seq}\,\mathbb{N} \mid \mathrm{ran}\, sx = setofcands \land \#setofcands = \#sx \bullet \\
draw_by_lot(setofcands) = sx
\end{array}
$$

A function is defined to take in a votemass, the current count and the set of candidates to be ordered and returns the sequence of candidates in

order. The set of candidates are assumed to be continuing as a different ordering is needed on elected candidates.

$$Inorder : (VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate) \rightarrow \operatorname{seq} Candidate$$

$$\begin{array}{l}
\forall\, vm : VoteMass;\ count : \mathbb{N};\ setcands : \mathbb{P}\, Candidate \mid \\
\quad \exists\, orderedseq : \operatorname{seq} Candidate \mid \operatorname{ran} orderedseq = setcands \bullet \\
\quad\quad \forall\, i, j : \operatorname{dom} orderedseq \mid i < j \bullet \\
\quad\quad\quad ((totalvaloffunct(vm\ orderedseq\ i, count) > \\
\quad\quad\quad totalvaloffunct(vm\ orderedseq\ j, count)) \vee \\
\quad\quad\quad (totalvaloffunct(vm\ orderedseq\ i, count) = \\
\quad\quad\quad totalvaloffunct(vm\ orderedseq\ j, count) \wedge \\
\quad\quad\quad ((\exists\, c : \mathbb{N} \mid 1 \mathinner{..} count \bullet \\
\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ i, c) > \\
\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ j, c) \wedge \\
\quad\quad\quad\quad \forall\, ic : c + 1 \mathinner{..} count \bullet \\
\quad\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ i, ic) = \\
\quad\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ j, ic)) \vee \\
\quad\quad\quad (\forall\, c : 1 \mathinner{..} count \bullet \\
\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ i, c) = \\
\quad\quad\quad\quad totalvaloffunct(vm\ orderedseq\ j, c) \wedge \\
\quad\quad\quad\quad drawbylot(\{i, j\}) = \langle i, j \rangle))))) \bullet \\
\quad Inorder(vm, count, seteands) = orderedseq
\end{array}$$

A candidate is deemed to be elected at the earliest possible opportunity, i.e. at the first count which the candidates combined value of ballots is over or equal to the quota. At this point it can be decided whether the candidate should be elected or eliminated, depending on the gender and school constraints. It is necessary to know the number of the count when the candidate reached the quota for the first time.

$$firstcountover : (VoteMass \times Candidate \times finBagBallots \times \mathbb{N}) \nrightarrow \mathbb{N}$$

$$\begin{array}{l}
\forall\, vm : VoteMass;\ cand : Candidate;\ Ballots : FinBagBallots; \\
\quad\quad no_seats : \mathbb{N} \mid \\
\quad \exists\, n : \mathbb{N} \bullet \\
\quad totalvaloffunct(vm\ cand, n - 1) < findquota(Ballots, no_seats) \wedge \\
\quad totalvaloffunct(vm\ cand, n) \geq findquota(Ballots, no_seats) \bullet \\
firstcountover(vm, cand, Ballots, no_seats) = n
\end{array}$$

When deciding which candidate to process next, if there are a number of candidates who have reached the quota, *getnextoverquota* returns that candidate.

$$
\begin{array}{l}
getnextoverquota : \\
\qquad (VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate \times finBagBallots \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\qquad \rightarrowtail Candidate \\
\hline
\forall vm : VoteMass;\ count : \mathbb{N};\ Ballots : finBagBallots;\ no_seats : \mathbb{N} \mid \\
\exists overeqquota : \mathbb{P}\, Candidate;\ seqovereqquota : seqCandidate \mid \\
\quad overquota = \{c : Candidate \mid \\
\qquad c \notin dealtwith(vm, count - 1, AllCandidates, Ballots, \\
\qquad\qquad no_seats, mingen, minschool)) \land \\
\qquad totalvaloffunct(vm\ c, count) \geq \\
\qquad findquota(Ballots, no_seats)\} \land \\
\quad \mathrm{ran}\ seqovereqquota = overeqquota \land \\
\quad \forall i, j : \mathrm{dom}\ seqovereqquota \mid i < j \bullet \\
\qquad (firstcountover(vm, seqovereqquota\ i, Ballots, no_seats) < \\
\qquad firstcountover(vm, seqovereqquota\ j, Ballots, no_seats)) \lor \\
\qquad (firstcountover(vm, seqovereqquota\ i, Ballots, no_seats) = \\
\qquad firstcountover(vm, seqovereqquota\ j, Ballots, no_seats)) \land \\
\qquad \exists prevcontseq : seqCandidate \mid \\
\qquad\quad prevcontseq = Inorder(vm, \\
\qquad\qquad\qquad firstcountover(vm, seqovereqquota\ i, \\
\qquad\qquad\qquad\qquad Ballots, no_seats) - 1, \\
\qquad\qquad\qquad overeqquota) \bullet \\
\qquad\quad prevcontseq^{\sim}(seqovereqquota\ i) < \\
\qquad\quad prevcontseq^{\sim}(seqovereqquota\ j)) \bullet \\
getnextoverquota(vm, count, Ballots, AllCandidates, \\
\qquad\qquad no_seats, mingen, minschool) = \\
\qquad\quad head\ \ seqovereqquota
\end{array}
$$

If there are no candidates over quota to be dealt with, the next option is to find the next suitable candidate for elimination. The lowest candidate in the order of continuing candidates is the first to be checked (gender and school balance wise), next lowest and so on until the first lowest candidate is found to obey the school and gender balance checks .

$$
\begin{array}{l}
getnexttoexclude : \mathbb{P}\, Candidate \times VoteMass \times \\
\qquad \mathbb{N} \times \mathbb{P}\, Candidates \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\qquad\qquad \rightarrowtail Candidate \\
\hline
\forall\, continuing, AllCandidates : \mathbb{P}\, Candidateslvm : VoteMass; \\
count, mingen, minschool, no_seats : \mathbb{N} \mid \\
\exists\, contseq : \mathrm{seq}\, Candidate;\ pos : 1 \mathinner{..} \#contseq \mid \\
\quad \mathrm{ran}\, contseq = continuing \land \\
\quad contseq = Inorder(vm, count, contseq) \land \\
\\
\quad (continuing \setminus \{contseq\ pos\}, \\
\quad \mathrm{ran}\, electedseq(vm, count, AllCandidates, mingen, minschool, no_seats)) \in \\
\quad GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \land \\
\quad \neg\ \exists\, opos : pos + 1 \mathinner{..} \#contseq \bullet \\
\quad (continuing \setminus \{contseq\ opos\}, \\
\quad \mathrm{ran}\, electedseq(vm, count, AllCandidates, mingen, minschool, no_seats)) \in \\
\quad GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \bullet \\
getnexttoexclude(continuing, vm, count, AllCandidates, \\
\qquad mingen, minschool, no_seats) = \\
\qquad conseq\ pos
\end{array}
$$

A list of candidates who have been processed (on at each count) is needed at certain stages in the specification. *dealtwith* produces a set of such candidates. It calls *findnextcand* which itself calls *dealtwith* but on an earlier count.

$$
\begin{array}{l}
dealtwith : (VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate \times finBagBallot \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\qquad \rightarrowtail \mathbb{P}\, Candidate \\
\hline
\forall\, vm : VoteMass;\ count : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidate; \\
Ballots : finBagBallots;\ no_seats, mingen, minschool : \mathbb{N} \mid \\
\quad \exists\, dealtw : \mathbb{P}\, Candidate \mid \\
\qquad \forall\, i : 1 \mathinner{..} count \bullet \\
\qquad findnextcand(vm, i, \mathrm{ran}\, contseq(vm, count, AllCandidates), \\
\qquad\quad AllCandidates, Ballots, no_seats, mingen, minschool) \in \\
\qquad\quad dealtw \land \\
\qquad \#dealtw = count \bullet \\
dealtwith(vm, count, AllCandidates, Ballots, no_seats, \\
\qquad mingen, minschool)) = \\
\qquad dealtw
\end{array}
$$

findnextcand decides which is the next appropriate action, elect or eliminate a candidate and calls the appropriate operations.

$$
\begin{array}{l}
findnextcand : (VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate \times finBagBallots \times \\
\qquad\qquad \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\qquad\quad \nrightarrow Candidate \\
\hline
\forall vm : VoteMass;\ count : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidate; \\
\qquad\quad Ballots : finBagBallots; \\
\qquad no_seats, mingen, minschool : \mathbb{N} \bullet \\
findnextcand(vm, count, AllCandidates, Ballots, no_seats, \\
\qquad\quad mingen, minschool) = \\
if \\
\{c : Candidate \mid totalvaloffunct(vm\ \ c, count) \geq \\
\qquad\quad findquota(no_seats, Ballots)\} \\
\qquad\quad \backslash dealtwith(vm, count - 1, etc) \neq \\
\qquad getnextoverquota(vm, count, AllCandidates, Ballots, \\
\qquad\quad no_seats, mingen, minschool) \\
else \\
\qquad getnexttoexcude(vm, count, AllCandidates, Ballots, \\
\qquad\quad no_seats, mingen, minschool)
\end{array}
$$

To find a candidate's surplus, the quota and candidates present vote is needed.

$$
\begin{array}{l}
findsurplus : VoteMass \times Candidate \times finBagBallots \times \mathbb{N} \nrightarrow \mathbb{N} \\
\hline
\forall vm : VoteMass;\ cand : Candidate;\ Ballots : finBagBallots; \\
\qquad no_seats : \mathbb{N} \bullet \\
findsurplus(vm, cand, Ballots, no_seats) = \\
let\ totalvote = totalvaloffunct(vm\ \ max\ \ \mathrm{dom}\, vm\ \ cand); \\
\qquad quota = findquota(Ballots, no_seats) \\
\qquad in\ totalvote - quota \\
end
\end{array}
$$

A function is needed to return the next preference candidate on a Ballot. This candidate must be a continuing candidate. There may be no such candidate (in this case the Ballot is deemed to be non-transferable). Note that the preference part of the Ballot is injective.

$$\begin{array}{l} nextpref : (Ballot \times Candidate \times \mathbb{P}\,Candidate) \pfun Candidate \\ \hline \forall\, b : Ballot;\ currcand : Candidate;\ continuing : \mathbb{P}\,Candidate \mid \\ \quad \#(((b.preference^{\sim}(currcand) \mathbin{..} \#b.preference) \upharpoonleft b.preference) \upharpoonright \\ \qquad continuing) > 0 \bullet \\ \quad nextpref(b, currcand, continuing) = \\ \qquad head(((b.preference^{\sim}(currcand) \mathbin{..} \#b.preference) \\ \qquad\quad \upharpoonleft b.preference) \upharpoonright continuing) \end{array}$$

When a candidate's ballots are being transferred, it is always the last set of ballots(only), modelled as a bag of ballots, that was assigned to the candidate that is transferred. To prepare for this, the following function takes the bag of ballots last assigned to the current candidate and returns a function from Candidate to a bag of ballots where this function defines where each ballot should transfer to, if that place exists. The value of each Ballot is not changed here.

$$\begin{array}{l} preparetransbag : finBagBallots \times Candidate \times \mathbb{P}\,Candidate \\ \quad \pfun (Candidate \pfun finBagBallots) \\ \hline \forall\, lastbag : finBagBallots;\ currcand : Candidate; \\ \quad continuing : \mathbb{P}\,Candidate \mid \\ \exists\, candwithbags : Candidate \pfun finBagBallots \bullet \\ \quad \forall\, b : Ballot;\ n : \mathbb{N} \mid \\ \quad b \mapsto n \in lastbag \land \\ \quad (b, currcand, continuing) \in \mathrm{dom}\, nextpref \bullet \\ \quad b \mapsto n \in candwithbagscurrcandnextpref(b, currcand, continuing) \bullet \\ \quad preparetransbag(lastbag, currcand, continuing) = candwithbags \end{array}$$

For transferring purposes, it is handy to have a function that returns the bag of ballots which is non-transferable, given the current set of continuing candidates.

$$
\begin{array}{|l}
nontransferables : finBagBallots \times Candidate \times \mathbb{P}\, Candidate \rightarrow \\
\quad finBagBallots \\
\hline
\forall\, lastbag : finBagBallots;\ currcand : Candidate; \\
\quad continuing : \mathbb{P}\, Candidate \mid \\
\exists\, nontransfers : finBagBallots \mid \\
\quad \forall\, b : Ballot;\ n : \mathbb{N} \mid \\
\qquad b \mapsto n \in lastbag \land \\
\qquad (b, currcand, continuing) \notin \operatorname{dom} nextpref \bullet \\
\quad b \mapsto n \in nontransfers \bullet \\
\quad nontransferables(lastbag, currcand, continuing) = nontransfers
\end{array}
$$

The following function returns the bag of ballots to be transferred. This will be used for calculation of transferweight.

$$
\begin{array}{|l}
transferables : finBagBallots \times Candidate \times \mathbb{P}\, Candidate \rightarrow \\
\quad finBagBallots \\
\hline
\forall\, lastbag : finBagBallots;\ currcand : Candidate; \\
\quad continuing : \mathbb{P}\, Candidate \mid \\
\exists\, transfers : finBagBallots \mid \\
\quad \forall\, b : Ballot;\ n : \mathbb{N} \mid \\
\qquad b \mapsto n \in lastbag \land \\
\qquad (b, currcand, continuing) \in \operatorname{dom} nextpref \bullet \\
\quad b \mapsto n \in transfers \bullet \\
\quad transferables(lastbag, currcand, continuing) = transfers
\end{array}
$$

Throughout the transfer process, it is often necessary to change the values of each ballot in a bag of ballots by a given factor. The following function effects this change.

$$
\begin{array}{|l}
changeweight : finBagBallot \times \mathbb{N} \nrightarrow finBagBallots \\
\hline
\forall\, bag : finBagBallots;\ factor : \mathbb{N} \mid \\
\exists\, changedbag : finBagBallots \mid \\
\quad \forall\, b : Ballot;\ n : \mathbb{N} \mid b \mapsto n \in bag \bullet \\
\qquad \exists\, b' : Ballot \bullet \\
\qquad b'.value = b.value * factor \land \\
\qquad b'.preference = b.preference \bullet \\
\quad b' \mapsto n \in changebag \bullet \\
\quad changeweight(bag, factor) = changedbag
\end{array}
$$

The following function transfers the ballots last received by the candidate. The weight of transferred votes depends primarily on whether non-transferables need to be dealt with. Non-transferables may need to be 'partially' transferred. The elected candidate should be left with a weight exactly equal to the quota. This means that the candidates receiving ballots will get ballots with a fraction of the weight of the original ballot. The transfer weight is defined below.

