Performing Algorithmic Refinement before Data Refinement in B

Michael Butler and Mairead Meagher

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2000-5
May 2000

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Performing Algorithmic Refinement before Data Refinement in B

Michael Butler[1] and Mairead Meagher[2]

[1]Department of Electronics & Computer Science,
University of Southampton, Southampton SO17 1BJ,
United Kingdom
M.J.Butler@ecs.soton.ac.uk

[2]Department of Physical & Quantitative Sciences,
School of Science,
Waterford Institute of Technology,
Waterford, Ireland
mmeagher@wit.ie

June 13, 2000

Abstract

Algorithmic Refinement is part of the theory of the B method both at the refinement and implementation stages. It a sign of how little loop introduction is used in practice at the refinement stage that neither the B-Toolkit nor Atelier-B provide support for loop introduction until the implementation stage. This paper examines the use of algorithmic refinement in general before data refinement. This involves extending the usual scope of data refinement which usually happens before algorithmic refinement. Two case studies are used to compare and contrast the application of algorithmic refinement before data refinement and vice versa. Some extensions are needed in the B-Toolkit to implement this style (i.e., algorithmic before data refinement) and are proposed. Some workarounds are also presented when appropriate.

1 Introduction

We deal with systems developed using the B method. 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 machine. Refining a machine means making it less abstract whilst preserving the previous refined machine's properties. There are two main types of such refinements. These are data refinement and algorithmic 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.

In this paper, we examine the relative merits of introducing algorithmic refinement before data refinement and vice versa. It seems that most developments using B take the approach of data refinement first. In each of our case studies, we illustrate the two approaches. When we introduce algorithmic refinement before data refinement, we use rules on the distribution of data refinement through an algorithmically refined structure as used in [5, 8, 9, 11, 12].

The standard approach in the development of systems using the B method is to apply data refinement and then proceed with algorithmic refinement on the concrete data types. If we take the example of an operation which changes state, then using this approach the first refinement(s) will involve introducing data refinements into the B specification, i.e., the form of the state changed by the operation has become more concrete. Once this is done, attention is turned to 'how' the specification, based on the concrete data, will be implemented. Future refinements will introduce correctness-preserving programming language-like structures, i.e., algorithmic refinements.

An alternative approach is to apply algorithmic refinement immediately on the abstract data types. Data refinement is then applied to the algorithms based on abstract data types. This style, although part of the theory and literature of the B method [1] is not directly implementable in either of the B-Toolkit or Atelier-B. Specifically, loop introduction is not allowed until the implementation stage. This suggests that this style is neither used nor sought in practice. If we again take the example of the operation which changes state, algorithm-like structures (usually a loop including sequential statements) are immediately introduced. So we have, very early, attended to 'how' the implementation might be achieved. Next, we change the form of the state being changed, using distribution laws [5, 8, 9, 11, 12] which tell us under which circumstances the transformations from one statement to another are correctness-preserving under the current data refinement.

2 Algorithmic Refinement before Data Refinement

We examine the approach of applying algorithmic refinement early, before data refinement. This leads to program-like structures based on abstract data types. We need to change what the program-like structures work on, i.e., bring the data closer to concrete, implementable data types. So we need to state what we mean by both algorithmic and data refinement.

Data refinement involves replacing abstract program variables with concrete program variables preserving an abstraction relation between them. For statement S and postcondition Q, $[S]Q$ represents the weakest-precondition under which S is guaranteed to terminate and establish Q [1]. Let S be a statement with abstract program variables a, and let T be a statement with concrete variables c. S is data-refined by T under abstraction relation R, written $S \sqsubseteq_R T$, if the following holds [1]:

$$R \wedge [S]true \quad \Rightarrow \quad [T]\neg[S]\neg R.$$

We also make use of the *least data-refinement* of a statement [12]. 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 $\mathcal{D}_R^{a,c}[\![\ S\ ]\!]$, that is:

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

Rules for distributing $\mathcal{D}_R^{a,c}$ through the structure of S may be found in [5, 8, 9, 11, 12]. Some of these rules are repeated in Fig. 1. The first rule shows that $\mathcal{D}_R^{a,c}$ distributes through sequential composition. The second and third rules 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 fourth rule shows the conditions under which nondeterministic assignments may be data refined. The fifth rule deals with the data refinement of blocks with local variables.

The sixth and final rule deals with a special type of operation. Let *Aproc* be an abstract parameterised operation whose only effect is to calculate and return the value of the output variable. We treat such operations as procedures, i.e., named pieces of code as in the refinement calculus [10]. The rule says that if the body of *Aproc* is data-refined by the body of *Cproc*, then a call to *Aproc* is data-refined by a call to *Cproc*. Note that this does not correspond to the standard definition of refinement in B where the refined operation should have exactly the same parameters as the abstract operation. In Section 6, we outline a way of representing this procedural refinement in standard B.

