Dunne, S. E. and Conroy, S. (2009) 'A practical single refinement method for B', First international conference, ABZ, London, UK, September 16-18, in Börger, E. et al. (eds) ***Abstract state machines, B and Z*****, Lecture notes in computer science. Heidelberg: Springer Berlin, pp.195-208.**

A Practical Single Refinement Method for B

Steve Dunne and Stacey Conroy

School of Computing, University of Teesside
Middlesbrough, TS1 3BA, UK
s.e.dunne@tees.ac.uk

Abstract. We propose a single refinement method for B, inspired directly by Gardiner and Morgan's longstanding single complete rule for data refinement, and rendered practical by application of the current first author's recent first-order characterisation of refinement between monotonic computations.

1 Introduction

In this paper we describe a method for verifying arbitrary refinements between B machines, in the absence of unbounded nondeterminism, in a single step rather than having to find an intermediate backward refinement of the "abstract" machine which is itself then forward-refined by the "concrete" machine. The idea of a single complete refinement rule is by no means new: such a rule for data refinement in a predicate-transformer setting was described as long ago as 1993 by Gardiner and Morgan [11], and it is indeed fundamentally their idea which we exploit in this paper. Gardiner and Morgan themsleves appear to have regarded their rule as of theoretical interest only; it seems they didn't seek to exploit it in practice. We will show that a slightly extended version of B provides a suitable setting for practical exploitation of Gardiner and Morgan's rule. Like Gardiner and Morgan we interpret a computation as a weakest-precondition (wp) predicate transformer from sets of final states to sets of starting states [6], and call it *monotonic* if its corresponding wp predicate-transformer is monotonic. A monotonic computation can exhibit both demonic and angelic nondeterminism. *Conjunctivity* and *disjunctivity* are special cases of monotonicity: a conjunctive computation can exhibit only demonic nondeterminism, while a disjunctive computation can exhibit only angelic nondeterminism [7].

Our contribution here is the formulation of a pair of simple first-order proof obligations for verifying refinements between monotonic computations, which renders such verifications amenable to mechanisation in a similar way to that which B already uses for refinements between conjunctive computations [1].

The remainder of the paper is structured as follows: in Section 2 we describe Gardiner and Morgan's single complete rule for data refinement and in Section 3 we take some mathematical insight from [2] to explain why an arbitrary data refinement can always be "factored" into a succession of backward and forward refinements. In Section 4 we summarise the relevant properties of extended substitutions which we subsequently exploit to develop our new single refinement method for B in Section 5; in Section 6 we illustrate the use of our new method on an example refinement scenario; in Section 7 we compare our single complete method for B with that formulated for Z in [4] before finally relating it to other relevant recent work and drawing some conclusions in Section 8.

2 Gardiner and Morgan's Rule for Data Refinement

Gardiner and Morgan [11] significantly advanced our understanding of data refinement when they showed that forward and backward refinement could be subsumed into a single complete refinement rule in which the traditional retrieve relation between abstract and concrete states is superseded by a monotonic predicate transformer of sets of abstract states to sets of concrete states. Such a predicate transformer can be regarded as characterising in terms of its wp semantics a heterogeneous monotonic computation from concrete states to abstract states called a *representation operation*. Our intuition is that in a particular refinement context such an operation "computes" for any given concrete state an abstract state which that concrete state can be said to "represent".

2.1 Cosimulation

For a pair of abstract data types Adt and Cdt with respective state spaces $Astate$ and $Cstate$, and respective initialisations ainit and cinit, finalisations afin and cfin, and repertoires of operations aop_i and cop_i for $i \in I$, then a monotonic representation operation rep from $Cstate$ to $Astate$ is a *cosimulation* if the following hold:

$$\begin{aligned} &\mathsf{ainit} \sqsubseteq \mathsf{cinit} \,;\, \mathsf{rep} \\ &\mathsf{rep} \,;\, \mathsf{aop}_i \sqsubseteq \mathsf{cop}_i \,;\, \mathsf{rep} \qquad \text{for each } i \in I \\ &\mathsf{rep} \,;\, \mathsf{afin} \sqsubseteq \mathsf{cfin} \end{aligned}$$

The significance of the existence of such a cosimulation is that it establishes that Cdt refines Adt. In the special case where rep is disjunctive then Cdt is a *forward refinement* of Adt, while if rep is conjunctive then Cdt is a *backward refinement* of Adt. There is, however, an important qualification on the completeness of Gardiner and Morgan's single rule, namely that the abstract operations and the representation operation itself must only be at most *boundedly* nondeterministic.