$$
\begin{array}{l}
\mathit{transferelected} : \\
\quad (\mathit{Candidate} \times \mathit{VoteMass} \times \mathit{NonTransfers} \times \mathbb{N} \times \\
\quad \mathit{finBagBallots} \times \mathbb{N} \times \mathbb{P}\, \mathit{Candidate}) \\
\qquad \rightarrow\!\!\!\!\!\!\mapsto (\mathit{VoteMass} \times \mathit{NonTransfers}) \\
\hline
\mathbf{let}\ \mathit{lastbag} == \mathit{vm}\ \mathit{currcand}\ \ \mathit{max}(\mathrm{dom}\, \mathit{vm}\ \mathit{currcand}); \\
\quad \mathit{transferbag} == \mathit{preparetransbag}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}); \\
\quad \mathit{weightedtransferbag} = \\
\qquad \mathit{changeweight}(\mathit{transferbag}, \mathit{surplus}\ \mathrm{div}\ \mathit{transferablevalue}); \\
\quad \mathit{surplus} == \mathit{findsurplus}(\mathit{vm}, \mathit{currcand}, \mathit{Ballots}, \mathit{no_seats}); \\
\quad \mathit{transferable} = \mathit{transferables}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}); \\
\quad \mathit{transferablevalue} == \mathit{bagvalue}(\mathit{transferables}); \\
\quad \mathit{nontransferables} == \mathit{nontransferables}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}); \\
\quad \mathit{nontransaway} = \\
\qquad \mathit{changeweight}(\mathit{nontransferables}, (-1\ *\ (\mathit{surplus} - \mathit{transferable}) \\
\qquad \mathrm{div}\, \mathit{bagvalue}(\mathit{nontransferables}))); \\
\quad \mathit{nontranstont} = \mathit{changeweight}(\mathit{nontransaway}, -1); \\
\quad \mathit{transnont} = \\
\qquad \mathit{changeweight}(\mathit{transferables}, ((\mathit{surplus} * -1)\ \mathrm{div}\ \mathit{transferablevalue})); \\
\quad \mathit{transisnt} = \mathit{changeweight}(\mathit{transferables}, -1) \\
\mathit{in} \\
\forall\, \mathit{currcand} : \mathit{Candidate};\ \mathit{vm} : \mathit{VoteMass};\ \mathit{nt} : \mathit{NonTransfers};\ \mathit{count} : \mathbb{N}; \\
\mathit{Ballots} : \mathit{finBagBallots};\ \mathit{no_seats} : \mathbb{N};\ \mathit{continuing} : \mathbb{P}\, \mathit{Candidate} \mid \\
\quad \exists\, \mathit{vm}' : \mathit{VoteMass};\ \mathit{nt}' : \mathit{NonTransfers} \bullet \\
\quad (\mathbf{let}\ \mathit{transferbag}' \\
\qquad \mathit{if}\ \ \mathit{surplus} < \mathit{transferable} \\
\qquad = \mathit{weightedtransferbag} \\
\qquad \mathit{else} \\
\qquad = \mathit{transferbag} \\
\quad \mathit{in} \\
\quad \forall\, c : \mathit{Candidate} \mid c \in \mathrm{dom}\, \mathit{transferbag} \\
\qquad \mathit{vm}'\ c = \mathit{vm}\ c \cup \{\mathit{count} \mapsto \mathit{transferbag}'\ c\} \\
\quad \mathit{end}) \land \\
\qquad \{c : \mathit{Candidate} \mid c \in \mathrm{dom}\, \mathit{transferbag}\} \cup \{\mathit{currcand}\} \mathbin{\lhd\!\!\!-} \mathit{vm}' = \\
\qquad \{c : \mathit{Candidate} \mid c \in \mathrm{dom}\, \mathit{transferbag}\} \cup \\
\{\mathit{currcand}\} \mathbin{\lhd\!\!\!-} \mathit{vm} \land \\
\quad ((\mathit{surplus} \geq \mathit{transferablevalue} \land \\
\quad \mathit{vm}'\ \mathit{currcand} = \mathit{vm}\ \mathit{currcand} \cup \{\mathit{count} \mapsto \mathit{transnont}\} \land \\
\quad \mathit{nt} = \mathit{nt}') \lor \\
\quad (\mathit{surplus} < \mathit{transferablevalue} \land \\
\quad \mathit{vm}'\ \mathit{currcand} = \mathit{vm}\ \mathit{currcand} \cup \{\mathit{count} \mapsto (\mathit{transisnt} \uplus \mathit{nontransaway})\} \\
\quad \mathit{nt}' = \mathit{nt} \cup \{\mathit{count} \mapsto \mathit{nontranstont}\})) \\
\qquad \mathit{transferelected}(\mathit{currcand}, \mathit{vm}, \mathit{nt}, \mathit{count}, \mathit{Ballots}, \mathit{no_seats}, \mathit{continuing}) = \\
\qquad\qquad (\mathit{vm}', \mathit{nt}') \\
\mathit{end}
\end{array}
$$

When a candidate is eliminated, the ballots are transferred to the next preference candidate, if they exist, or to non-transferables. The weights of the ballots remain the same as the eliminated candidates total will be 0 at the end of the operation.

$$
\begin{array}{l}
\mathit{transfereliminated} : (\mathit{Candidate} \times \mathit{VoteMass} \times \mathit{NonTransfers} \times \\
\qquad\qquad \mathbb{N} \times \mathbb{P}\, \mathit{Candidiate}) \\
\qquad \pfun (\mathit{VoteMass}, \mathit{NonTransfers}) \\
\hline
\mathbf{let}\ \mathit{lastbag} = \mathit{vm}\ \mathit{currcands}\ \mathit{max}(\mathrm{dom}\, \mathit{vm}\ \mathit{currcand}); \\
\qquad \mathit{transferbag} = \mathit{preparetransbag}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}); \\
\mathit{in} \\
\forall \mathit{currcand} : \mathit{Candidate};\ \mathit{vm} : \mathit{VoteMass};\ \mathit{nt} : \mathit{NonTransfers};\ \mathit{count} : \mathbb{N}; \\
\qquad \mathit{continuing} : \mathbb{P}\, \mathit{Candidate} \\
\exists \mathit{vm}' : \mathit{VoteMass}, \mathit{nt}' : \mathit{NonTransfers} \mid \\
\qquad (\forall c : \mathit{Candidate} \mid c \in \mathrm{dom}\, \mathit{transferbag} \bullet \\
\qquad\qquad \mathit{vm}'\ c = \mathit{vm}\ c \cup \{\mathit{count} \mapsto \mathit{transferbag}\ c\} \land \\
\qquad \forall c : \mathit{Candidate} \mid c \notin \mathrm{dom}\, \mathit{transferbag} \cup \{\mathit{currcand}\} \bullet \\
\qquad\qquad \mathit{vm}'\ c = \mathit{vm}\ c \land \\
\qquad \mathit{vm}'\ \mathit{currcand} = \mathit{vm}\ \mathit{currcand} \cup \\
\qquad\qquad\qquad \{\mathit{count} \mapsto \mathit{changeweight}(\mathit{vm}\ \mathit{currcand}(\mathit{count} - 1), -1)\} \land \\
\qquad ((\mathit{nontransferables}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}) \neq) \\
\qquad\qquad \land\ \mathit{nt}' = \mathit{nt} \cup \{\mathit{count} \mapsto \mathit{nontransferables}\}) \lor \\
\qquad (\mathit{nontransferables}(\mathit{lastbag}, \mathit{currcand}, \mathit{continuing}) =) \land \mathit{nt}' = \mathit{nt}))) \bullet \\
\qquad \mathit{transfereliminated}(\mathit{currcand}, \mathit{vm}, \mathit{nt}, \mathit{count}, \mathit{continuing}) = (\mathit{vm}', \mathit{nt}')
\end{array}
$$

The next function returns the sequence of elected candidates, at a particular count.

$$
\begin{array}{l}
electedseq : VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrowtail \\
\qquad \mathrm{seq}\, Candidate \\
\hline
\forall\, vm : VoteMass;\ count : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidate; \\
\quad mingen : \mathbb{N};\ minschool : \mathbb{N};\ no_seats : \mathbb{N} \mid \\
\exists\, dealtwithoverquota, electeddealtwith, overquotanotdw, \\
\quad tobeelected : \mathrm{seq}\, Candidate \mid \\
\quad dealtwithoverquota = \\
\qquad\qquad squash\{c : Candidate;\ cnt : \mathbb{N} \mid cnt : 1 \mathinner{\ldotp\ldotp} count - 1 \wedge \\
\qquad\qquad findnextcand(vm, cnt) = c \wedge \\
\qquad\qquad totalvaloffunct(vm\ \ c, cnt) \geq findquota(Ballots, no_seats) \bullet \\
\qquad\qquad cnt \mapsto c\} \\
\quad \mathrm{ran}\, electeddealtwith \subseteq \mathrm{ran}\, dealtwithoverquota \wedge \\
\quad electeddealtwith = dealtwithoverquota \upharpoonright ranelecteddealtwith \wedge \\
\quad (\forall\, i : 1 \mathinner{\ldotp\ldotp} \#electeddealtwith \bullet \\
\qquad (AllCandidates \setminus \\
\quad dealtwith(vm, i-1, AllCandidates, Ballots, no_seats, mingen, minschool), \\
\qquad \mathrm{ran}(1 \mathinner{\ldotp\ldotp} \lhd electeddealtwith)) \\
\quad \in GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \wedge \\
\quad \neg\ \exists\, othercans : \mathbb{P}\, Candidate \mid \\
\qquad othercands \subseteq (\mathrm{ran}\, dealtwithoverquota \setminus \mathrm{ran}\, electeddealtwith) \wedge \\
\qquad \#othercands > 0 \bullet \\
\qquad \exists\, oseq : \mathrm{seq}\, Candidate \mid \mathrm{ran}\, oseq = \mathrm{ran}\, electeddealtwith \cup othercands \bullet \\
\qquad \forall\, i : 1 \mathinner{\ldotp\ldotp} \#oseq \bullet \\
\qquad (AllCandidates \setminus \\
\quad dealtwith(vm, i-1, AllCandidates, Ballots, no_seats, mingen, minschool) \\
\quad ran(1 \mathinner{\ldotp\ldotp} i \lhd electeddealtwith)) \\
\quad \in GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \wedge \\
\quad \mathrm{ran}\, overquotanotdw \cap dealtwith(vm, count, AllCandidates, Ballots, \\
\qquad\qquad no_seats, mingen, minschool) = \wedge \\
\quad \forall\, c : ranoverquotanotdw \bullet \\
\qquad totalvaloffunct(vm\ \ c, count) \geq findquota(Ballots, no_seats) \wedge \\
\quad overquotanotdw = \\
\qquad Inorder(vm, count, AllCandidates \setminus \\
\qquad\qquad dealtwith(vm, count - 1, AllCandidates, Ballots, \\
\quad no_seats, mingen, minschool)) \wedge \\
\quad \mathrm{ran}\, tobeelected \subseteq overquotanotdw \wedge \\
\quad tobeelected = overquotanotdw \upharpoonright \mathrm{ran}\, tobeelected \wedge \\
\quad \forall\, i : 1 \mathinner{\ldotp\ldotp} \#tobeelected \bullet \\
\qquad (AllCandidates \setminus \\
\qquad dealtwith((vm, i, AllCandidates, Ballots, no_seats, mingen, minschool), \\
\quad \mathrm{ran}\, electeddealtwith \cup \mathrm{ran}(1 \mathinner{\ldotp\ldotp} i \lhd tobeelected)) \in \\
\qquad GenandSchoolBalanced(minschool, mingen, no_seats, AllCandidates) \wedge \\
\quad \neg\ \exists\, more : \mathbb{P}\, Candidate \mid more \subseteq \mathrm{ran}\, overquotanotdw \setminus \mathrm{ran}\, tobeelected \wedge \\
\qquad\qquad \#more > 0 \bullet \\
\quad \exists\, oseq : \mathrm{seq}\, Candidatd \mid \mathrm{ran}\, oseq = \mathrm{ran}\, tobeelected \cup more \bullet
\end{array}
$$

$$
\begin{array}{|l}
\forall i : 1 \mathrel{..} \# oseq \bullet \\
\quad (AllCandidates \setminus \\
(dealtwith(vm, i-1, AllCandidates, Ballots, no_seats, mingen, minschool) \\
\qquad \cup (\mathrm{ran}(1 \mathrel{..} \lhd oseq)), \\
\mathrm{ran}\ electeddealtwith \cup \mathrm{ran}(1 \mathrel{..} i) \lhd oseq) \\
\in GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \\
\quad \bullet \\
electedseq(vm, count, AllCandidates, Ballots, mingen, minschool, no_seats) = \\
\quad electeddealtwith \frown tobeelected
\end{array}
$$

A function is defined to order continuing candidates at a particular count:

$$
\begin{array}{|l}
contseq : VoteMass \times \mathbb{N} \times \mathbb{P}\, Candidate \pfun \mathrm{seq}\ Candidate \\
\hline
\forall vm : VoteMass, count : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidates \mid \\
\quad \exists continuing : \mathbb{P}\, Candidates;\ \ cseq : \mathrm{seq}\ Candidates \mid \\
\quad continuing = AllCandidates \setminus \\
\qquad \{c : Candidate;\ \ cnt : 1 \mathrel{..} count - 1 \mid \\
\qquad\quad c = findnextcand(vm, cnt, AllCandidates, Ballots, \\
\qquad\qquad no_seats, mingen, minschool) \bullet c\} \\
\qquad \cup \{c : Candidate \mid \\
\qquad\quad totalvaloffunct(vm\ \ c, count) \geq \\
\qquad\quad findquota(Ballots, no_seats)\} \land \\
\quad cseq = Inorder(vm, count, continuing) \bullet \\
\quad contseq(vm, count, Ballots, no_seats) = cseq
\end{array}
$$

The following function chooses the next candidate to deal with and either elects (and transfers) or eliminates (and transfers).

$$
\begin{array}{|l}
electoreliminate : (Candidate \times VoteMass \times NonTransfers \times \mathbb{N} \times \\
\quad \mathbb{P}\, Candidate \times finBagBallot \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow \\
\qquad (VoteMass \times NonTransfers) \\
\hline
\forall\, cand : Candidate;\ vm : VoteMass;\ nt : NonTransfers;\ count : \mathbb{N}; \\
\quad AllCandidates : \mathbb{P}\, Candidate;\ Ballots : finBagBallots; \\
\quad mingen, minschool, no_seats : \mathbb{N} \mid \\
\mathbf{let}\ continuing = \mathrm{ran}\ contseq(vm, count, AllCandidates); \\
\quad elected = \mathrm{ran}\ electedseq(vm, count, AllCandidates, Ballots, \\
\qquad mingen, minschool, no_seats) \\
in \\
(vm', nt') = \\
\quad if \\
\quad ((continuing \setminus \\
\{cand\}, elected \cup \{cand\}) \in \\
\quad GenandSchoolBalanced(mingen, minschool, no_seats, AllCandidates) \wedge \\
\quad totalvaloffunct(vm\ cand, count) \geq findquota(Ballots, no_seats)) \\
\quad then \\
\qquad transferelected(cand, vm, nt, count, Ballots, no_seats, continuing) \\
\quad else \\
\qquad transfereliminated(cand, vm, nt, count, Ballots, no_seats, continuing) \\
end
\end{array}
$$

The election count process will terminate if

1. The number of elected candidates is equal to the number of seats to be filled.
2. The number of elected candidates + the number of continuing candidated is equal to the number of number of seats to be filled.