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 $\mathcal{D}_R^{a,c}[\![\ S_1\ ]\!] \sqsubseteq_R S_1'$ and $\mathcal{D}_R^{a,c}[\![\ S_2\ ]\!] \sqsubseteq_R S_2'$, and appealing to the data refinement laws of Fig. 1, then:

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

So,

$$S_1; S_2 \quad \sqsubseteq_R \quad S_1'; S_2'.$$

As well as the data refinement rules of Fig. 1, we use standard algorithmic refinement rules from [1, 10] throughout the case studies in this paper.

3 System of Interest

Both case studies used in this paper are taken from a system currently under development. The overall system is a system for the counting of votes for 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). An implementation of a Proportional Representational system, known as Single Transferable Vote, is used [6].

The input to this system is a collection of votes, which are then counted according to a set of rules. (These rules are not of interest to us for the scope of this paper). 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 at specification stage 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.

At the abstract level, the input is modelled as a bag of papers where a paper is a partial function from candidate to $\mathbb{N}$ and $bag\ T == T \rightarrow\!\!\!\!\!\!+ \mathbb{N}_1$, 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 first case study will deal with this operation, whose abstract specification is called $Make\_Ballot$. The preprocessing of the bag of papers returns a bag of ballots. Not all votes will be valid, so some input will be discarded. The second case study will deal with this overall operation, whose abstract specification is called $Pre\_Process$.

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).

$$\mathcal{D}_R^{a,c}[\![S_1 ; S_2]\!] \sqsubseteq \mathcal{D}_R^{a,c}[\![S_1]\!] ; \mathcal{D}_R^{a,c}[\![S_2]\!] \tag{1.1}$$

$$\frac{p \wedge R \;\Rightarrow\; G \Longleftrightarrow H}{\begin{array}{c}\mathcal{D}_R^{a,c}[\![\textbf{PRE } p \textbf{ THEN WHILE } G \textbf{ DO } S \textbf{ END}]\!] \\ \sqsubseteq \;\textbf{WHILE } H \textbf{ DO } \mathcal{D}_R^{a,c}[\![S]\!] \textbf{ END}\end{array}} \tag{1.2}$$

$$\frac{p \wedge R \;\Rightarrow\; G \Longleftrightarrow H}{\begin{array}{c}\mathcal{D}_R^{a,c}[\![\textbf{PRE } p \textbf{ THEN IF } G \textbf{ THEN } S \textbf{ END}]\!] \\ \sqsubseteq \;\textbf{IF } H \textbf{ THEN } \mathcal{D}_R^{a,c}[\![S]\!] \textbf{ END}\end{array}} \tag{1.3}$$

$$\frac{R \wedge Q \wedge p \;\Rightarrow\; (\exists a' \bullet [a, c := a', c'] R \wedge P)}{\mathcal{D}_R^{a,c}[\![\textbf{PRE } p \textbf{ THEN } @a' \bullet (P \implies a := a')]\!] \sqsubseteq @c' \bullet (Q \implies c := c')} \tag{1.4}$$

$$\frac{\begin{array}{c} R \Rightarrow (\exists a_1, c_1 \bullet Q) \\ (\forall a_1, c_1 \bullet R) \iff R \\ (@a_1 \bullet S) \; = \; S \end{array}}{\mathcal{D}_R^{a_0,c_0}[\![\textbf{VAR } a_1 \textbf{ IN } S \textbf{ END}]\!] \sqsubseteq \textbf{VAR } c_1 \textbf{ IN } \mathcal{D}_{R \wedge Q}^{(a_0,a_1),(c_0,c_1)}[\![S]\!] \textbf{ END}} \tag{1.5}$$

$$\frac{\begin{array}{c} a_2 \longleftarrow Aproc(a_1) \mathrel{\widehat{=}} A \\ c_2 \longleftarrow Cproc(c_1) \mathrel{\widehat{=}} C \\ \mathcal{D}_R^{(a_1,a_2),(c_1,c_2)}[\![A]\!] \sqsubseteq C \end{array}}{\mathcal{D}_{R'}^{(a_1',a_2'),(c_1',c_2')}[\![a_2' \longleftarrow Aproc(a_1')]\!] \sqsubseteq c_2' \longleftarrow Cproc(c_1')} \tag{1.6}$$

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

Figure 1: Data Refinement Laws

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.

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.

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:

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

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 the validated ballot. The specification of the preprocessing operation is shown in Fig. 2. We present two case studies based on this operation.