In prominent formal modelling methods such as B [1], Z [16] and VDM [12] finalisations are invariably just projections onto the global space, so the finalisation condition is trivially met providing that rep is total (*i.e.* everywhere feasible).

3 Factorising an Arbitrary Refinement

For any relation $R \in X \leftrightarrow Y$ Back and von Wright [2] define two particular computations from X to Y. They call these respectively the *demonic* and *angelic* *relational updates* on R, and denote them respectively by $[R]$ and $\{R\}$. The former is characterised by a conjunctive wp predicate transformer, the latter by a disjunctive one. If x and y range respectively over X and Y, R is expressed as predicate $R(x, y)$ and $Q(y)$ is any postcondition predicate, we have

$$\begin{aligned} \mathrm{wp}([R], Q) \quad &=_{df} \quad \forall\, y \,.\, R \Rightarrow Q \\ \mathrm{wp}(\{R\}, Q) \quad &=_{df} \quad \exists\, y \,.\, R \wedge Q \end{aligned}$$

In [2] it is shown that for any monotonic computation comp from X to Y an intermediate state space Z can be constructed with relations $R_1 \in X \leftrightarrow Z$ and $R_2 \in Z \leftrightarrow Y$ such that $\mathsf{comp} \;=\; \{R_1\}\,;[R_2]$. This explains why an arbitrary refinement of a data type *Adt* by another *Cdt* can always be factored into a backward refinement of *Adt* by some intermediate data type *Bdt* and then a forward refinement of that by *Cdt*. In these refinements the relations R_1 and R_2 play the familiar role of retrieve relations between the concrete and abstract states.

3.1 Traditional Representation of Refinements in B

Currently in both classical and Event-B refinement the retrieve relation concerned is of course subsumed along with the concrete machine's state invariant into what is known as the "gluing" invariant. The concrete machine is therefore not explicitly exhibited in the refinement component which is actually presented, although it is always inferrable from the latter. It is important to appreciate that this is a merely the way the original architects of the B method chose to represent refinements, rather than being fundamental to the concept of refinement itself in B. Other possibilities for representing refinements in B are quite imaginable. For example, in [3] a new RETRENCHMENT construct is proposed which refers to a pair of existing machines to express the existence of a *retrenchment* relation between them. In the same way B might have had a REFINEMENT construct which refers to a pair of existing machines and provides an appropriate retrieve relation between them.

4 Extended Substitutions

In [10] B's generalised substitution language is extended by the introduction of angelic choice, and a theory of so-called *extended* substitutions is developed. In particular, the bounded angelic and demonic choice operators are denoted respectively by "$\sqcup$" and "$\sqcap$". Like ordinary generalised substitutions [1, 8], extended substitutions can naturally express heterogeneous computations (those whose starting and final state spaces are distinct). This merely requires that their *passive* (read frame) variables are all associated with the starting state space, while their *active* (write frame) variables are all associated with the final state space[1]. The significance here is that extended substitutions provide a means of

[1] In the theory of generalised substitutions in [8] and of extended substitutions in [10] the active frame of a substitution is simply called its frame.

expressing a heterogeneous monotonic representation operation in B. We note that the read frame of an operation includes its input parameters, while its write frame includes its output parameters.

4.1 Relational characterisation of an extended substitution

Extended substitutions have several important associated characteristic predicates. For our purpose here the most significant of these is the so-called before-after *power co-predicate*[2] $\mathrm{cod}(S)$, defined for an extended substitution S with frame s as follows:

$$\mathrm{cod}(S) \quad =_{df} \quad [S]\, s \in u$$

Here the atomic variable u is assumed fresh, and ranges over sets of final states, where each such final state is denoted by a tuple whose components correspond to the individual variables of the final state in lexical order of their names, while the frame variable s is interpreted here as a similar tuple whose whose components collectively denote a starting state of the computation characterised by S. Thus $\mathrm{cod}(S)$ is a relational predicate whose free variables are those comprising s together with the fresh variable u.