The following function take as input the number of seats to be filled and returns each possible set of pairs of (elected, continuing) sets of candidates which will lead to termination of the process.

$$
\begin{array}{|l}
finished : \mathbb{N} \rightarrow \mathbb{P}(\mathbb{P}\, Candidate \times \mathbb{P}\, Candidate) \\
\hline
\forall\, no_seats : \mathbb{N} \bullet \\
\quad finished(no_seats) = \\
\qquad \{selected, scont : \mathbb{P}\, Candidate \mid \\
\qquad\quad \#selected = no_seats \bullet (selected, scont)\} \\
\qquad \cup \\
\qquad \{selected, scont : \mathbb{P}\, Candidate \mid \\
\qquad\quad \#scont + \#selected = no_seats \bullet (selected, scont)\}
\end{array}
$$

This function recursively calls itself until the count is finished, i.e. as defined above.

$$
\begin{array}{l}
election_count : (VoteMass \times NonTransfers \times \mathbb{N} \times \\
\quad finBagBallots \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{P}\, Candidate) \rightarrow\!\!\!\!\!+ \\
\quad (VoteMass \times NonTransfers) \\
\hline
\forall\, vm : VoteMass;\ nt : NonTransfers;\ i : \mathbb{N}, Ballots : finBagBallots; \\
\quad mingen, minschool, no_seats : \mathbb{N};\ AllCandidates : \mathbb{P}\, Candidate \bullet \\
\quad election_count(vm, nt, i, Ballots, mingen, minschool, no_seats, \\
\qquad AllCandidates) \\
\quad = \\
\quad \textbf{let}\ continuing == \operatorname{ran} contseq(vm, i, AllCandidates); \\
\qquad elected == \operatorname{ran} electedseq(vm, i, AllCandidates, Ballots, \\
\qquad\quad mingen, minschool, no_seats); \\
\qquad nextcand == findnextcand() \\
\quad in \\
\quad if(elected, continuing) \in finished(no_seats) \\
\quad else \\
\quad let(vm', nt') = \\
\quad electoreliminate \\
\qquad (findnextcand \\
\qquad\quad (vm, i, AllCandidates, Ballots, no_seats, mingen, minschool), \\
\qquad\quad vm, nt, i, AllCandidates, Ballots, mingen, minschool, no_seats) \\
\quad in \\
\qquad election_count(vm', nt', i + 1, Ballots, mingen, minschool, \\
\qquad\quad no_seats, AllCandidates)
\end{array}
$$

This will be called as

$$(votemass!, non_transferables!) = election_count(setupfirstcount(Ballots), 2)$$

setupfirstcount takes the pre-processed ballots and 'deals them out' according to their first preferences. No weight change is needed.

$$
\begin{array}{l}
setupfirstcount : finBagBallots \times \mathbb{P}\, Candidate \rightarrow VoteMass \\
\hline
\forall\, Ballots : finBagBallots;\ AllCandidates \mid \\
\exists\, vm' : VoteMass \mid \\
\quad \forall\, b : Ballot;\ n : \mathbb{N} \mid b \mapsto n \in Ballots \bullet \\
\qquad \exists_1\, c : Candidate \mid c \in AllCandidates \land head\ b.preference = c \bullet \\
\quad b \mapsto n \in vm'\ c\ 1 \bullet \\
\quad setupfirstcount(Ballots, AllCandidates) = vm'
\end{array}
$$

B.5 Count Operation

Finally, the operation for counting is defined.

Count

$VotingPapers? : \mathrm{seq}(Candidate \pfun \mathbb{N})$
$AllCandidates? : \mathbb{P}\ Candidate$
$mingen?, minschool, no_seats : \mathbb{N}$
$Ballots! : finBagBallots$
$votemass! : VoteMass$
$non_transferables! : NonTransfers$
$loss_of_value : \mathbb{N} \pfun \mathbb{N}$
$seqelected! : \mathrm{seq}\ Candidate$
$elected! : \mathbb{P}\ Candidate$
$spoiled_votes! : \mathrm{seq}(Candidate \pfun \mathbb{N})$

$Ballots! = pre_process(VotingPapers?)$
$(votemass!, non_transferables!) =$
$\quad election_count(setupfirstcount(Ballots!, AllCandidates!), , 2,$
$\quad Ballots!, mingen?, minschool?, no_seats)$
$spoiled_votes! = spoiledvotes(VotingPapers?)$
$seqelected! = electdseq(votemass!, max(\mathrm{dom}\{c : Candidate \mid$
$\qquad c \in \mathrm{dom}\ votemass! \bullet$
$\qquad votemass!\})$
$\quad , AllCandidates?, Ballots!, mingen?, minschool?, no_seats)$
$\forall i : 1 \,..\, max\ \mathrm{dom}\{c : Candidate \mid c \in \mathrm{dom}\ votemass! \bullet votemass!\ c\} \bullet$
$\quad weightatcount(i, votemass!, AllCandidates?)$
$\quad + loss_of_value\ i = 0$

Appendix C

Laws used in Refinement Calculus Example

The following laws are adapted from Morgan [23] Appendix C - Summary of Laws. These laws are used in the development of a program, when introducing algorithmic refinement using Morgan's Refinement Calculus in Chapter 2. They are re-numbered for clarity and they appear in alphabetical order

Law C1 *alternation*
If $pre \Rrightarrow GG$, then

$$w : [pre,\ post]$$
$$\sqsubseteq \textbf{if } (\ i \bullet\ G_i \rightarrow w : [G_i \land pre,\ post])\ \textbf{fi}$$

Law C2 *assignment*
If $pre \Rrightarrow post[w \backslash E]$, *then*

$$w, x : [pre, post] \sqsubseteq w := E$$

Law C3 *following assignment*
For any term E,

$$w, x : [pre, post]$$
$$\sqsubseteq w, x : [pre, post[x \backslash E]];$$
$$x := E$$

Law C4 *introduce local variable*
If x does not occur in w, pre or $post$ then

$$w : [pre, post] \ \sqsubseteq\ |[\ \textbf{var}\ x : T;\ \ \textbf{and}\ inv\ \bullet\ w,\ x : [pre, post]\ ]|$$

Law C5 *iteration*
Let *inv*, the *invariant*, be any formula; let V, the *variant*, be any integer-valued expression. Then, if G is the *Guard* and
$inv \wedge \neg G \Rightarrow post$ then

$$
\begin{array}{l}
w : [\ inv, post] \\
\sqsubseteq\ \mathbf{do}\ G \rightarrow \\
\quad w : [\ inv \wedge\ G,\ inv\ \wedge\ (0\ \leq\ V\ <\ V_0)\ ] \\
\mathbf{od}
\end{array}
$$

Law C6 *leading assignment*
For disjoint w and x,

$$w, x := E, F[w \backslash E] \quad = \quad w := E;\ \ x := F$$

Law C7 *sequential composition*
For any formula *mid*, where neither *mid* or *post* contain initial variables:

$$w : [\ pre,\ post\ ]\ \ \sqsubseteq\ \ w : [\ pre,\ mid\ ]\ ;\ \ w : [\ mid,\ post\ ]$$

Law C8 *strengthen postcondition*
If $post' \ \Rrightarrow\ post$, then

$$w : [pre\ ,\ post]\ \sqsubseteq\ w : [pre\ ,\ post']$$

The following law is adapted from Morgan & Vickers [22] and deals with a special kind of data refinement, i.e. when the abstraction invariant(R) is functional. The justification is found in 2.3.1
Law C9 *data refinement(functional)*
If $R\ \equiv\ a = f(c)$ (functional) (a is abstract, c is concrete) and when the postcondition does not refer to initial variables:
$a : [pre,\ post]\ \sqsubseteq_R\ c : [pre[a \backslash f(c)],\ post[a \backslash f(c)]\ ]$

Appendix D

B Specification of Academic Council count

D.1 Introduction

This appendix contains the specification of the counting system of the Academic Council voting system. The rules of this system are written in Appendix A. There are slight differences in this specification to the parts of the system as specified in the Case Studies of Chapter 5. The main difference is that the ballot in this case contains a weight whereas in the Case Study this is not the case. As the original value of the weight is always 1000 and during refinement it was spotted that it was unnecessary and wasteful to hold the ballots' weight at the start. However, this change has not been pulled through to this specification.

This specification is presented as a valid specification, written using B's AMN. We present the MACHINEs. The order in which they are presented follows a top-down approach. The driving operation is contained in the *Election* MACHINE which is presented first. The order of presentation of Machines is as follows:

- *Election*
- *Global Variables*
- *Bool_TYPE*
- *BallotBag*
- *PaperBag*

- *P_Prepare_Ballots*
- *Counts*
- *CountingFunctions*
- *GetNextCandidate*
- *OrderingFunctions*
- *FindBalanceMins*

D.2 B Specification

D.2.1 Overall Election

MACHINE *Election*
/*This machine contains the main driving operation of the system.
countvotes returns the results of the election.*/
SEES
GlobalVariables,
OrderingFunctions,
BallotBag,
Bool_TYPE,
Countingfunctions,
FindBalanceMins,
GetNextCandidate,
Counts,
P_Prepare_Ballots

VARIABLES *Votes*,
no_seats,
min_genpercent,
min_gen,
min_school,
min_dept,
Candidates,
Result

INVARIANT
$Votes \in BagPapers \wedge$
$no_seats \in \mathbb{N} \wedge$
$min_genpercent \in \mathbb{N} \wedge$
$min_gen \in \mathbb{N} \wedge$
$(no_seats > 0 \Rightarrow min_gen =$
$\quad min(\{nn \mid nn \in \mathbb{N} \wedge nn > no_seats * min_genpercent/100\}) \wedge$
$no_seats = 0 \Rightarrow min_gen = 0) \wedge$
$min_school \in \mathbb{N} \wedge$
$min_dept \in DEPARTMENT \rightarrow \mathbb{N} \wedge$
$Candidates \in SetCandidates \wedge$
$Result \in iseq(Candidate * ACT)$

INITIALISATION

$Votes := \varnothing \;||$
$no_seats := 0 \;||$
$min_genpercent := 0 \;||$
$min_gen := 0 \;||$
$min_school := 0 \;||$
$min_dept := DEPARTMENT \times \{0\} \;||$
$Candidates := \varnothing \;||$
$Result := <>$

OPERATIONS

$enterdata \;\widehat{=}$
BEGIN
$Votes :\in Paper \mathrel{+\!\!\!\!\rightarrow\!\!\!\!\rightarrow} \mathbb{N} \;||$
$no_seats :\in \mathbb{N} \;||$
$min_genpercent :\in \mathbb{N} \;||$
$min_school :\in \mathbb{N} \;||$
$min_dept :\in DEPARTMENT \rightarrow \mathbb{N}$
END ;

$Results \longleftarrow countvotes \;\widehat{=}$
/* This is the main driving operation of the system. The Results contain a sequence of pairs, each pair containing a Candidate and the action associated with that Candidate. The order of the sequence indicates the order that the actions took place.
All required information is contained here. */
BEGIN
ANY $Ballots$ **WHERE**
$Ballots \in BagBallots \wedge Ballots = P_prepare_ballots(Votes)$
THEN
ANY $firstvm, quota$ **WHERE**
$firstvm \in VoteMass \wedge quota \in \mathbb{N} \wedge$
$firstvm = Setup_First_Count(Ballots, Candidates) \wedge$
$quota = Find_Quota(Ballots, no_seats)$
THEN
ANY vm, nt, dw **WHERE**
$vm \in VoteMass \wedge nt \in NonTransfers \wedge dw \in DealtWith \wedge$
$vm \mapsto nt \mapsto dw =$
$Election_Count(firstvm, \varnothing, \varnothing, 2, Ballots, Candidates, min_gen,$
$min_school, min_dept, no_seats, quota)$

```
                THEN
                    Results := dw
                END
            END
        END
    END

END
```

D.2.2 Global Variables

MACHINE *GlobalVariables*

DEFINITIONS

$Candidate == NAME \times GENDER \times SCHOOL \times DEPARTMENT$;
$SetCandidates == \mathbb{P}(Candidate)$;
$VoteMass == Candidate \nrightarrow (\mathbb{N} \nrightarrow BagBallots)$;
$Paper == Candidate \nrightarrow \mathbb{N}$;
$DealtWith == iseq(Candidate \times ACT)$;
$NonTransfers == \mathbb{N} \nrightarrow BagBallots$;
$MinDept == DEPARTMENT \rightarrow \mathbb{N}$;
$BagPapers == Paper \nrightarrow \mathbb{N}_1$;
$Paper == Candidate \nrightarrow \mathbb{N}$;
$Ballot == seq(Candidate) \times \mathbb{N} \times SIGN$;
$BagBallots == Ballot \nrightarrow \mathbb{N}_1$

SETS

NAME; /*deferred until candidates officially entered */
$GENDER = \{male, female\}$;
$SCHOOL = \{engineering, business, science, humanities\}$;
$DEPARTMENT = \{adult_ed, other\}$;
/* As adult_ed is the only special case, I ignore the department of others, except to state that they are not in adult_ed. If an other department is similarly prescribed, then this department can be added in. If the condition for adult_ed is dropped, then simply change the mapping of adult_ed to zero (in deptmin) or change candidates to other. */
$ACT = \{elect, exclude\}$;
$SIGN = \{pos, neg\}$

CONSTANTS

school,
gender,
name,
department

PROPERTIES

$school \in Candidate \rightarrow SCHOOL \wedge$
$gender \in Candidate \rightarrow GENDER \wedge$
$name \in Candidate \rightarrow NAME \wedge$
$department \in Candidate \rightarrow DEPARTMENT \wedge$
$\forall(nn, gg, ss, dd).(nn \in NAME \wedge gg \in GENDER \wedge$
$ss \in SCHOOL \wedge dd \in DEPARTMENT \Rightarrow$
$school(nn, gg, ss, dd) = ss \wedge$
$gender(nn, gg, ss, dd) = gg \wedge$
$name(nn, gg, ss, dd) = nn \wedge$
$department(nn, gg, ss, dd) = dd$
$)$