CONSTANTS $make\_ballot$, $pre\_process$
PROPERTIES
$make\_ballot \in Paper \rightarrow Ballot \ \wedge$
$pre\_process \in bag\ Paper \rightarrow bag\ Ballot \ \wedge$
$\forall\ paper.(paper \in Paper \Rightarrow$
$\quad make\_ballot(paper) =$
$\quad\quad 1..(min(\{nn \mid nn \in 1..no\_cands + 1 \ \wedge card(paper^{\sim}[\{nn\}]) \neq 1\}) \ - 1)$
$\quad\quad\quad \lhd\ paper^{\sim}$
$\wedge$
$\forall\ bagpapers.(bagpapers \in bag\ Paper \Rightarrow$
$\quad pre\_process(bagpapers) \ =$
$\quad\quad \{bb, nn \mid bb \in Ballot \wedge nn \in \mathbb{N} \wedge$
$\quad\quad\quad \exists\ pp.(pp \in dom(bagpapers) \wedge bb = make\_ballot(pp)) \wedge$
$\quad\quad\quad\quad nn = \Sigma pp \bullet (bb = make\_ballot(pp) \mid bagpapers(pp))$
$\quad\quad\quad \wedge card(bb) > 0\}$
OPERATIONS
$bagballots \longleftarrow Pre\_Process(bagpapers) \ \widehat{=}$
PRE
$\quad bagpapers \in bag\ Paper$
THEN
$\quad bagballots := pre\_process(bagpapers)$
END

Figure 2: Abstract Specification of Preprocessing

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 ≥ 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 second case study illustrates the development of the $Pre\_Process$ operation which takes a collection of ballots, each returned by the $Make\_Ballot$ operation, and inserts them into an output bag under certain conditions.

4 Case Study 1 - Development of Make_Ballot

The abstract specification of $Make\_Ballot$ is as follows:

$aballot \longleftarrow Make\_Ballot(apaper) \widehat{=}$
PRE
$\quad apaper \in Paper$
THEN
$\quad aballot := make\_ballot(apaper)$
END.

$$
\begin{array}{l}
\textbf{VAR } so\_far \textbf{ IN} \\
\quad so\_far := 1; \\
\quad \textbf{WHILE } so\_far < first\_skip\_or\_dup \textbf{ DO} \\
\quad\quad aballot(so\_far) := apaper^{\sim}(so\_far); \\
\quad\quad so\_far := so\_far + 1; \\
\quad \textbf{INVARIANT} \quad LI_1 \\
\quad \textbf{VARIANT} \quad first\_skip\_or\_dup - so\_far \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figure 3: First Loop Introduction on *Make_Ballot*

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

$$
\begin{array}{l}
aballot := (\ 1..min(\{nn \mid nn \in 1..no\_cands + 1 \wedge card(apaper^{\sim}[\{nn\}]) \neq 1\}) - 1\ ) \\
\qquad\qquad \lhd\ apaper^{\sim}.
\end{array}
$$

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

$$
aballot := (\ 1..first\_skip\_or\_dup - 1\ ) \quad \lhd\ apaper^{\sim}.
$$

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 outline the shape of a number of different possible implementations. We compare these for efficiency, using [2] and decide on a best possible implementation. (This decision making-process is of no further interest to us for the purpose of this paper). We then have a possible implementation to aim for which helps us to make decisions during development. Using this approach means that for the case studies in this paper, we start with the same specification and expect to arrive at a similar implementation using both styles of development.

4.1 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:

$$
LI_1 \ \widehat{=} \qquad aballot = \{\ nn \mapsto cc \mid nn \in 1..so\_far - 1 \wedge cc \mapsto nn \in apaper\ \}.
$$

The guard of the loop is $so\_far < first\_skip\_or\_dup$. We introduce the loop shown in Fig. 3. Note that $apaper^{\sim}(so\_far)$ is well defined since:

$$
\begin{array}{l}
apaper \in Candidate \nrightarrow \mathbb{N} \wedge so\_far < first\_skip\_or\_dup \\
\Rightarrow\ \forall ii.(ii \in 1.so\_far \Rightarrow card(apaper^{\sim}[\{ii\}]) = 1 \\
\Rightarrow\ 1..so\_far \lhd apaper^{\sim} \in \mathbb{N} \nrightarrow Candidate.
\end{array}
$$

$$
\begin{array}{l}
\textbf{VAR } so\_far, bb \textbf{ IN} \\
\quad so\_far, bb := 1, \emptyset; \\
\quad bb := BB \ ; \\
\quad \textbf{PRE } \ bb = BB \ \textbf{ THEN} \\
\quad\quad \textbf{WHILE } so\_far < first\_skip\_or\_dup \textbf{ DO} \\
\quad\quad\quad aballot(so\_far) := apaper^{\sim}(so\_far); \\
\quad\quad\quad so\_far := so\_far + 1; \\
\quad\quad \textbf{END} \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figure 4: $Make\_Ballot$ after the introduction of intermediate variable bb.