For example, if S is $x, y := 7, x+1 \sqcap x := 8$ then since $\mathrm{frame}(S) = x, y$ the s in the definition of $\mathrm{cod}(S)$ above is interpreted here as the tuple (x, y), so we have that

$$\begin{aligned}
\mathrm{cod}(S) \quad &= \quad [x, y := 7, x+1 \sqcap x := 8]\,(x, y) \in u \\
&= \quad [x, y := 7, x+1]\,(x, y) \in u \ \wedge\ [x := 8]\,(x, y) \in u \\
&= \quad (7, x+1) \in u \ \wedge\ (8, y) \in u
\end{aligned}$$

Thus here $\mathrm{cod}(S)$ relates each starting state (x, y) to every corresponding set u of final states which includes states $(7, x+1)$ and $(8, y)$, and *inter alia*, therefore, to the minimal set of final states $\{(7, x+1), (8, y)\}$. Notice that the variable u in $\mathrm{cod}(S)$ is just a placeholder for sets of possible final states of the monotonic computation characterised by S, in the same way that the primed frame variables in the before-after predicate $\mathrm{prd}(T)$ of an ordinary generalised substitution T in [8] are collectively just a placeholder for individual possible final states of the conjunctive computation characterised by T.

4.2 Refinement of extended substitutions

An ordinary generalised substitution is characterised by its frame, its termination predicate trm and its before-after predicate prd [8], whereas in contrast an extended substitution is characterised by its frame and its cod alone without

[2] It is called a power co-predicate to distinguish it from its dual the *power predicate* $\mathrm{pod}(S) \ =_{df} \ \neg\, [S]\, s \notin u$ also defined in [10], which with the frame s provides an alternative full characterisation of S.

need of its trm. Indeed [10, Prop 5.6] establishes the following important first-order characterisation of refinement between extended substitutions S and T with the same frame, where v denotes the list of all free variables of $\mathrm{cod}(S)$ and $\mathrm{cod}(T)$ –including of course the special atomic variable u used in the definition of cod:

$$S \ \sqsubseteq \ T \quad \Leftrightarrow \quad \forall\, v \,.\ \mathrm{cod}(S) \Rightarrow \mathrm{cod}(T)$$

5 A Complete Single Refinement Rule for B

We exploit the above formulation of extended-substitution refinement to re-express Gardiner and Morgan's single complete refinement rule described in Section 2 by replacing its explicit occurrences of the refinement symbol $\sqsubseteq$. This yields the following complete first-order characterisation of the refinement of one B machine *Amach*, with initialisation ainit and operations aop_i for $i \in I$, by another *Cmach* with initialisation cinit and corresponding operations cop_i for $i \in I$. Such a refinement is verified if a representation operation rep can be specified from *Cmach*'s states to *Amach*'s states, expressed as a total boundedly nondeterministic extended substitution, such that

$$\forall\, v \ . \ \ \mathrm{cod}(\mathsf{ainit}) \ \Rightarrow \ \mathrm{cod}(\mathsf{cinit}\, ;\mathsf{rep})$$

$$\forall\, v \ . \ \ \mathrm{cod}(\mathsf{rep}\, ; \mathsf{aop}_i) \ \Rightarrow \ \mathrm{cod}(\mathsf{cop}_i \ ; \mathsf{rep}) \qquad\qquad \text{for each } i \in I$$

where v again signifies the list of all free variables of the cods concerned here.

5.1 Nature of a first-order characterisation

The above pair of obligations represent a first-order characterisation of refinement since they can be re-written to eliminate first all the references to cod by applying its definition and then the resulting substitutions by applying them as wp predicate transformers. This will result in a finite collection of proof obligations expressed only in first-order logic with set-membership and equality, and therefore eminently amenable to manual or machine-assisted proof.