END

MACHINE *Bool_TYPE*
/* This machine simply introduces the boolean type*/

SETS $BOOL = \{FALSE, TRUE\}$

END

D.2.3 Bags - Paper and Ballot

MACHINE *BallotBag*
/* These operations are specific to the structure , based on bags of ballots. We will use these during the count */

SEES
GlobalVariables

INCLUDES
GenericBag(Ballot)

CONSTANTS
preferences,
value,
sign,
valueballot,
signballot,
plus_minus,
bagvalue,
bagrange,
totalballotvalue

PROPERTIES

$preferences \in Ballot \rightarrow seq(Candidate) \wedge$
$value \in \mathbb{N} \times SIGN \rightarrow \mathbb{N} \wedge$
$sign \in \mathbb{N} \times SIGN \rightarrow SIGN \wedge$
$valueballot \in Ballot \rightarrow \mathbb{N} \wedge$
$signballot \in Ballot \rightarrow SIGN \wedge$
$\forall (pref, val, sgn).(pref \in seq(Candidate) \wedge val \in \mathbb{N} \wedge sgn \in SIGN \Rightarrow ($
$preferences(pref \mapsto val \mapsto sgn) = pref \wedge$
$value(val \mapsto sgn) = val \wedge$
$sign(val \mapsto sgn) = sgn \wedge$
$valueballot(pref \mapsto val \mapsto sgn) = val \wedge$
$signballot(pref \mapsto val \mapsto sgn) = sgn))$

$$
\begin{array}{l}
\mathit{plus_minus} \in ((\mathbb{N} \times \mathit{SIGN}) \times (\mathbb{N} \times \mathit{SIGN})) \rightarrow (\mathbb{N} \times \mathit{SIGN}) \wedge \\
\forall(B1, B2).(B1 \in \mathbb{N} \times \mathit{SIGN} \wedge B2 \in \mathbb{N} \times \mathit{SIGN} \Rightarrow (\\
\quad (\exists(\mathit{val}, \mathit{sgn}).(\mathit{val} \in \mathbb{N} \wedge \mathit{sgn} \in \mathit{SIGN} \wedge \\
\quad\quad ((\mathit{sign}(B1) = \mathit{pos} \wedge \mathit{sign}(B2) = \mathit{pos} \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B1) + \mathit{value}(B2) \wedge \mathit{sgn} = \mathit{pos}) \vee \\
\quad\quad (\mathit{sign}(B1) = \mathit{neg} \wedge \mathit{sign}(B2) = \mathit{neg} \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B1) + \mathit{value}(B2) \wedge \mathit{sgn} = \mathit{neg}) \vee \\
\quad\quad (\mathit{sign}(B1) = \mathit{pos} \wedge \mathit{sign}(B2) = \mathit{neg} \wedge \\
\quad\quad \mathit{value}(B1) >= \mathit{value}(B2) \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B1) - \mathit{value}(B2) \wedge \mathit{sgn} = \mathit{pos}) \vee \\
\quad\quad (\mathit{sign}(B1) = \mathit{pos} \wedge \mathit{sign}(B2) = \mathit{neg} \wedge \\
\quad\quad \mathit{value}(B1) < \mathit{value}(B2) \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B2) - \mathit{value}(B1) \wedge \mathit{sgn} = \mathit{neg}) \vee \\
\quad\quad\quad (\mathit{sign}(B1) = \mathit{neg} \wedge \mathit{sign}(B2) = \mathit{pos} \wedge \\
\quad\quad\quad \mathit{value}(B1) >= \mathit{value}(B2) \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B1) - \mathit{value}(B2) \wedge \mathit{sgn} = \mathit{neg}) \vee \\
\quad\quad (\mathit{sign}(B1) = \mathit{neg} \wedge \mathit{sign}(B2) = \mathit{pos} \wedge \\
\quad\quad \mathit{value}(B1) < \mathit{value}(B2) \wedge \\
\quad\quad \mathit{val} = \mathit{value}(B2) - \mathit{value}(B1) \wedge \mathit{sgn} = \mathit{pos})) \wedge \\
\mathit{plus_minus}(B1, B2) = \mathit{val} \mapsto \mathit{sgn} \;))\;)\;)
\end{array}
$$

$$
\begin{array}{l}
\mathit{bagvalue} \in \mathit{BagBallots} \rightarrow \mathbb{N} \times \mathit{SIGN} \wedge \\
\mathit{bagvalue}() = 0 \mapsto \mathit{pos} \wedge \\
\forall(\mathit{bballot}, \mathit{nn}).(\mathit{bballot} \in \mathit{Ballot} \wedge \mathit{nn} \in \mathbb{N}_1 \Rightarrow \\
\quad \mathit{bagvalue}(\mathit{bballot} \mapsto \mathit{nn}) = \mathit{valueballot}(\mathit{bballot}) \times \mathit{nn} \mapsto \mathit{signballot}(\mathit{bballot})) \wedge \\
\quad \forall(\mathit{ABagBallots}, \mathit{BBagBallots}). \\
\quad\quad (\mathit{ABagBallots} \in \mathit{BagBallots} \wedge \mathit{BBagBallots} \in \mathit{BagBallots} \Rightarrow (\\
\quad\quad \mathit{bagvalue}(\mathit{bagplus}(\mathit{ABagBallots}, \mathit{BBagBallots})) = \\
\quad\quad\quad \mathit{plus_minus}(\mathit{bagvalue}(\mathit{ABagBallots}), \mathit{bagvalue}(\mathit{BBagBallots})) \\
\quad)\;) \\
\wedge \\
\mathit{bagrange} \in (\mathbb{N} \nrightarrow \mathit{BagBallots}) \nrightarrow \mathit{BagBallots} \wedge \\
\mathit{bagrange}(\{\}) = \{\} \wedge \\
\forall(\mathit{ABagBallots}, \mathit{nn}).(\mathit{ABagBallots} \in \mathit{BagBallots} \wedge \mathit{nn} \in \mathbb{N} \Rightarrow (\\
\quad \mathit{bagrange}(\mathit{nn} \mapsto \mathit{ABagBallots}) = \mathit{ABagBallots})) \wedge \\
\quad \forall(\mathit{ff}, \mathit{gg}). \\
\quad\quad (\mathit{ff} \in \mathbb{N} \nrightarrow \mathit{BagBallots} \wedge \mathit{gg} \in \mathbb{N} \nrightarrow \mathit{BagBallots} \wedge \\
\quad\quad \mathit{dom}\ \mathit{ff} \cap \mathit{dom}\ \mathit{gg} = \varnothing \Rightarrow \\
\quad \mathit{bagrange}(\mathit{ff} \cup \mathit{gg}) = \mathit{bagplus}(\mathit{bagrange}(\mathit{ff}), \mathit{bagrange}(\mathit{gg})) \\
)
\end{array}
$$

$$
\begin{array}{l}
\wedge \\
totalballotvalue \in BagBallots \rightarrow \mathbb{N} \times SIGN \ \wedge \\
\forall \, ABagBallots.(ABagBallots \in BagBallots \Rightarrow \\
\qquad totalballotvalue(ABagBallots) = bagvalue(ABagBallots)\ )
\end{array}
$$

MACHINE $PaperBag$

SEES

$GlobalVariables$

CONSTANTS

$Paper_count,$
$Paper_tcount,$
$Paper_bagplus,$
$Paper_bagminus$

PROPERTIES

$Paper_count \in BagPapers \nrightarrow (Paper > - >> NAT) \wedge$
$\forall\, bg.(bg \in (BagPapers) \Rightarrow$
$\quad Paper_count(bg) = \lambda\, xx.(xx \in Paper \mid 0) < +bg)$
$\wedge$
$Paper_tcount \in ((BagPapers) \times Paper) \nrightarrow NAT \wedge$
/ * total function version of count */
$\forall(bg, xx).(bg \in (BagPapers) \wedge xx \in Paper \Rightarrow$
$\quad Paper_tcount(bg \mapsto xx) = (Paper_count(bg))(xx))$
$\wedge$
$Paper_bagplus \in (BagPapers) \times (BagPapers) \nrightarrow (BagPapers) \wedge$
$Paper_bagminus \in (BagPapers) \times (BagPapers) \nrightarrow (BagPapers) \wedge$
$\forall(abag, bbag, xx).(abag \in (BagPapers) \wedge bbag \in (BagPapers) \wedge xx \in Paper \Rightarrow$
$\quad Paper_tcount(Paper_bagplus(abag, bbag), xx) =$
$\qquad Paper_tcount(abag, xx) + Paper_tcount(bbag, xx) \wedge$
$\quad Paper_tcount(Paper_bagminus(abag, bbag), xx) =$
$\qquad max(\{Paper_tcount(abag, xx) - Paper_tcount(bbag, xx), 0\}))$

VARIABLES

$pbag$

INVARIANT

$pbag \in BagPapers$

INITIALISATION

$pbag := \varnothing$

OPERATIONS

$nn \longleftarrow bagcount \ \widehat{=}$

```
    BEGIN
        nn := card(pbag)
    END
END
```

D.2.4 Pre-Processing of Votes

MACHINE *P_Prepare_Ballots (no_cands)*

SEES

GlobalVariables ,
BallotBag ,
PaperBag

CONSTANTS

P_prepare_ballots ,
P_make_bag_ballots ,
P_throwaway_empties ,
P_make_ballot ,
P_find_first_hole_or_dup

PROPERTIES

$P_find_first_hole_or_dup \in Paper \rightarrow \mathbb{N} \wedge$
$\forall\ paper\ .\ (\ paper \in Paper \Rightarrow$
$\quad P_find_first_hole_or_dup\ (\ paper\ ) =$
$\qquad \mathsf{min}\ (\ \{\ nn\ \mid\ nn \in 1\ ..\ no_cands + 1 \wedge$
$\qquad\qquad \mathsf{card}\ (\ paper^{-1}\ [\ \{\ nn\ \}\ ]\ ) \neq 1\ \}\ \}\ )\ )$
$\wedge$
$P_make_ballot \in Paper \rightarrow Ballot \wedge$
$\forall\ paper\ .\ (\ paper \in Paper \Rightarrow$
$\quad preferences\ (\ P_make_ballot\ (\ paper\ )\ ) =$
$\qquad 1\ ..\ P_find_first_hole_or_dup\ (\ paper\ ) - 1 \lhd paper^{-1} \wedge$
$\quad valueballot\ (\ P_make_ballot\ (\ paper\ )\ ) = 1000 \wedge$
$\quad signballot\ (\ P_make_ballot\ (\ paper\ )\ ) = pos\ )$
$\wedge$
$P_throwaway_empties \in BagBallots \rightarrow BagBallots \wedge$
$\forall\ bballot\ .\ (\ bballot \in BagBallots \Rightarrow$
$\quad P_throwaway_empties\ (\ bballot\ ) =$
$\qquad \{\ bb\ \mid\ bb \in \mathsf{dom}\ (\ bballot\ ) \wedge preferences\ (\ bb\ ) = \varnothing\ \} \lhd\!\!\!-\ bballot\ )$
$\wedge$
$P_make_bag_ballots \in BagPapers \rightarrow BagBallots \wedge$
$\forall\ bpapers\ .\ (\ bpapers \in BagPapers \Rightarrow$
$\quad P_make_bag_ballots\ (\ bpapers\ ) =$
$\qquad \{\ bb\ ,\ nn\ \mid\ bb \in Ballot \wedge nn \in \mathbb{N}_1 \wedge$
$\qquad \exists\ pp\ .\ (\ pp \in \mathsf{dom}\ (\ bpapers\ ) \wedge bb =$
$\qquad\quad P_make_ballot\ (\ pp\ ) \wedge nn = bpapers\ (\ pp\ )\ )\ \}\ )$

$\wedge$

$P_prepare_ballots \in BagPapers \rightarrow BagBallots \wedge$
$\forall\ bpapers\ .\ (\ bpapers \in BagPapers \Rightarrow$
$P_prepare_ballots\ (\ bpapers\ ) =$
$P_throwaway_empties\ (\ P_make_bag_ballots\ (\ bpapers\ )\ )\ )$

VARIABLES

allvotes ,
procballots

INVARIANT

$allvotes \in BagPapers \wedge$
$procballots \in BagBallots$

INITIALISATION

$allvotes := \varnothing \ ||$
$procballots := \varnothing$

OPERATIONS

$procballots \longleftarrow processvotes \ \widehat{=}$

BEGIN

$procballots := P_prepare_ballots\ (\ allvotes\ )$

END

END

D.2.5 Counting Functions

MACHINE *Counts*
SEES
GlobalVariables ,
OrderingFunctions ,
BallotBag ,
Bool_TYPE ,
Countingfunctions ,
FindBalanceMins ,
GetNextCandidate

CONSTANTS
Change_Weight ,
Change_All_Weights,
Do_Transferexcl ,
Transfer_Excluded ,
next ,
Do_a_count ,
Non_Transferables ,
Transferables ,
Transfer_Elected,
Prepare_Transbag,
Find_Surplus ,
Next_Pref ,
Setup_First_Count,
Finished ,
Election_Count ,
Add_Action

PROPERTIES
$Change_Weight \in BagBallots \times \mathbb{N} \rightarrow BagBallots \wedge$
$\forall\ (\ bag\ ,\ factor\ )\ .\ (\ bag \in BagBallots \wedge factor \in \mathbb{N} \Rightarrow$
$\quad \exists\ changedbag\ .\ (\ changedbag \in BagBallots \wedge$
$\quad\quad \forall\ (\ bb\ ,\ nn\ )\ .\ (\ bb \in Ballot \wedge nn \in \mathbb{N} \wedge bb \mapsto nn \in bag \Rightarrow$
$\quad\quad\quad \exists\ bb''\ .\ (\ bb'' \in Ballot \wedge$
$\quad\quad\quad\quad valueballot\ (\ bb''\ ) = valueballot\ (\ bb\ ) \times factor \wedge$
$\quad\quad\quad\quad preferences\ (\ bb''\ ) = preferences\ (\ bb\ ) \wedge$
$\quad\quad\quad\quad signballot\ (\ bb''\ ) = signballot\ (\ bb\ ) \wedge$
$\quad\quad\quad\quad bb'' \mapsto nn \in changedbag\ )\ ) \wedge$

$Change_Weight\ (\ bag\ ,\ factor\ ) = changedbag\ \ )\ )$

$\wedge$

$Change_All_Weights \in (\ Candidate \mathrel{+\!\!\!\rightarrow} BagBallots\ ) \times \mathbb{N} \rightarrow$
$(\ Candidate \mathrel{+\!\!\!\rightarrow} BagBallots\ ) \wedge$

$\forall\ (\ transbag\ ,\ factor\ )\ .\ (\ transbag \in Candidate \mathrel{+\!\!\!\rightarrow} BagBallots \wedge$
$factor \in \mathbb{N} \Rightarrow$

$\exists\ weightedbag\ .\ (\ weightedbag \in Candidate \mathrel{+\!\!\!\rightarrow} BagBallots \wedge ($