Most of the proof obligations generated from the introduction of this loop are easily discharged. We have a close look at the P-Rule [13], i.e., $LI_1 \wedge G \Rightarrow [Body]LI_1$.

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

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:

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

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. 4.

We need functions to return component parts of pairs, for elements of bb. $f_3(bb)(ii)$ is the candidate at position ii and $f_4(bb)(ii)$ is the number of candidates who have been given this preference. The definitions of these functions are obvious. We introduce (an overloaded) dot notation as syntactic sugar for these functions, so *bb(ii).cand* returns the candidate at position ii and *bb(ii).no* returns the number of candidates at preference ii. We use this syntactic sugar from now on.

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

$$GI_1 \quad \widehat{=} \quad aballot = cballot.$$

$$
\begin{array}{l}
\textbf{VAR } so\_far, bb \textbf{ IN} \\
\quad so\_far, bb := 1, \emptyset; \\
\quad bb := BB\ ; \\
\quad \textbf{WHILE } bb(so\_far).no = 1 \textbf{ DO} \\
\quad\quad cballot(so\_far) := bb(so\_far).cand; \\
\quad\quad so\_far := so\_far + 1; \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figure 5: $Make\_Ballot$ after first data refinement

$$
\begin{array}{l}
\textbf{VAR } processed \textbf{ IN} \\
\quad bb, processed := 1..no\_cands \times \{(\perp, 0)\}, \emptyset; \\
\quad \textbf{WHILE } dom(apaper) - processed \neq \emptyset \textbf{ DO} \\
\quad\quad \textbf{VAR } currcand, index \textbf{ IN} \\
\quad\quad\quad currcand :\in dom(apaper) - processed\ ; \\
\quad\quad\quad index := apaper(currcand)\ ; \\
\quad\quad\quad bb(index).cand := currcand\ ; \\
\quad\quad\quad bb(index).no := bb(index).no + 1\ ; \\
\quad\quad\quad processed := processed \cup \{currcand\} \\
\quad\quad \textbf{END} \\
\quad \textbf{INVARIANT} \quad LI_2 \\
\quad \textbf{VARIANT} \quad card(dom(apaper) - processed) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figure 6: Calculating intermediate variable bb.

We proceed as follows: The outermost **VAR** statement is refined using Law 1.5 from Fig. 1. This does not result in any change in the introduced variables, but distributes the data refinement inside the statement. We accordingly apply Law 1.1 from Fig. 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 Law 1.2 from Fig. 1. We can show that

$$
\begin{array}{lcl}
bb = BB & \Rightarrow & so\_far < first\_skip\_or\_dup \;=\; bb(so\_far).no = 1 \;\wedge \\
bb = BB & \Rightarrow & apaper^{\sim}(so\_far) \;=\; bb(so\_far).cand.
\end{array}
$$

The data refined version of Fig. 4 is shown in Fig. 5.

Next, we refine the calculation of the local variable bb using the loop shown in Fig. 6. The loop invariant is:

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

Again, the main proof is easily discharged.