The fact that an extended substitution is characterised by its frame and cod alone without need of trm conveniently serves to limit the number of proof obligations so generated. This is in contrast to traditional classical B refinement [1] which generates two proof obligations for each operation, one essentially concerned with before-after effects and one concerned with termination. In the following section we will illustrate our refinement method with an example.

6 Schrődinger's Cat Revisited

The trio of machines below is almost the same as the *Schrődinger's Cat* example given in [9] as one of several examples of "intuitively obvious" co-refinements which nevertheless can only be proved in one direction but not the other by B's traditional forward refinement method.

Our *ACat* and *BCat* machines each model from an external perspective the scenario of putting a cat into an opaque box, and then later taking it out and thereupon discovering whether it has survived or died during its confinement, its fate having been dealt nondeterministically.

6.1 The abstract and concrete specifications

First we introduce our *GivenSets* machine declaring relevant types:

MACHINE $\quad$ *GivenSets*

SETS

$\qquad BOXSTATE = \{empty, full\}$
$\qquad CATSTATE = \{alive, dead\}$

END

In the *Acat* machine below the cat's fate is actually sealed when it is placed in the box, because it is then that the state variable *cat* is nondeterministically assigned its relevant value *alive* or *dead* which will subsequently be reported when that cat is taken out of the box:

MACHINE $\quad$ *Acat*

SEES $\quad$ *GivenSets*

VARIABLES $\quad$ $acat, abox$

INVARIANT $\quad$ $abox \in BOXSTATE \;\wedge\; acat \in CATSTATE$

INITIALISATION $\quad$ $abox := empty \;||\; acat :\in CATSTATE$

OPERATIONS

$\mathsf{put} \;\widehat{=}\;$ PRE $abox = empty$
THEN $abox := full \;||\; acat :\in CATSTATE$
END ;

$rr \longleftarrow \mathsf{take} \;\widehat{=}$
PRE $abox = full$
THEN $abox, rr := empty, acat$
END

END

On the other hand, in the *BCat* machine below the cat's fate isn't sealed until it is taken out of the box, because only then is the report variable *rr* nondeterministically assigned its value *alive* or *dead*:

MACHINE $\quad$ *Bcat*

SEES $\quad$ *GivenSets*

VARIABLES $bbox$

INVARIANT $bbox \in BOXSTATE$

INITIALISATION $bbox := empty$

OPERATIONS

$$
\begin{array}{lll}
\mathsf{put} & \widehat{=} & \text{PRE} \;\; bbox = empty \\
& & \text{THEN} \;\; bbox := full \\
& & \text{END} \;\; ; \\[1ex]
rr \longleftarrow \mathsf{take} & \widehat{=} & \\
& & \text{PRE} \;\; bbox = full \\
& & \text{THEN} \;\; box := empty \;\; || \;\; rr :\in CATSTATE \\
& & \text{END}
\end{array}
$$

END

Clearly an external observer must remain entirely oblivious of this fine distinction between these machines' respective internal workings concerning just when the cat's fate is actually determined. From his perspective the machines behave identically. With a complete refinement method we ought to be able to prove both that $Acat \sqsubseteq Bcat$ and $Bcat \sqsubseteq Acat$. With B's standard refinement method we can only prove that $Bcat \sqsubseteq Acat$, but not that $Acat \sqsubseteq Bcat$. In [9] we developed a counterpart *backward* refinement, but even that doesn't allow us to prove directly here that $Acat \sqsubseteq Bcat$, since this isn't purely a backward refinement either[3].

6.2 Proof of Refinement

We will now prove directly that $Acat \sqsubseteq Bcat$ using our new single complete refinement method. For this we deem $Acat$ as the abstract datatype while $Bcat$ is the concrete one.

Representation operation First, we specify an appropriate representation operation:

$$
\begin{array}{lll}
\mathsf{rep} & \widehat{=} & \text{IF} \;\; bbox = empty \\
& & \text{THEN} \;\; abox, acat := empty, alive \;\sqcup\; abox, acat := empty, dead \\
& & \text{ELSE} \;\; abox, acat := full, alive \;\sqcap\; abox, acat := full, dead \\
& & \text{END}
\end{array}
$$

We note that rep employs both demonic choice "$\sqcap$" and angelic choice "$\sqcup$" so it is non-trivially monotonic.