$\forall\ (\ cc\ ,\ bb\ )\ .\ (\ cc \in \mathsf{dom}\ (\ transbag\ ) \wedge bb \in \mathsf{ran}\ (\ transbag\ ) \wedge$
$cc \mapsto bb \in transbag \Rightarrow$

$\exists\ bb"\ .\ (\ bb" \in BagBallots \wedge$
$bb" = Change_Weight\ (\ bb\ ,\ factor\ ) \wedge$
$cc \mapsto bb" \in weightedbag\ )\ \ ) \wedge$

$Change_All_Weights\ (\ transbag\ ,\ factor\ ) = weightedbag\ )\ \ )\ )$

$\wedge$

$$
\begin{array}{l}
Do_Transferexcl \in Candidate \times VoteMass \times NonTransfers \\
\quad \times \mathbb{N} \times \mathbb{N} \times SetCandidates \rightarrow \\
\quad VoteMass \times NonTransfers \wedge \\
\forall (currcand , vm , nt , count , subcount , continuing) . \\
\ (currcand \in Candidate \wedge vm \in VoteMass \wedge nt \in NonTransfers \wedge \\
\quad count \in \mathbb{N} \wedge subcount \in \mathbb{N} \wedge continuing \in SetCandidates \Rightarrow \\
\qquad \exists (currbag , transferbag , nontransfers , vm" , nt") . \\
\qquad (\\
\qquad currbag \in BagBallots \wedge \\
\qquad transferbag \in Candidate \twoheadrightarrow BagBallots \wedge \\
\qquad vm" \in VoteMass \wedge nt" \in NonTransfers \wedge \\
\qquad currbag = vm (currcand) (subcount) \wedge \\
\qquad transferbag = Prepare_Transbag (currbag , currcand , continuing) \wedge \\
\qquad nontransfers = Non_Transferables (currbag , currcand , continuing) \wedge \\
\qquad \forall cand . (cand \in \mathrm{dom} (transferbag) \Rightarrow \\
\qquad\quad count \in \mathrm{dom} (vm (cand)) \wedge \\
\qquad\quad vm" (cand) = vm (cand) \cup \{ count \mapsto transferbag (cand) \} \vee \\
\qquad\quad (count \notin \mathrm{dom} (vm (cand)) \wedge \\
\qquad\quad vm" (cand) = \\
\qquad\quad vm (cand) \lhd\!\!- \{ count \mapsto bagplus (vm (cand) (count) , \\
\qquad\qquad\qquad transferbag (cand)) \})) \wedge \\
\qquad \forall cand . (cand \in Candidate \wedge cand \in \mathrm{dom} (vm) \wedge \\
\qquad\qquad cand \notin \mathrm{dom} (transferbag) \Rightarrow \\
\qquad\quad vm" (cand) = vm (cand) \wedge \\
\qquad\quad (count \in \mathrm{dom} (nt) \wedge nontransfers \neq \varnothing \wedge \\
\qquad\quad nt" = nt \cup \{ count \mapsto bagplus (nt (count) , nontransfers) \} \vee \\
\qquad\quad (count \in \mathbb{N} \wedge count \notin \mathrm{dom} (vm (currcand)) \wedge \\
\qquad\quad nontransfers \neq \varnothing \wedge nt" = nt \cup \{ count \mapsto nontransfers \}) \vee \\
\qquad\quad (nontransfers = \varnothing \wedge nt" = nt))) \wedge \\
\quad Do_Transferexcl (currcand , vm , nt , count , subcount , continuing) = \\
\qquad\qquad vm" \mapsto nt")) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
\mathit{Transfer_Excluded} \in \mathit{Candidate} \times \mathit{VoteMass} \times \mathit{NonTransfers} \times \\
\qquad\qquad \mathit{DealtWith} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathit{SetCandidates} \rightarrow \\
\qquad\qquad \mathit{VoteMass} \times \mathit{NonTransfers} \wedge \\
\forall \, (\mathit{currcand} , \mathit{vm} , \mathit{nt} , \mathit{dw} , \mathit{count} , \mathit{subcount} , \mathit{quota} , \mathit{allcandidates}) \, . \\
\; (\mathit{currcand} \in \mathit{Candidate} \wedge \mathit{vm} \in \mathit{VoteMass} \wedge \mathit{nt} \in \mathit{NonTransfers} \wedge \\
\; \mathit{dw} \in \mathit{DealtWith} \wedge \mathit{count} \in \mathbb{N} \wedge \mathit{subcount} \in \mathbb{N} \wedge \mathit{quota} \in \mathbb{N} \wedge \\
\; \mathit{allcandidates} \in \mathit{SetCandidates} \Rightarrow \\
\quad \exists \, (\mathit{continuing} , \mathit{maxsubcount}) \, . \, (\mathit{continuing} \in \mathit{SetCandidates} \wedge (\\
\qquad \mathit{continuing} = \\
\qquad\quad \mathit{allcandidates} - \\
\qquad\quad \mathsf{dom} (\mathsf{ran} (\mathit{1} \, .. \, \mathit{count} - \mathit{2} \lhd \mathit{dw})) - \\
\qquad\quad \{ \mathit{cc} \mid \mathit{cc} \in \mathit{allcandidates} \wedge \\
\qquad\qquad \mathit{value} (\mathit{totalvaloffunct} (\mathit{vm} (\mathit{cc}) , \mathit{count} - \mathit{1})) \geq \mathit{quota} \} \wedge \\
\qquad \mathit{maxsubcount} = \mathsf{max} (\mathsf{dom} (\mathit{vm} (\mathit{currcand}))) \wedge \\
\qquad (\mathit{subcount} > \mathit{maxsubcount} \wedge \\
\qquad \exists \, \mathit{vm}'' \, . \, (\mathit{vm}'' \in \mathit{VoteMass} \wedge (\\
\qquad\quad \mathit{vm}'' = \mathit{vm} \lhd\!\!\!- \{ \mathit{currcand} \mapsto \\
\qquad\qquad \mathit{vm} (\mathit{currcand}) \cup \\
\qquad\qquad \{ \mathit{count} \mapsto \\
\qquad\qquad \mathit{Change_Weight} (\mathit{bagrange} (\mathit{vm} (\mathit{currcand})) , \mathit{1}) \} \\
\qquad\qquad\quad \} \wedge \\
\qquad \mathit{Transfer_Excluded} (\mathit{currcand} , \mathit{vm} , \mathit{nt} , \mathit{dw} , \mathit{count} , \mathit{subcount} , \\
\qquad\qquad \mathit{quota} , \mathit{allcandidates}) = \\
\qquad \mathit{vm}'' \mapsto \mathit{nt})) \vee \\
\qquad (\mathit{subcount} \leq \mathit{maxsubcount} \wedge \\
\qquad \exists \, (\mathit{vm}'' , \mathit{nt}'') \, . \, (\mathit{vm}'' \in \mathit{VoteMass} \wedge \mathit{nt}'' \in \mathit{NonTransfers} \wedge \\
\qquad\quad \mathit{vm}'' \mapsto \mathit{nt}'' = \mathit{Do_Transferexcl} (\mathit{currcand} , \mathit{vm} , \mathit{nt} , \mathit{count} , \\
\qquad\qquad\qquad \mathit{subcount} , \mathit{continuing}) \wedge \\
\qquad \mathit{Transfer_Excluded} (\mathit{currcand} , \mathit{vm} , \mathit{nt} , \mathit{dw} , \mathit{count} , \mathit{subcount} , \\
\qquad\qquad\qquad \mathit{quota} , \mathit{allcandidates}) = \\
\qquad \mathit{Transfer_Excluded} (\mathit{currcand} , \mathit{vm}'' , \mathit{nt}'' , \mathit{dw} , \mathit{count} , \\
\qquad\qquad\qquad \mathit{next} (\mathit{subcount} , \mathsf{dom} (\mathit{vm} (\mathit{currcand}))) \\
\qquad\qquad\qquad , \mathit{quota} , \mathit{allcandidates}))))))) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
next \in \mathbb{N} \times \mathbb{P} (\mathbb{N}) \rightarrow \mathbb{N} \wedge \\
\forall (currno , set) . (set \in \mathbb{P} (\mathbb{N}) \wedge currno \in \mathbb{N} \wedge currno \in set \Rightarrow \\
\quad \exists\, nextno . (nextno \in set \cup \{ \mathsf{max} (set) + \mathit{1} \} \wedge \\
\quad\quad currno .. nextno = \{ currno , nextno \} \wedge \\
\quad next (currno , set) = nextno)) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
Do_a_count \in VoteMass \times \mathbb{N} \times DealtWith \times SetCandidates \times \\
\quad NonTransfers \times BagBallots \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times MinDept \times \mathbb{N} \rightarrow \\
\quad VoteMass \times NonTransfers \times DealtWith \wedge \\
\forall \; (\; vm \; , \; count \; , \; dealt_with \; , \; allcandidates \; , \; non_transfers \; , \; bballots \; , \\
\quad no_seats \; , \; mingen \; , \; minschool \; , \; mindept \; , \; quota \;) \; . \\
\quad (\; vm \in VoteMass \wedge count \in \mathbb{N} \wedge dealt_with \in DealtWith \wedge \\
\quad allcandidates \in SetCandidates \wedge non_transfers \in NonTransfers \wedge \\
\quad bballots \in BagBallots \wedge no_seats \in \mathbb{N} \wedge mingen \in \mathbb{N} \wedge \\
\quad minschool \in \mathbb{N} \wedge mindept \in MinDept \wedge quota \in \mathbb{N} \Rightarrow \\
\quad \exists \; (\; continuing \; , \; cand \; , \; act \; , \; vm'' \; , \; nt'' \; , \; dw'' \;) \; . \\
\quad\quad (\; continuing \subseteq allcandidates \wedge cand \in allcandidates \wedge act \in ACT \wedge \\
\quad\quad vm'' \in VoteMass \wedge nt'' \in NonTransfers \wedge dw'' \in DealtWith \wedge \\
\quad\quad continuing = \\
\quad\quad\quad allcandidates \; - \\
\quad\quad\quad \mathsf{dom} \; (\; \mathsf{ran} \; (\; 1 \; .. \; count \; - \; 2 \lhd dealt_with \;) \;) \; - \\
\quad\quad\quad \{ \; cc \;\; | \;\; cc \in allcandidates \wedge \\
\quad\quad\quad\quad value \; (\; totalvaloffunct \; (\; vm \; (\; cc \;) \; , \; count \; - \; 1 \;) \;) \geq quota \; \} \wedge \\
\quad\quad cand \mapsto act = \\
\quad\quad\quad Find_Next_Cand \; (\; vm \; , \; count \; , \; dealt_with \; , \\
\quad\quad\quad\quad allcandidates \; , \; bballots \; , \; no_seats \; , \\
\quad\quad\quad\quad mingen \; , \; minschool \; , \; mindept \; , \; quota \;) \wedge \\
\quad\quad (\; act = elect \wedge \\
\quad\quad vm'' \mapsto nt'' = Transfer_Elected \; (\; cand \; , \; vm \; , \; non_transfers \; , \\
\quad\quad\quad\quad count \; , \; quota \; , \; continuing \;) \\
\quad\quad \vee \\
\quad\quad (\; act = exclude \wedge \\
\quad\quad vm'' \mapsto nt'' = \\
\quad\quad\quad Transfer_Excluded \; (\; cand \; , \; vm \; , \; non_transfers \; , \; dealt_with \; , \\
\quad\quad\quad count \; , \; \mathsf{min} \; (\; \mathsf{dom} \; (\; vm \; (\; cand \;) \;) \;) \; , \; quota \; , \; allcandidates \;) \;) \;) \wedge \\
\quad dw'' = dealt_with \leftarrow (\; cand \mapsto act \;) \wedge \\
\quad Do_a_count \; (\; vm \; , \; count \; , \; dealt_with \; , \; allcandidates \; , \\
\quad\quad\quad non_transfers \; , \; bballots \; , \; no_seats \; , \; mingen \; , \; minschool \; , \\
\quad\quad\quad mindept \; , \; quota \;) = \\
\quad\quad vm'' \mapsto nt'' \mapsto dw'' \;) \;) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
\mathit{Transfer_Elected} \in \mathit{Candidate} \times \mathit{VoteMass} \times \mathit{NonTransfers} \times \\
\qquad\qquad \mathbb{N} \times \mathbb{N} \times \mathit{SetCandidates} \rightarrow \\
\qquad\qquad \mathit{VoteMass} \times \mathit{NonTransfers} \wedge \\
\forall \ (\ \mathit{currcand} \ , \ \mathit{vm} \ , \ \mathit{nt} \ , \ \mathit{count} \ , \ \mathit{quota} \ , \ \mathit{continuing} \ ) \ . \\
\quad (\ \mathit{currcand} \in \mathit{Candidate} \wedge \mathit{vm} \in \mathit{VoteMass} \wedge \mathit{nt} \in \mathit{NonTransfers} \wedge \\
\quad \mathit{count} \in \mathbb{N} \wedge \mathit{quota} \in \mathbb{N} \wedge \mathit{continuing} \in \mathit{SetCandidates} \Rightarrow \\
\quad \exists \ (\ \mathit{lastc} \ , \ \mathit{lastbag} \ , \ \mathit{transferbag} \ , \ \mathit{weightedtransferbag} \ , \ \mathit{surplus} \ , \\
\qquad \mathit{transferables} \ , \ \mathit{transferablevalue} \ , \ \mathit{nontransferables} \ , \ \mathit{nontransaway} \ , \\
\qquad \mathit{nontranstont} \ , \ \mathit{transnont} \ , \ \mathit{transisnt} \ , \ \mathit{transferbag''} \ , \ \mathit{vm''} \ , \ \mathit{nt''} \ ) \ . \\
\qquad (\ \mathit{lastc} \in \mathbb{N} \wedge \mathit{lastc} = \mathsf{max} \ (\ \mathsf{dom} \ (\ \mathit{vm} \ (\ \mathit{currcand} \ ) \ ) \ ) \wedge \\
\qquad \mathit{lastbag} \in \mathit{BagBallots} \wedge \\
\qquad \mathit{lastbag} = \mathit{vm} \ (\ \mathit{currcand} \ ) \ (\ \mathit{lastc} \ ) \wedge \\
\qquad \mathit{transferbag} \in \mathit{Candidate} \mathrel{+\!\!\!\rightarrow} \mathit{BagBallots} \wedge \\
\qquad \mathit{transferbag} = \mathit{Prepare_Transbag} \ (\ \mathit{lastbag} \ , \ \mathit{currcand} \ , \ \mathit{continuing} \ ) \wedge \\
\qquad \mathit{weightedtransferbag} \in \mathit{Candidate} \mathrel{+\!\!\!\rightarrow} \mathit{BagBallots} \wedge \\
\qquad \mathit{weightedtransferbag} = \\
\qquad\quad \mathit{Change_All_Weights} \ (\ \mathit{transferbag} \ , \ \mathit{surplus} \ / \ \mathit{transferablevalue} \ ) \wedge \\
\qquad \mathit{surplus} \in \mathbb{N} \wedge \\
\qquad \mathit{surplus} = \mathit{Find_Surplus} \ (\ \mathit{vm} \ (\ \mathit{currcand} \ ) \ , \ \mathit{quota} \ ) \wedge \\
\qquad \mathit{transferables} \in \mathit{BagBallots} \wedge \\
\qquad \mathit{transferables} = \mathit{Transferables} \ (\ \mathit{lastbag} \ , \ \mathit{currcand} \ , \ \mathit{continuing} \ ) \wedge \\
\qquad \mathit{transferablevalue} \in \mathbb{N} \wedge \\
\qquad \mathit{transferablevalue} = \mathit{value} \ (\ \mathit{bagvalue} \ (\ \mathit{transferables} \ ) \ ) \wedge \\
\qquad \mathit{nontransferables} = \\
\qquad\quad \mathit{Non_Transferables} \ (\ \mathit{lastbag} \ , \ \mathit{currcand} \ , \ \mathit{continuing} \ ) \wedge \\
\qquad \mathit{nontransaway} = \\
\qquad\quad \mathit{Change_Weight} \ (\ \mathit{nontransferables} \ , \\
\qquad\qquad\qquad\quad 1 \times (\ \mathit{surplus} - \mathit{transferablevalue} \ ) \ / \\
\qquad\qquad\qquad\quad \mathit{value} \ (\ \mathit{bagvalue} \ (\ \mathit{nontransferables} \ ) \ ) \ ) \wedge \\
\qquad \mathit{nontranstont} = \mathit{Change_Weight} \ (\ \mathit{nontransaway} \ , \ 1 \ ) \wedge \\
\qquad \mathit{transnont} = \\
\qquad\quad \mathit{Change_Weight} \ (\ \mathit{transferables} \ , \ \mathit{surplus} \times 1 \ / \ \mathit{transferablevalue} \ ) \wedge \\
\qquad \mathit{transisnt} = \mathit{Change_Weight} \ (\ \mathit{transferables} \ , \ 1 \ ) \wedge \\
\qquad \mathit{vm''} \in \mathit{VoteMass} \wedge \mathit{nt''} \in \mathit{NonTransfers} \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad\quad transferbag'' \in Candidate \mathrel{+\!\!\!\rightarrow} BagBallots \land \\
\quad\quad\quad (\ surplus < transferablevalue \land \\
\quad\quad\quad transferbag'' = weightedtransferbag \lor \\
\quad\quad\quad (\ surplus \geq transferablevalue \land transferbag'' = transferbag\ )\ )\ \land \\
\quad\quad \forall\ cand\ .\ (\ cand \in \mathsf{dom}\ (\ transferbag''\ ) \Rightarrow \\
\quad\quad\quad vm''\ (\ cand\ ) = vm\ (\ cand\ ) \cup \\
\quad\quad\quad\quad\quad\quad\quad\quad \{\ count \mapsto transferbag''\ (\ cand\ )\ \}\ )\ \land \\
\quad\quad\quad\quad \{\ cand\ \mid\ cand \in \mathsf{dom}\ (\ transferbag\ )\ \} \cup \\
\quad\quad\quad\quad \{\ currcand\ \} \mathbin{\triangleleft\!\!\!-} vm'' = \\
\quad\quad\quad\quad \{\ cand\ \mid\ cand \in \mathsf{dom}\ (\ transferbag\ )\ \} \\
\quad\quad\quad\quad \cup\ \{\ currcand\ \} \mathbin{\triangleleft\!\!\!-} vm \land \\
\quad\quad (\ surplus \geq transferablevalue \land \\
\quad\quad vm''\ (\ currcand\ ) = \\
\quad\quad\quad\quad\quad vm\ (\ currcand\ ) \cup \{\ count \mapsto transnont\ \} \land \\
\quad\quad nt = nt'' \lor \\
\quad\quad (\ surplus < transferablevalue \land \\
\quad\quad vm''\ (\ currcand\ ) = \\
\quad\quad\quad\quad\quad vm\ (\ currcand\ ) \cup \\
\quad\quad\quad\quad\quad \{\ count \mapsto bagplus\ (\ transisnt\ ,\ nontransaway\ )\ \} \land \\
\quad\quad nt'' = nt \cup \{\ count \mapsto nontranstont\ \}\ )\ )\ \land \\
\quad Transfer_Elected\ (\ currcand\ ,\ vm\ ,\ nt\ ,\ count\ ,\ quota\ ,\ continuing\ ) = \\
\quad\quad vm'' \mapsto nt''\ )\ ) \\
\quad \land
\end{array}
$$