We now apply data refinement to the code in Fig. 6. We replace an abstract paper, represented as a partial function, by a sequence of $Candidate, \mathbb{N}$ pairs.

```
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 7: Data-refined loop for calculating bb

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

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:

$$\begin{aligned} GI_3 \;\widehat{=}\; & processed = \{c \mid \exists ii \bullet (ii \in 1..so\_far - 1 \wedge cpaper(ii).cand = c)\} \wedge \\ & so\_far \in 1..no\_cands + 1. \end{aligned}$$

We can show that

$$\begin{aligned} & GI_2 \wedge GI_3 \;\Rightarrow\; dom(apaper) - processed \neq \emptyset \;=\; so\_far - 1 \leq card(cpaper) \\ & \mathcal{D}[\![\; currcand :\in dom(apaper) - processed \;]\!] \\ & \qquad \sqsubseteq\; currcand := cpaper(so\_far).cand \\ & \mathcal{D}[\![\; index := apaper(currcand) \;]\!] \;\sqsubseteq\; index := cpaper(so\_far).pref \\ & \mathcal{D}[\![\; processed := processed \cup \{currcand\} \;]\!] \;\sqsubseteq\; so\_far := so\_far + 1. \end{aligned}$$

Using the data refinement rules, the loop of Fig. 6 (excluding the initialisation of bb) is data-refined by the loop of Fig. 7. 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.

4.2 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 4. 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 $c\_first\_skip\_or\_dup$.

$$
\begin{array}{l}
c\_first\_skip\_or\_dup = \\
\qquad min(\{\ nn \mid nn \in 1..no\_cands + 1 \wedge card(ran(cpaper)^{\sim}[\{nn\}]) \neq 1\ \}.
\end{array}
$$

$$
\begin{array}{l}
cballot \longleftarrow C\_Make\_Ballot(cpaper) \widehat{=} \\
\textbf{PRE} \\
\quad cpaper \in iseq(Candidate \times \mathbb{N}) \\
\textbf{THEN} \\
\quad cballot := \{nn \mapsto cc \mid nn \in 1..c\_first\_skip\_or\_dup - 1) \wedge \\
\qquad\qquad cc \mapsto nn \in ran(cpaper)\} \\
\textbf{END}.
\end{array}
$$

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

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

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

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

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.

$$
\begin{array}{l}
LI_5\ \widehat{=}\ 1..so\_far \lhd cb = 1..so\_far - 1 \times \{(\bot, 0)\} \wedge \\
\quad so\_far \in 1.no\_cands + 1.
\end{array}
$$

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

$$
\begin{array}{l}
LI_6\ \widehat{=}\ cballot = \{nn \mapsto cc \mid nn \in 1..so\_far - 1 \wedge cc \mapsto nn \in ran(cpaper)\} \wedge \\
\qquad so\_far \in 1..no\_cands + 1.
\end{array}
$$

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. 6), the first loop had more non-determinism.

5 Case Study 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. 2. 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 4.

5.1 C++ STL Multisets

We implement the bags (of papers and ballots) using C++ Standard Template Library's multisets [4]. These 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 sequence of pairs, each pair being made up of a data item and the number of times this item occurs in the 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.

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

$$\textbf{MACHINE} \quad Multiset(Datatype, X\_Order),$$

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. 8) and $MultisetBallots$ (Fig. 9). Obviously, all multiset machines should have the same operations specified and implemented. The figures present only those operations used specifically for this case study.

5.2 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:

$$
\begin{array}{l}
LI_7 \;\widehat{=}\\
\quad bagballots =\\
\qquad \{\ bb, nn \mid bb \in bag\ Ballot \wedge nn \in \mathbb{N} \wedge\\
\qquad\qquad \exists pp \bullet (pp \in processed \wedge bb = make\_ballot(pp)\ \wedge\\
\qquad\qquad nn = \Sigma pp \bullet (pp \in processed \wedge bb = make\_ballot(pp) \mid bagpapers(pp)) \wedge\\
\qquad\qquad\qquad card(bb) > 0\ \}\\
\qquad \wedge\\
\quad processed \subseteq dom(bagpapers).
\end{array}
$$

Looking at the post-condition of $Pre\_Process$, we immediately introduce a loop as shown in Fig. 10. It can be shown that all the proof obligations can be discharged.

MACHINE $MulitsetPapers$
CONSTANTS $Ordered$
PROPERTIES
  $PaperOrder \in \mathbb{P}(iseq(Paper \times \mathbb{N}))$ /* Set of all Ordered sequences */
VARIABLES
  $msetpapers$, /* The sequence containing the pairs (Paper, Count) */
  $iter$ /* The multiset *iterator*, which indicates the current
    position in the multset. */
INVARIANT
  $msetpapers \in iseq(Paper \times \mathbb{N}) \wedge$
  $iter \in 1..card(msetpapers) + 1 \wedge$
  $msetpapers \in PaperOrder$
OPERATIONS
  $Start \mathrel{\widehat{=}} iter := 1;$

  $pair \longleftarrow GetNext \mathrel{\widehat{=}}$
  PRE
    $iterNotAtEnd$
  THEN
    $pair := msetpapers(iter) \parallel iter := iter + 1;$

DEFINITIONS
  $iterNotAtEnd \mathrel{\widehat{=}} (iter \leq card(msetpapers))$
END

Figure 8: Mulitset of papers machine

$$
\begin{array}{l}
\textbf{MACHINE} \quad \mathit{MultisetBallot} \\
\textbf{CONSTANTS} \quad \mathit{Ordered} \\
\textbf{PROPERTIES} \\
\quad \mathit{BallotOrder} \in \mathbb{P}(\mathit{iseq}(\mathit{Ballot} \times \mathbb{N})) \\
\textbf{VARIABLES} \quad \mathit{msetballots} \\
\textbf{INVARIANTS} \\
\quad \mathit{msetballots} \in \mathit{iseq}(\mathit{Ballot} \times \mathbb{N}) \wedge \\
\quad \mathit{msetballots} \in \mathit{BallotOrder} \\
\textbf{INITIALISATION} \\
\quad \mathit{msetballots} := \emptyset \\
\textbf{OPERATIONS} \\
\quad \mathit{MultiInsert}(\mathit{item}, \mathit{number}) \mathrel{\widehat{=}} \\
\quad \textbf{PRE} \\
\quad\quad \mathit{item} \in \mathit{Ballot} \ \wedge \ \mathit{number} \in \mathbb{N} \\
\quad \textbf{THEN} \\
\quad\quad \textbf{IF} \ \mathit{item} \in \mathit{dom}(\mathit{ran}(\mathit{msetballots})) \ \textbf{THEN} \\
\quad\quad\quad \textbf{ANY} \ \mathit{pos} \ \textbf{WHERE} \\
\quad\quad\quad\quad \mathit{fst}(\mathit{msetballots}(\mathit{pos})) = \mathit{item} \\
\quad\quad\quad \textbf{THEN} \\
\quad\quad\quad\quad \mathit{msetballots}(\mathit{pos}) := \\
\quad\quad\quad\quad\quad \mathit{fst}(\mathit{msetballots}(\mathit{pos})) \mapsto \mathit{snd}(\mathit{msetballots}(\mathit{pos})) + \mathit{number} \\
\quad\quad\quad \textbf{END} \\
\quad\quad \textbf{ELSE} \\
\quad\quad\quad \textbf{ANY} \ \mathit{msetballots}', \mathit{aa}, \mathit{bb} \ \textbf{WHERE} \\
\quad\quad\quad\quad \mathit{msetballots} = \mathit{aa} \frown \mathit{bb} \wedge \\
\quad\quad\quad\quad \mathit{msetballots}' = \mathit{aa} \frown [\mathit{item} \mapsto \mathit{number}] \frown \mathit{bb} \wedge \\
\quad\quad\quad\quad \mathit{msetballots}' \in \mathit{BallotOrder} \\
\quad\quad\quad \textbf{THEN} \\
\quad\quad\quad\quad \mathit{msetballots} := \mathit{msetballots}' \\
\quad\quad\quad \textbf{END} \\
\quad\quad \textbf{END} \\
\quad \textbf{END}; \\
\\
\quad \mathit{MakeEmpty} \mathrel{\widehat{=}} \mathit{msetballots} := \emptyset; \\
\textbf{END}
\end{array}
$$

Figure 9: Mulitset of ballots machine

$bagballots \longleftarrow Pre\_Process(bagpapers) \widehat{=}$
PRE $bagpapers \in bag\ Paper$
THEN
  VAR $processed$ **IN**
    $bagballots, processed := \emptyset, \emptyset;$
    WHILE $(dom(bagpapers) - processed) \neq \emptyset$ **DO**
      VAR pp, bb **IN**
        $pp :\in (dom(bagpapers) - processed)$;
        $bb \longleftarrow Make\_Ballot(pp);$
        IF $card(bb) > 0$
        THEN
          $bagballots :=$
            $bagballots \lessdot\!\!+ \{bb \mapsto bagballots(bb) + bagpapers(pp)\}$
        END ;
        $processed := processed \cup \{pp\}$
      END
    INVARIANT LI_7
    VARIANT $card(dom(bagpapers) - processed)$
    END
  END
END

Figure 10: Algorithmically-refined preprocessing

We now apply data refinement techniques as described in Section 2. We use the transformations and discharge the proof obligations. The gluing invariant between $bagpapers$, $bagballots$ and $msetpapers$, $msetballots$ is

$$\begin{aligned} GI_4 \ \widehat{=} \ \ bagpaper \ &= \ \{ap, nn \mid \exists\, cp \bullet (cp \mapsto nn \in ran(msetpapers) \ \wedge \\ &\qquad\qquad ap = ran(cp))\ \} \ \ \wedge \\ bagballots \ &= \ ran(msetballots). \end{aligned}$$

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

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

we can show that

$$\begin{aligned} &GI_4 \wedge GI_5 \ \ \Rightarrow \\ &\quad dom(Papers) - processed \neq \emptyset \ = \ ran(msetpapers) - mprocessed \neq \emptyset. \end{aligned}$$