$$
\begin{array}{l}
Prepare_Transbag \in BagBallots \times Candidate \times SetCandidates \rightarrow \\
\quad\quad (\ Candidate \mathrel{+\!\!\!\rightarrow} BagBallots\ ) \land \\
\forall\ (\ inbag\ ,\ currcand\ ,\ continuing\ )\ . \\
\quad (\ inbag \in BagBallots \land currcand \in Candidate \land \\
\quad continuing \in SetCandidates \Rightarrow \\
\quad\quad \exists\ nextcandbag\ .\ (\ nextcandbag \in Candidate \mathrel{+\!\!\!\rightarrow} BagBallots \land \\
\quad\quad\quad \forall\ (\ ballot\ ,\ nn\ )\ . \\
\quad\quad\quad\quad (\ ballot \in Ballot \land nn \in \mathbb{N} \land ballot \mapsto nn \in inbag \land \\
\quad\quad\quad\quad ballot \mapsto currcand \mapsto continuing \in \mathsf{dom}\ (\ Next_Pref\ ) \Rightarrow \\
\quad\quad\quad\quad ballot \mapsto nn \in \\
\quad\quad\quad\quad\quad nextcandbag\ (\ Next_Pref\ (\ ballot\ ,\ currcand\ ,\ continuing\ )\ )\ )\ \land \\
Prepare_Transbag\ (\ inbag\ ,\ currcand\ ,\ continuing\ ) = nextcandbag\ )\ ) \\
\land
\end{array}
$$

$$
\begin{array}{l}
Find_Surplus \in FunctBag \times \mathbb{N} \nrightarrow \mathbb{N} \wedge \\
\forall \, (fbag \, , quota) \, . \, (fbag \in FunctBag \wedge quota \in \mathbb{N} \Rightarrow \\
\quad Find_Surplus \, (fbag \, , quota) = \\
\qquad quota - \\
\qquad value \, (totalvaloffunct \, (fbag \, , \mathsf{max} \, (\mathsf{dom} \, (fbag)))) \;) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
Next_Pref \in Ballot \times Candidate \times SetCandidates \nrightarrow Candidate \wedge \\
\forall \, (ballot \, , currcand \, , continuing) \, . \, (ballot \in Ballot \wedge \\
\quad currcand \in Candidate \wedge continuing \in SetCandidates \wedge \\
\quad (preferences \, (ballot)) ^{-1} \, (currcand) + 1 \, .. \\
\quad \mathsf{card} \, (preferences \, (ballot)) \lhd preferences \, (ballot) \\
\qquad \rhd continuing \neq \varnothing \Rightarrow \\
\quad \exists \, (preference \, , available) \, . \\
\qquad (preference \in \mathsf{iseq} \, (Candidate) \wedge available \in \mathbb{N} \nrightarrow Candidate \wedge \\
\qquad preference = preferences \, (ballot) \wedge \\
\qquad available = \\
\qquad\quad preference ^{-1} \, (currcand) + 1 \, .. \\
\qquad\quad \mathsf{card} \, (preference) \lhd preference \rhd continuing \wedge \\
\quad Next_Pref \, (ballot \, , currcand \, , continuing) = \\
\qquad\quad available \, (\mathsf{min} \, (\mathsf{dom} \, (available)))) \,) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
\quad Transferables \in BagBallots \times Candidate \times SetCandidates \rightarrow \\
\qquad\quad BagBallots \wedge \\
\quad Non_Transferables \in BagBallots \times Candidate \times SetCandidates \rightarrow \\
\qquad\quad BagBallots \wedge \\
\quad \forall\ (\ inbag\ ,\ currcand\ ,\ continuing\ )\ . \\
\qquad (\ inbag \in BagBallots \wedge currcand \in Candidate \wedge \\
\qquad continuing \in SetCandidates \Rightarrow \\
\qquad \exists\ (\ transfers\ ,\ nontransfers\ )\ . \\
\qquad\quad (\ transfers \in BagBallots \wedge nontransfers \in BagBallots \wedge \\
\qquad\quad \forall\ (\ ballot\ ,\ nn\ )\ . \\
\qquad\qquad (\ ballot \in Ballot \wedge nn \in \mathbb{N} \wedge ballot \mapsto nn \in inbag \Rightarrow \\
\qquad\qquad\quad ballot \mapsto currcand \mapsto continuing \in \\
\qquad\qquad\qquad \mathsf{dom}\ (\ Next_Pref\ ) \wedge \\
\qquad\qquad\quad ballot \mapsto nn \in transfers \vee \\
\qquad\qquad\quad (\ ballot \mapsto currcand \mapsto continuing \notin \\
\qquad\qquad\qquad \mathsf{dom}\ (\ Next_Pref\ ) \wedge \\
\qquad\qquad\quad ballot \mapsto nn \in nontransfers\ )\ ) \wedge \\
\qquad Transferables\ (\ inbag\ ,\ currcand\ ,\ continuing\ ) = transfers \wedge \\
\qquad Non_Transferables\ (\ inbag\ ,\ currcand\ ,\ continuing\ ) = nontransfers\ )\ ) \\
\quad \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad Setup_First_Count \in BagBallots \times SetCandidates \rightarrow VoteMass \wedge \\
\quad \forall\ (\ bagballots\ ,\ allcandidates\ )\ . \\
\qquad (\ bagballots \in BagBallots \wedge allcandidates \in SetCandidates \Rightarrow \\
\qquad \exists\ vm''\ .\ (\ vm'' \in VoteMass \wedge \\
\qquad\quad \forall\ (\ ballot\ ,\ nn\ )\ .\ (\ ballot \in Ballot \wedge nn \in \mathbb{N} \wedge \\
\qquad\qquad ballot \mapsto nn \in bagballots \Rightarrow \\
\qquad\qquad \exists\ cand\ .\ (\ cand \in allcandidates \wedge \\
\qquad\qquad\quad \mathsf{first}\ (\ preferences\ (\ ballot\ )\ ) = cand \wedge \\
\qquad\qquad\quad ballot \mapsto nn \in vm''\ (\ cand\ )\ (\ 1\ )\ )\ )\ ) \wedge \\
\qquad Setup_First_Count\ (\ bagballots\ ,\ allcandidates\ ) = vm''\ )\ ) \\
\wedge
\end{array}
$$

```
Finished ∈ DealtWith × SetCandidates × ℕ × ℕ × ℕ × ℕ ×
        MinDept ⇸ BOOL ∧
∀ ( dealt_with , allcandidates , count , no_seats ,
    mingen , minschool , mindept ) .
    ( dealt_with ∈ DealtWith ∧ allcandidates ∈ SetCandidates ∧
    count ∈ ℕ ∧ no_seats ∈ ℕ ∧ mingen ∈ ℕ ∧
    minschool ∈ ℕ ∧ mindept ∈ MinDept ⇒
    ∃ ( elected , excluded , res ) .
        ( elected ∈ SetCandidates ∧ excluded ∈ SetCandidates ∧
         res ∈ BOOL ∧
         elected =
            { cand | cand ∈ Candidate ∧
                ∃ ii . ( ii ∈ 1 .. count − 2 ∧
                fst ( dealt_with ( ii ) ) = cand ∧
                snd ( dealt_with ( ii ) ) = elect ) } ∧
        excluded =
            { cand | cand ∈ Candidate ∧
                ∃ ii . ( ii ∈ 1 .. count − 2 ∧ fst ( dealt_with ( ii ) ) = cand ∧
                snd ( dealt_with ( ii ) ) = exclude ) } ∧
        ( card ( elected ) = no_seats ∨
        ( card ( allcandidates ) − card ( excluded ) = no_seats ∧
        balancepossible ( allcandidates , elected ,
                    allcandidates − excluded − elected , no_seats ,
                    mingen , minschool , mindept ) = TRUE ) ∧
        res = TRUE ∨
        ( card ( elected ) < no_seats ∧
        ( card ( allcandidates ) − card ( excluded ) > no_seats ∨
        balancepossible ( allcandidates , elected ,
                    allcandidates − excluded − elected , no_seats ,
                    mingen , minschool , mindept ) = FALSE ) ∧
        res = FALSE ) ) ∧
    Finished ( dealt_with , allcandidates , count , no_seats ,
                    mingen , minschool , mindept ) =
        res ) )
∧
```

$$
\begin{aligned}
&Add_Action \in \mathsf{iseq}\ (\ Candidate\ ) \times ACT \rightarrow \mathsf{iseq}\ (\ Candidate \times ACT\ ) \wedge \\
&\forall\ (\ scand\ ,\ act\ )\ .\ (\ scand \in \mathsf{iseq}\ (\ Candidate\ ) \wedge act \in ACT \Rightarrow \\
&\quad \exists\ otherscand\ .\ (\ otherscand \in \mathsf{iseq}\ (\ Candidate \times ACT\ ) \wedge \\
&\quad\quad \forall\ cand\ .\ (\ cand \in \mathsf{ran}\ (\ scand\ ) \Rightarrow \\
&\quad\quad\quad otherscand\ (\ scand^{\ -1}\ (\ cand\ )\ ) = cand \mapsto act\ ) \wedge \\
&\quad Add_Action\ (\ scand\ ,\ act\ ) = otherscand\ )\ )
\end{aligned}
$$