Within the loop body we use:

$$GI_6 \ \widehat{=} \ pp = ran(fst\ mp) \ \wedge \ snd(mp) = bagpapers(pp) \ \wedge \ bb = cb.$$

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

$$\begin{aligned} &\mathcal{D}[\![\ bagballots := \emptyset\ ]\!] \ \sqsubseteq \ msetballots := \emptyset \\ &\mathcal{D}[\![\ pp :\in dom(bagpapers) - processed\ ]\!] \ \sqsubseteq \\ &\qquad mp :\in ran(msetpapers) - mprocessed \\ &\mathcal{D}[\![\ bb \longleftarrow Make\_Ballot(pp)\ ]\!] \ \sqsubseteq \ cb \longleftarrow C\_Make\_Ballot(fst\ mp) \\ &\mathcal{D}[\![\ bagballots := bagballots \lessdot\!\!+ \{bb \mapsto bagballots(bb) + bagpapers(pp)\}\ ]\!] \ \sqsubseteq \\ &\qquad MultiInsert(cb, snd\ mp) \\ &\mathcal{D}[\![\ processed \cup \{pp\}\ ]\!] \ \sqsubseteq \ mprocessed \cup \{mp\}. \end{aligned}$$

This leads to the following program:

$$
\begin{array}{l}
msetballots \longleftarrow C\_Pre\_Process(msetpapers) \widehat{=} \\
\textbf{PRE } msetpapers \in multisetPapers \textbf{ THEN} \\
\quad \textbf{VAR } mprocessed \textbf{ IN} \\
\quad\quad msetballots, mprocessed := \emptyset, \emptyset; \\
\quad\quad \textbf{WHILE } ran(msetpapers) - mprocessed \neq \emptyset \textbf{ DO} \\
\quad\quad\quad \textbf{VAR } mp, cb \textbf{ IN} \\
\quad\quad\quad\quad mp :\in ran(msetpapers) - mprocessed \ ; \\
\quad\quad\quad\quad cb \longleftarrow C\_Make\_Ballot(fst\ mp) \ ; \\
\quad\quad\quad\quad \textbf{IF } card(cb) > 0 \textbf{ THEN } MultiInsert(cb, snd(mp)) \quad \textbf{END} \ ; \\
\quad\quad\quad\quad mprocessed := mprocessed \cup \{mp\}; \\
\quad\quad\quad \textbf{END} \\
\quad\quad \textbf{END} \\
\quad \textbf{END} \\
\textbf{END}.
\end{array}
$$

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

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

The refined program is defined in terms of the multiset machines (Fig. 8 and Fig. 9):

$$
\begin{array}{l}
MakeEmpty; \\
Start; \\
\textbf{WHILE } iterNotAtEnd \textbf{ DO} \\
\quad mp \longleftarrow GetNext; \\
\quad cb \longleftarrow C\_Make\_Ballot(fst\ mp); \\
\quad \textbf{IF } card(cb) > 0 \textbf{ THEN} \\
\quad\quad MultiInsert(cb, snd(mp)) \\
\quad \textbf{END} \\
\textbf{END}.
\end{array}
$$

This concludes the first path for $Pre\_Process$.

5.3 Pre_Process - Data Refinement before Algorithmic Refinement

In this section, we proceed with Data Refinement before introducing algorithmic refinements. We immediately apply data refinement and then proceed with algorithmic refinement. We use the function $make\_seq$ which takes a bag of ballots and returns an ordered sequence of pairs:

$$
\begin{array}{l}
make\_seq : bag\ Ballot \rightarrow seq\ (Ballot \times \mathbb{N}) \\
\forall b \bullet ran(make\_seq\ b) = b \\
\forall b \bullet make\_seq(b) \in BallotOrder.
\end{array}
$$

The data refined specification of the preprocessing operation is:

$$
\begin{array}{l}
msetballots \longleftarrow C\_Pre\_Process(msetpapers) \mathrel{\widehat{=}} \\
\mathbf{PRE} \\
\quad msetpapers \in seq(CPaper \times \mathbb{N}) \\
\mathbf{THEN} \\
\quad msetballots := make\_seq \; \{ \; cb \mapsto nn \mid \\
\qquad\qquad \exists ii \bullet (ii \leq card(msetpapers) \;\; \wedge \\
\qquad\qquad\quad cb = make\_ballot(ran(fst(msetpapers(ii)))) \;) \;\; \wedge \\
\qquad\qquad nn = \Sigma ii \bullet (ii \leq card(msetpapers) \;\; \wedge \\
\qquad\qquad\qquad cb = make\_ballot(ran(fst(msetpapers(ii)))) \mid \\
\qquad\qquad\qquad snd(msetpapers(ii)) \;) \;\; \wedge \\
\qquad\qquad card(cb) > 0) \; \} \\
\mathbf{END}.
\end{array}
$$

The following loop invariant is used to introduce the loop:

$$
\begin{array}{l}
LI_8 \mathrel{\widehat{=}} \\
\quad msetballots = make\_seq \; \{cb \mapsto nn \mid \\
\qquad\qquad \exists ii \bullet (ii \leq so\_far \;\; \wedge \\
\qquad\qquad\quad cb = make\_ballot(ran(fst(msetpapers(ii)))) \;) \;\; \wedge \\
\qquad\qquad nn = \Sigma ii \bullet (ii \leq so\_far \;\; \wedge \\
\qquad\qquad\qquad cb = make\_ballot(ran(fst(msetpapers(ii)))) \mid \\
\qquad\qquad\qquad snd(msetpapers(ii)) \;) \\
\qquad\qquad \wedge \; card(cb) > 0) \; \} \; \wedge \\
\quad so\_far \in 1..card(msetpapers).
\end{array}
$$

This can be compared to LI_7. Clearly it is more complex making the loop verification conditions more involved than the loop introduction at the abstract level. Note also that, in this case, the data refinement of the operation is performed in a single step which again leads to more complicated verification conditions.

6 Dealing with Procedural Refinement in the B-Toolkit

The B tools do not at present support the data refinement distribution as discussed in Section 2. However, the rule for refinement of procedures in Fig. 1 can be checked with the B tools by using the following technique. Assume we have

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

where a_1, c_1 are linked by I_1 and a_2, c_2 are linked by I_2. $Aproc$ 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 $Aproc$. As earlier discussed, B refinements are normally required to preserve interfaces. Instead, we construct machines M_1' and M_2' respectively with operations $Aproc'$ and $Cproc'$ as follows:

$$
\begin{array}{l}
c_2 \longleftarrow AProc'(a_1) \mathrel{\widehat{=}} \\
\quad \mathbf{VAR} \; a_2 \; \mathbf{IN} \; a_2 \longleftarrow Aproc(a_1) \; ; \; c_2 :\in \{c_2 \mid I_2\} \; \mathbf{END}
\end{array}
$$

$$
\begin{array}{l}
c_2 \longleftarrow CProc'(a_1) \mathrel{\widehat{=}} \\
\quad \mathbf{VAR} \; c_1 \; \mathbf{IN} \; c_1 :\in \{c_1 \mid I_1\} \; ; \; c_2 \longleftarrow Cproc(c_1) \; \mathbf{END}
\end{array}
$$

Now $AProc'$ and $Cproc'$ have the same interface and the tool can be used to check that $Aproc'$ is refined by $Cproc'$.

The justification for this is based on properties of data refinement described by von Wright [12]. 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 $c :\in \{c \mid I\}$. Hence showing that $Aproc'$ is refined by $Cproc'$ is equivalent to showing that $Aproc$ is data-refined by $Cproc$ under I.

7 Conclusions

We have described, using two case studies, how two styles of refinement work in B, namely that of applying algorithmic refinement before data refinement and vice versa. We have used laws on distribution of data refinement to implement the former approach based on [5, 8, 9, 11, 12].

In both case studies, we have been more detailed in the case of the former style, because it is not the usual style used. We have, as part of describing the structure of the approach for the latter style, compared the approaches via comparison of the loop invariants. We have not attempted to compare the two development paths in any more quantitative terms.

The main conclusion is 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.

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 4. We ensure that this possible implementation is efficient before choosing it using techniques in [2].

We used the idea of overloaded dot notation in Section 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.

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. 1) and allowing algorithmic refinement during the refinement stage. Presently, refinement must be interface-preserving. This is not always appropriate, as seen in both case studies. A workaround for non interface-preserving refinements in the B-Toolkit is presented in Section 6.

References

[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] U. Breyman. *Designing Components with the C++ STL: A New Approach to Programming.* Addison Wesley, 1998.

[5] M.J. Butler. Calculational Derivation of Pointer Algorithms from Tree Operations. *Science of Computer Programming,* 33 (1999) 221–260.

[6] Derbyshire & Derbyshire. *Political Systems of the World (2nd Edition).* Oxford Helicon, 1996. ISBN 185 986 1148.

[7] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science, Springer, 1989.

[8] C.A.R Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica,* 30:701–739, 1993.

[9] C.C. Morgan and T. Vickers, editors. *On the Refinement Calculus.* Formal Approaches to Computing and Information Technology. Springer, 1994.

[10] C.C. Morgan. *Programming from Specifications (2nd Edition).* Prentice–Hall, 1994.

[11] J.M. Morris. Laws of data refinement. *Acta Informatica,* 26:287–308, 1989.

[12] J. von Wright. The lattice of Data Refinement. *Acta Inform.,* 31 (2) (1994) 105–135.

[13] J.B. Wordsworth. *Software Engineering with B.* Addison-Wesley,1996.