$\wedge$

$$
\begin{aligned}
& Election_Count \in VoteMass \times NonTransfers \times DealtWith \times \mathbb{N} \times \\
& \qquad BagBallots \times SetCandidates \times \mathbb{N} \times \mathbb{N} \times MinDept \times \mathbb{N} \times \mathbb{N} \to \\
& \qquad VoteMass \times NonTransfers \times DealtWith \wedge
\end{aligned}
$$

$$
\begin{aligned}
& \forall (vm, nt, dw, count, bagballots, allcandidates, mingen, minschool, \\
& \quad mindept, no_seats, quota). \\
& \quad (vm \in VoteMass \wedge nt \in NonTransfers \wedge dw \in DealtWith \wedge \\
& \quad count \in \mathbb{N} \wedge bagballots \in BagBallots \wedge \\
& \quad allcandidates \in SetCandidates \wedge mingen \in \mathbb{N} \wedge minschool \in \mathbb{N} \wedge \\
& \quad mindept \in MinDept \wedge no_seats \in \mathbb{N} \wedge quota \in \mathbb{N} \Rightarrow \\
& \qquad (Finished(dw, allcandidates, count, no_seats, \\
& \qquad\qquad mingen, minschool, mindept) \; = \; TRUE \; \wedge \\
& \qquad (\; \exists \; (continuing, elected). \\
& \qquad\quad (continuing \in SetCandidates \wedge elected \in SetCandidates \wedge \\
& \qquad\quad continuing = allcandidates - dom(ran(1..count - 1 \lhd dw)) - \\
& \qquad\quad \{cc \mid cc \in allcandidates \wedge \\
& \qquad\qquad\quad value(totalvaloffunct(vm(cc), count - 1)) >= quota\} \wedge \\
& \qquad\quad elected = \{cand \mid cand \in Candidate \wedge \exists\, ii.(ii \in 1 \,..\, count \wedge \\
& \qquad\qquad\qquad fst(dw(ii)) = cand \wedge \\
& \qquad\qquad\qquad snd(dw(ii)) = elect)\} \wedge \\
& \qquad\quad \exists\, dw''.(dw'' \in DealtWith \wedge \\
& \qquad\qquad (\; (card(elected) = no_seats \wedge dw'' = dw) \; \vee \\
& \qquad\qquad (card(elected) < no_seats \wedge \\
& \qquad\qquad card(elected) + card(continuing) = no_seats \wedge \\
& \qquad\qquad dw'' = dw \frown \\
& \qquad\qquad\quad Add_Action(rev(Put_in_order(vm, count, continuing, \\
& \qquad\qquad\qquad allcandidates, bagballots, card(allcandidates))), elect) \wedge \\
& \qquad Election_Count(vm, nt, dw, count, bagballots, allcandidates, \\
& \qquad\qquad mingen, minschool, mindept, no_seats, quota) = \\
& \qquad\quad vm \mapsto nt \mapsto dw'') \\
& \qquad) \,)) \,) \\
& \qquad \vee \\
& \qquad (Finished(dw, allcandidates, count, no_seats, \\
& \qquad\qquad mingen, minschool, mindept) = FALSE \wedge \\
& \qquad (\exists (vm'', nt'', dw''). \\
& \qquad\quad (vm'' \in VoteMass \wedge nt'' \in NonTransfers \wedge dw'' \in DealtWith \wedge \\
& \qquad\qquad Do_a_count(vm, count, dw, allcandidates, nt, bagballots, \\
& \qquad\qquad\qquad no_seats, mingen, minschool, mindept, quota) \; = \\
& \qquad\qquad\quad vm'' \mapsto nt'' \mapsto dw'' \wedge
\end{aligned}
$$

$Election_Count(vm, nt, dw, count, bagballots, allcandidates,$
$mingen, minschool, mindept, no_seats, quota) =$
$Election_Count(vm'', nt'', dw'', count + 1, bagballots, allcandidates,$
$mingen, minschool, mindept, no_seats, quota)$)))

)

)

MACHINE *Countingfunctions*
/* This contains functions to count the total value of a
candidate's votes and a function to calculate the quota */

SEES
GlobalVariables ,
BallotBag

CONSTANTS
totalvaloffunct ,
Find_Quota

PROPERTIES

$$
\begin{array}{l}
totalvaloffunct \in FunctBag \times \mathbb{N} \to \mathbb{N} \times SIGN \wedge \\
\forall\ (\ countno\ ,\ candfunct\ )\ .\ (\ countno \in \mathbb{N} \wedge candfunct \in FunctBag \Rightarrow \\
\quad totalvaloffunct\ (\ candfunct\ ,\ countno\ ) = \\
\quad\quad bagvalue\ (\ bagrange\ (\ 1\ ..\ countno \lhd candfunct\ )\ )\ \ ) \\
\wedge \\
Find_Quota \in BagBallots \times \mathbb{N} \to \mathbb{N} \wedge \\
\forall\ (\ ballots\ ,\ no_seats\ )\ .\ (\ ballots \in BagBallots \wedge no_seats \in \mathbb{N} \Rightarrow \\
\quad Find_Quota\ (\ ballots\ ,\ no_seats\ ) = \\
\quad\quad (\ bagvalue\ (\ ballots\ )\ )\ /\ (\ no_seats + 1\ ) + 1\ )
\end{array}
$$

Cross-references

BagBallots	*GlobalVariables*	DEFINITIONS
SIGN	*GlobalVariables*	SETS
bagrange	*BallotBag*	CONSTANTS
bagvalue	*BallotBag*	CONSTANTS
bagvalue	*BallotBag*	CONSTANTS

DEFINITIONS
$FunctBag \;==\; \mathbb{N} \mathrel{+\!\!\!\!\rightarrow} BagBallots;$
$CandsBallots \;==\; \mathbb{N} \mathrel{+\!\!\!\!\rightarrow} (\ \mathbb{N} \mathrel{+\!\!\!\!\rightarrow} BagBallots\ )$
END

Cross-references for Countingfunctions

BagBallots	*GlobalVariables*	DEFINITIONS
BallotBag		MACHINE
GlobalVariables		MACHINE
SIGN	*GlobalVariables*	SETS
bagrange	*BallotBag*	CONSTANTS
bagvalue	*BallotBag*	CONSTANTS
BallotBag		CONSTANTS

D.2.6 Ordering Functions

MACHINE *GetNextCandidate*

SEES

$GlobalVariables$,
$OrderingFunctions$,
$BallotBag$,
$Bool_TYPE$,
$Countingfunctions$,
$FindBalanceMins$

CONSTANTS

fst, snd,
$First_count_over$,
$Get_next_over_quota$,
$Get_next_to_exclude$,
$Find_Next_Cand$

PROPERTIES

$fst \in (Candidate \times ACT) \rightarrow Candidate \wedge$
$snd \in (Candidate \times ACT) \rightarrow ACT \wedge$
$\forall (cc, ac).(cc \in Candidate \wedge ac \in ACT \Rightarrow$
$\quad fst(cc \mapsto ac) = cc \wedge$
$\quad snd(cc \mapsto ac) = ac)$
$\wedge$
$First_count_over \in VoteMass \times Candidate \times \mathbb{N} \nrightarrow \mathbb{N} \wedge$
/*Partial function - only applicable when candidate is over quota*/
$\forall (vm, cand, quota, no_cands).$
$\quad (vm \in VoteMass \wedge cand \in Candidate \wedge quota \in \mathbb{N} \wedge no_cands \in \mathbb{N} \Rightarrow$
$\quad First_count_over(vm, cand, quota) =$
$\quad\quad min(\{nn \mid nn \in \mathbb{N} \wedge$
$\quad\quad\quad value(totalvaloffunct(vm(cand), nn)) >= quota\})$
$\quad)$

$$
\begin{array}{l}
\wedge \\
Get_next_over_quota \in (VoteMass \times \mathbb{N} \times SetCandidates \times \\
\quad SetCandidates \times BagBallots \times \mathbb{N} \times \mathbb{N} \times \\
\quad \mathbb{N} \times \mathbb{N} \times MinDept \times SetCandidates \times \\
\quad SetCandidates \times \mathbb{N}) \\
\quad \twoheadrightarrow\!\!\!\!\!+\ (Candidate \times ACT) \ \wedge \\
\forall (vm, count, overquota, allcands, bballots, no_cands, \\
\quad quota, mingen, minschool, mindept, elected, continuing, no_seats). \\
\quad (vm \in VoteMass \wedge count \in \mathbb{N} \wedge overquota \in SetCandidates \wedge \\
\quad allcands \in SetCandidates \wedge bballots \in BagBallots \wedge \\
\quad no_cands \in \mathbb{N} \wedge quota \in \mathbb{N} \wedge mingen \in \mathbb{N} \wedge \\
\quad minschool \in \mathbb{N} \wedge mindept \in MinDept \wedge elected \in SetCandidates \wedge \\
\quad continuing \in SetCandidates \wedge no_seats \in \mathbb{N} \Rightarrow \\
\qquad (\exists\ (fcountover, earliest, cand, do). \\
\qquad\quad (fcountover \in Candidate \twoheadrightarrow\!\!\!\!\!+\ \mathbb{N} \wedge \\
\qquad\quad fcountover = \lambda\, candi.(candi \in overquota \mid \\
\qquad\qquad First_count_over(vm, candi, quota)) \wedge \\
\qquad\quad earliest = min(ranfcountover) \wedge \\
\qquad\quad cand = last(Put_in_order(vm, earliest, overquota, \\
\qquad\qquad allcands, bballots, no_cands)) \wedge \\
\qquad\quad do \in ACT \wedge \\
\qquad\qquad ((balancepossible(allcands, elected \cup cand, \\
\qquad\qquad\quad continuing - cand, no_seats, mingen, \\
\qquad\qquad\quad minschool, mindept) = TRUE \wedge \\
\qquad\qquad do = elect) \\
\qquad\qquad \vee \\
\qquad\qquad (balancepossible(allcands, elected \cup cand, \\
\qquad\qquad\quad continuing - cand, no_seats, mingen, \\
\qquad\qquad\quad minschool, mindept) = FALSE \wedge \\
\qquad\qquad do = exclude)) \wedge \\
\quad Get_next_over_quota(vm, count, overquota, allcands, bballots, \\
\qquad\quad no_cands, quota, mingen, minschool, mindept, elected, \\
\qquad\quad continuing, no_seats) = \\
\qquad\quad cand \mapsto do \\
\quad)) \\
)
\end{array}
$$

$$
\begin{array}{l}
\wedge \\
Get_next_to_exclude \in (\,VoteMass \times \mathbb{N} \times \\
\quad\quad SetCandidates \times SetCandidates \times SetCandidates \times \\
\quad\quad BagBallots \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times MinDept) \rightarrow \\
\quad (Candidate \times ACT) \wedge \\
\text{/*need to insist on balance*/} \\
\forall (vm, count, elected, continuing, allcands, bballots, no_cands, \\
\quad\quad no_seats, mingen, minschool, mindept). \\
\quad (vm \in VoteMass \wedge count \in \mathbb{N} \wedge elected \in SetCandidates \wedge \\
\quad continuing \in SetCandidates \wedge allcands \in SetCandidates \wedge \\
\quad bballots \in BagBallots \wedge no_cands \in \mathbb{N} \wedge no_seats \in \mathbb{N} \wedge \\
\quad mingen \in \mathbb{N} \wedge minschool \in \mathbb{N} \wedge mindept \in MinDept \Rightarrow \\
\quad\quad \exists\, ordered.(ordered \in seq(Candidate) \wedge \\
\quad\quad\quad ordered = Put_in_order(vm, count, continuing, allcands, \\
\quad\quad\quad\quad\quad bballots, no_cands) \wedge \\
\quad\quad\quad \exists\, pos.(pos \in 1 \,..\, card(ordered) \wedge \\
\quad\quad\quad\quad \forall\, pp.(pp \in 1 \,..\, pos \Rightarrow \\
\quad\quad\quad\quad\quad balancepossible(allcands, elected, \\
\quad\quad\quad\quad\quad\quad continuing - ordered(pp), no_seats, mingen, \\
\quad\quad\quad\quad\quad\quad minschool, mindept) = FALSE) \wedge \\
\quad\quad\quad\quad\quad balancepossible(allcands, elected, \\
\quad\quad\quad\quad\quad\quad continuing - ordered(pos), no_seats, \\
\quad\quad\quad\quad\quad\quad mingen, minschool, mindept) = TRUE \wedge \\
\quad Get_next_to_exclude(vm, count, elected, continuing, allcands, bballots, \\
\quad\quad no_cands, no_seats, mingen, minschool, mindept) = \\
\quad\quad\quad ordered(pos) \mapsto exclude) \\
\quad\quad)) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
Find_Next_Cand \in (VoteMass \times \mathbb{N} \times DealtWith \times \\
\quad SetCandidates \times BagBallots \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \\
\quad MinDept \times \mathbb{N}) \rightarrow \\
\quad (Candidate \times ACT) \wedge \\
\forall (vm, count, dealt_with, allcandidates, bballots, no_seats, \\
\quad mingen, minschool, mindept, quota). \\
\quad (vm \in VoteMass \wedge count \in \mathbb{N} \wedge dealt_with \in DealtWith \wedge \\
\quad\quad allcandidates \in SetCandidates \wedge bballots \in BagBallots \wedge \\
\quad\quad no_seats \in \mathbb{N} \wedge mingen \in \mathbb{N} \wedge minschool \in \mathbb{N} \wedge \\
\quad\quad mindept \in MinDept \wedge quota \in \mathbb{N} \Rightarrow \\
\quad\quad\quad (\exists (overquota, continuing, elected, cand, ac). \\
\quad\quad\quad\quad (overquota \in SetCandidates \wedge \\
\quad\quad\quad\quad continuing \in SetCandidates \wedge cand \in Candidate \wedge \\
\quad\quad\quad\quad ac \in ACT \wedge elected \in SetCandidates \wedge \\
\quad\quad\quad\quad overquota = \{cc \mid cc \in allcandidates \wedge \\
\quad\quad\quad\quad\quad\quad value(totalvaloffunct(vm(cc), count - 1)) \\
\quad\quad\quad\quad\quad\quad\quad >= quota\} - \\
\quad\quad\quad\quad dom(ran(1 \mathrel{..} count - 1 \lhd dealt_with)) \wedge \\
\quad\quad\quad\quad continuing = \\
\quad\quad\quad\quad\quad allcandidates - \\
\quad\quad\quad\quad\quad dom(ran(1 \mathrel{..} count - 1 \lhd dealt_with)) - \\
\quad\quad\quad\quad\quad \{cc \mid cc \in allcandidates \wedge \\
\quad\quad\quad\quad\quad\quad\quad value(totalvaloffunct(vm(cc), count - 1)) \\
\quad\quad\quad\quad\quad\quad\quad >= quota\} \wedge \\
\quad\quad\quad\quad elected = \{cand \mid cand \in Candidate \wedge \\
\quad\quad\quad\quad\quad\quad \exists\, ii.(ii \in dom\ dealt_with \wedge \\
\quad\quad\quad\quad\quad\quad fst(dealt_with(ii)) = cand \wedge \\
\quad\quad\quad\quad\quad\quad snd(dealt_with(ii)) = elect)\} \wedge \\
\quad\quad\quad\quad ((overquota \neq \varnothing \wedge \\
\quad\quad\quad\quad cand \mapsto ac = \\
\quad\quad\quad\quad\quad Get_next_over_quota(vm, count - 1, overquota, \\
\quad\quad\quad\quad\quad allcandidates, bballots, card(allcandidates), quota, \\
\quad\quad\quad\quad\quad mingen, minschool, mindept, elected, continuing, \\
\quad\quad\quad\quad\quad no_seats)\ ) \vee \\
\quad\quad\quad\quad (overquota = \varnothing \wedge \\
\quad\quad\quad\quad cand \mapsto ac = \\
\quad\quad\quad\quad\quad Get_next_to_exclude(vm, count - 1, elected, \\
\quad\quad\quad\quad\quad continuing, allcandidates, bballots, card(allcandidates), \\
\quad\quad\quad\quad\quad no_seats, mingen, minschool, mindept) \\
\quad\quad\quad)) \wedge
\end{array}
$$

$Find_Next_Cand(vm, count, dealt_with, allcandidates, bballots,$
$no_seats, mingen, minschool, mindept, quota) =$
$cand \mapsto ac))$

)

END /* Machine *GetNextCandidate**/

MACHINE *OrderingFunctions*

SEES

$Bool_TYPE$,
$BallotBag$,
$GlobalVariables$,
$Countingfunctions$,
$FindBalanceMins$

CONSTANTS

$VoteMassatPref$,
$Order_of_Preferences$,
Put_in_order

PROPERTIES

$VoteMassatPref \in BagBallots \times SetCandidates \times \mathbb{N} \rightarrow VoteMass \wedge$
$\forall(bballots, allcandidates, preference).(bballots \in BagBallots \wedge$
$\quad allcandidates \in SetCandidates \wedge preference \in \mathbb{N} \Rightarrow$
$\quad \exists\, vm.(vm \in VoteMass \wedge$
$\quad\quad \forall(bb, nn).(bb \in Ballot \wedge nn \in \mathbb{N} \wedge bb \mapsto nn \in bballots \Rightarrow$
$\quad\quad\quad \exists\, cand.(cand \in allcandidates \wedge$
$\quad\quad\quad\quad preferences(bb)(preference) = cand \wedge$
$\quad\quad\quad\quad bb \mapsto nn \in vm(cand)(1)\ ) \wedge$
$\quad VoteMassatPref(bballots, allcandidates, preference) = vm\ )\ )\ )$

$\wedge$

```
Order_of_Preferences ∈ BagBallots × SetCandidates × ℕ →
        iseq(Candidate) ∧
/* After the first count, the ballots are arranged in 'order of
preferences' according to "Rules for Academic Council Election
(Academic Members)" Rule 2. This order of preference is used
later in case of ties between candidates. */
∀(bballots, allcandidates, no_cands).(bballots ∈ BagBallots ∧
        allcandidates ∈ SetCandidates ∧ no_cands ∈ ℕ ⇒
    ∃ orderedseq.(orderedseq ∈ iseqCandidate ∧ ran orderedseq = allcandidates ∧
        ∀(ii, jj).(ii ∈ dom orderedseq ∧ jj ∈ dom orderedseq ∧ ii < jj ⇒
            (∃ pref.(pref ∈ 1 .. no_cands ∧
                /* pref is the first time there's a diffence*/
                (value(totalvaloffunct(VoteMassatPref(
                        bballots, allcandidates, pref)(orderedseq(ii)), 1)) >
                value(totalvaloffunct(VoteMassatPref(
                        bballots, allcandidates, pref)(orderedseq(jj)), 1)))
                ∧
                ∀ pp.(pp ∈ 1 .. pref − 1 ⇒
                        (value(totalvaloffunct(VoteMassatPref(
                            bballots, allcandidates, pp)(orderedseq(ii)), 1)) =
                        value(totalvaloffunct(VoteMassatPref(
                            bballots, allcandidates, pp)(orderedseq(jj)), 1))))))
            ∨
            /* If all equal, 'draw lots' */
            (∀ pp.(pp ∈ 1 .. no_cands ⇒
                (value(totalvaloffunct(VoteMassatPref(
                        bballots, allcandidates, pp)(orderedseq(ii)), 1)) =
                value(totalvaloffunct(VoteMassatPref(
                        bballots, allcandidates, pp)(orderedseq(jj)), 1)))))
                )/ * or * / ∧
    Order_of_Preferences(bballots, allcandidates, no_cands) = orderedseq
    ))
∧
```

$$
\begin{array}{l}
Put_in_order \in (\,VoteMass \times \mathbb{N} \times SetCandidates \times \\
\qquad\qquad SetCandidates \times BagBallots \times \mathbb{N}) \rightarrow \\
\qquad iseq(Candidate) \wedge \\
\quad \forall (vm, count, setcands, allcands, bballots, no_cands). \\
\qquad vm \in VoteMass \wedge count \in \mathbb{N} \wedge setcands \in SetCandidates \wedge \\
\qquad allcands \in SetCandidates \wedge ballots \in BagBallots \wedge no_cands \in \mathbb{N} \Rightarrow \\
\qquad\quad \exists\, orderedseq.(orderedseq \in iseq(Candidate) \wedge \\
\qquad\qquad ran\ orderedseq = setcands \wedge \\
\qquad\qquad \forall (ii, jj).(ii \in dom\ orderedseq \wedge jj \in dom\ orderedseq \wedge ii < jj \Rightarrow \\
\qquad\qquad\quad (value(totalvaloffunct(vm(orderedseq(ii)), count)) < \\
\qquad\qquad\quad value(totalvaloffunct(vm(orderedseq(jj)), count))) \\
\qquad\qquad\quad \vee \\
\qquad\qquad\quad (value(totalvaloffunct(vm(orderedseq(ii)), count)) = \\
\qquad\qquad\quad value(totalvaloffunct(vm(orderedseq(jj)), count)) \wedge \\
\qquad\qquad\quad (\exists\, disting_c.(disting_c \in 1 \,..\, count \wedge \\
\qquad\qquad\qquad value(totalvaloffunct(vm(orderedseq(ii)), disting_c)) < \\
\qquad\qquad\qquad value(totalvaloffunct(vm(orderedseq(jj)), disting_c)) \wedge \\
\qquad\qquad\qquad \forall\, cc.(cc \in 1 \,..\, disting_c - 1 \Rightarrow \\
\qquad\qquad\qquad\quad value(totalvaloffunct(vm(orderedseq(ii)), cc)) = \\
\qquad\qquad\qquad\quad value(totalvaloffunct(vm(orderedseq(jj)), cc)) \\
\qquad\qquad\qquad\quad) \\
\qquad\qquad\qquad)) \\
\qquad\qquad\quad \vee \\
\qquad\qquad\quad (\forall\, cc.(cc \in 1 \,..\, count \Rightarrow \\
\qquad\qquad\qquad value(totalvaloffunct(vm(orderedseq(ii)), cc)) = \\
\qquad\qquad\qquad value(totalvaloffunct(vm(orderedseq(jj)), cc))) \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\quad Order_of_Preferences(bballots, allcands, no_cands) \\
\qquad\qquad\qquad\quad (orderedseq(ii)) < \\
\qquad\qquad\quad Order_of_Preferences(bballots, allcands, \\
\qquad\qquad\qquad\quad no_cands)\ (orderedseq(jj)) \\
\qquad\qquad)\,) \\
\qquad\quad) \wedge \\
\qquad Put_in_order(vm, count, setcands, allcands, bballots, no_cands) = \\
\qquad\qquad\qquad\qquad orderedseq\ \ )\ \ )
\end{array}
$$

END

D.2.7 Candidate Balance Functions

MACHINE$FindBalanceMins$

SEES

$GlobalVariables$,
$Bool_TYPE$

CONSTANTS

$findschoolmin$,
num_from_school,
num_from_gender,
$balancepossible$

PROPERTIES

$$
\begin{array}{l}
findschoolmin \in (SetCandidates \times SCHOOL \times \mathbb{N}) \rightarrow \mathbb{N} \wedge \\
\forall (AllCandidates, cschool, minschool).(AllCandidates \in SetCandidates \wedge \\
\qquad\qquad cschool \in SCHOOL \wedge \\
\qquad\qquad minschool \in \mathbb{N} \Rightarrow \\
\qquad findschoolmin(AllCandidates, cschool, minschool) = \\
\qquad\qquad min(\{card\{cc \mid cc \in AllCandidates \wedge school(cc) = cschool\}, \\
\qquad\qquad\quad minschool\}) \\
\qquad) \\
\wedge \\
num_from_school \in (SetCandidates \times SCHOOL) \rightarrow \mathbb{N} \wedge \\
\forall (cands, sch).(cands \in SetCandidates \wedge sch \in SCHOOL \Rightarrow \\
\qquad num_from_school(cands, sch) = \\
\qquad\qquad card(\{cc \mid cc \in cands \wedge school(cc) = sch\ \}\ ) \\
) \\
\wedge \\
num_from_gender \in (SetCandidates \times GENDER) \rightarrow \mathbb{N} \wedge \\
\forall (cands, gen).(cands \in SetCandidates \wedge gen \in GENDER \Rightarrow \\
\qquad num_from_gender(cands, gen) = \\
\qquad\qquad card(cc \mid cc \in cands \wedge gender(cc) = gen) \\
\qquad) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
\quad balancepossible \in (SetCandidates \times SetCandidates \times SetCandidates \times \\
\qquad\qquad\qquad \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times MinDept) \quad \rightarrow \\
\qquad\quad BOOL \quad \wedge \\
\forall (AllCandidates, elected, continuing, no_seats, mingen, minschool, mindept). \\
\quad (AllCandidates \in SetCandidates \wedge \\
\quad elected \in SetCandidates \wedge continuing \in SetCandidates \wedge \\
\quad no_seats \in \mathbb{N} \wedge mingen \in \mathbb{N} \wedge minschool \in \mathbb{N} \wedge \\
\quad mindept \in MinDept \quad \Rightarrow \\
\qquad balancepossible(AllCandidates, elected, continuing, no_seats, \\
\qquad\quad mingen, minschool, mindept) \quad = \\
\qquad bool(\\
\qquad\quad \exists\, tobeelected.(tobeelected \in SetCandidates \wedge \\
\qquad\quad tobeelected \notin continuing \wedge \\
\qquad\quad card(elected) + card(tobeelected) = no_seats \wedge \\
\qquad\quad \forall\, gen.(gen \in GENDER \Rightarrow \\
\qquad\qquad num_from_gender(elected \cup tobeelected, gen) >= mingen) \wedge \\
\qquad\qquad \forall\, sch.(sch \in SCHOOL \Rightarrow \\
\qquad\qquad\quad num_from_school(tobeelected \cup elected, sch) >= \\
\qquad\qquad findschoolmin(AllCandidates, sch, minschool)) \wedge \\
\qquad\quad \forall\, dpt.(dpt \in DEPARTMENT \Rightarrow \\
\qquad\qquad card(\{cc \mid cc \in elected \cup tobeelected \wedge \\
\qquad\qquad\qquad department(cc) = dpt\}) \quad >= \\
\qquad\qquad max(mindept(dpt), 0)) \\
\qquad\quad) \\
\qquad)\ /\text{*bool */} \\
)
\end{array}
$$

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Aho, Upcroft & Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[4] R.J.R. Back & M.J. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica,* 35:921-949, 1998.

[5] B–Core(UK)Ltd, *B–Toolkit User's Manual*, B–Core(UK) Ltd. 1997.

[6] R. Bird & P. Walder, *Introduction to Functional Programming*, Prentice Hall, 1988.

[7] U. Breyman. *Designing Components with the C++ STL: A New Approach to Programming.* Addison Wesley, 1998.

[8] M.J. Butler. Calculational Derivation of Pointer Algorithms from Tree Operations. *Science of Computer Programming*, 33 (1999) 221–260.

[9] M.J. Butler & M. Meagher. Performing Algorithmic Refinement before Data Refinement in B. In *Conference Proceedings - ZB2000:Formal Specification and Development in Z and B*, pages 324 - 343, Lecture Notes in Computer Science, 1878, Springer, 2000.

[10] B. Dehbonei & F. Mejia. Formal Development of Safety-critical Software Systems in Railway Signalling. In M.G. Hinchey & J.P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall, 1995.

[11] Derbyshire & Derbyshire. *Political Systems of the World (2nd Edition).* Oxford Helicon, 1996. ISBN 185 986 1148.

[12] E.J. Dijkstra. *A Discipline of Programming* . Prentice-Hall, 1976.

[13] R. Fraer. Minimum Spanning Tree. In E. Sekerinski & K. Sere, editors. *Program Development by Refinement. Case Studies using the B Method.* Formal Approaches to Computing and Information Technology. Springer, 1998.

[14] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science, Springer, 1989.

[15] J.P. Hoare.Application of the B-Method to CICS. In M.G. Hinchey & J.P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall, 1995.

[16] C.A.R Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[17] http://www.sgi.com/Technology/STL

[18] C.B. Jones, *Systematic Software Development using VDM (2nd Edition)*, Prentice Hall, 1990.

[19] S. King. Z and the refinement calculus. In D. Bjorner, C.A.R. Hoare and H. Langmaack, editors, *VDM'90:VDM and Z – Formal Methods in Software Development, Kiel,* volume 428 of *Lecture Notes in Computer Science.* Springer, 1990.

[20] K. Lano. *The B Language and Method - A Guide to Practical Formal Development.* Formal Approaches to Computing and Information Technology. Springer, 1996.

[21] D. MacQueen, R. Harper & R. Milner. *Standard ML.* Technical Report ECS–LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[22] C.C. Morgan & T. Vickers, editors. *On the Refinement Calculus.* Formal Approaches to Computing and Information Technology. Springer, 1994.

[23] C.C. Morgan. *Programming from Specifications (2nd Edition).* Prentice–Hall, 1994.

[24] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[25] P. Mukherjee & B.A. Wichmann. STV: A Case Stuy in VDM. National Physical Laboratory, Teddington, UK, 1993.

[26] M.R. Poppleton. The Single Transferable Voting System: Functional Decomposition in Formal Specification. From - *Conference Proceedings – IWFM'97 1st Irish Workshop in Formal Methods* , Pages 1 - 18, 1997.

[27] K.A. Robinson. Introduction to the B Method. From *Program Development by Refinement*, E. Sekerinski and K. Sere(Eds), Springer, 1999.

[28] Seanad *É*ireann. Seanad General Election, February 1993, Government Publications, 1993.

[29] I. Sorenson to M. Meagher - Private Correspondence(e-mail), 12/11/99.

[30] J. Michael Spivey. *The Z Notation: a Reference Manual.* Prentice Hall, 2nd edition, 1992.

[31] M. Waldén. Distributed Load Balancing. In E. Sekerinski & K. Sere, editors. *Program Development by Refinement. Case Studies using the B Method.* Formal Approaches to Computing and Information Technology. Springer, 1998.

[32] J. von Wright. The lattice of Data Refinement. *Acta Inform.*, 31 (2) (1994) 105–135.

[33] J. Woodcock & J. Davies. *Using Z, Specification, Refinement and Proof,* Prentice Hall, 1996.

[34] J.B. Wordsworth. *Software Engineering with B.* Addison-Wesley,1996.