[3] The original $Acat$ in [9] is subtly different from the one here: in addition to assigning values to $abox$ and rr its version of take also nondeterministically assigns either $alive$ or $dead$ to $acat$. This has no effect on the externally observable behaviour of the machine, but turns $Acat \sqsubseteq Bcat$ into a purely backward refinement which can be proved directly by [9]'s backward refinement method.

Initialisation Labelling the abstract ($Acat$) initialisation as ainit and the concrete ($Bcat$) one as binit, we have to prove that

$$\mathrm{cod}(\mathsf{ainit}) \Rightarrow \mathrm{cod}(\mathsf{binit}\,;\mathsf{rep})$$

Proof:

$$
\begin{aligned}
&\mathrm{cod}(\mathsf{ainit})\\
&\ = \{ \text{ defn of cod } \}\\
&[\mathsf{ainit}]\,(abox, acat) \in u\\
&\ = \{ \text{ body of } \mathsf{ainit} \ \}\\
&[abox := empty \ \ || \ \ acat :\in CATSTATE]\,(abox, acat) \in u\\
&\ = \{ \text{ rewrite } \ || \ \}\\
&[abox, acat := empty, alive \sqcap abox, acat := empty, dead]\,(abox, acat) \in u\\
&\ = \{ \text{ apply substitution } \}\\
&(empty, alive) \in u \ \wedge \ (empty, dead) \in u \qquad \textbf{(1)}
\end{aligned}
$$

whereas

$$
\begin{aligned}
&\mathrm{cod}(\mathsf{binit}\,;\mathsf{rep})\\
&\ = \{ \text{ defn of cod } \}\\
&[\mathsf{binit}\,;\mathsf{rep}]\,(abox, acat) \in u\\
&\ = \{ \text{ defn of } \ ; \ \}\\
&[\mathsf{binit}]\,[\mathsf{rep}]\,(abox, acat) \in u\\
&\ = \{ \text{ body of } \mathsf{rep} \ \}\\
&[\mathsf{binit}]\,[\text{IF ... END}]\,(abox, acat) \in u\\
&\ = \{ \text{ appln of IF ... END } \}\\
&[\mathsf{binit}]\,((bbox = empty \Rightarrow (empty, alive) \in u \ \vee \ (empty, dead) \in u)\\
&\qquad\qquad \wedge \ (bbox = full \Rightarrow (full, alive) \in u \ \wedge \ (full, dead) \in u))\\
&\ = \{ \text{ body of } \mathsf{binit} \ \}\\
&[bbox := empty]\,((bbox = empty \Rightarrow (empty, alive) \in u \ \vee \ (empty, dead) \in u)\\
&\qquad\qquad \wedge \ (bbox = full \Rightarrow (full, alive) \in u \ \wedge \ (full, dead) \in u))\\
&\ = \{ \text{ apply substitution, logic } \}\\
&(empty, alive) \in u \ \vee \ (empty, dead) \in u \qquad \textbf{(2)}
\end{aligned}
$$

whence it can be seen that **(1)** $\Rightarrow$ **(2)**

□

The put Operation To differentiate the abstract and concrete versions of put we label the former as aput and the latter as bput. We have to prove that

$$\mathrm{cod}(\mathsf{rep}\,;\mathsf{aput}) \Rightarrow \mathrm{cod}(\mathsf{bput}\,;\mathsf{rep})$$

Proof:

$$
\begin{aligned}
&\mathrm{cod}(\mathsf{rep}\,;\mathsf{aput})\\
&\ = \{ \text{ defn of cod } \}\\
&[\mathsf{rep}\,;\mathsf{aput}]\,(abox, acat) \in u\\
&\ = \{ \text{ defn of } ; \}\\
&[\mathsf{rep}]\,[\mathsf{aput}]\,(abox, acat) \in u\\
&\ = \{ \text{ body of } \mathsf{aput} \}\\
&[\mathsf{rep}]\,[abox = empty \mid (abox := full \ \ || \ \ acat :\in CATSTATE)]\,(abox, acat) \in u\\
&\ = \{ \text{ defn of } \mid \}\\
&[\mathsf{rep}]\,(abox = empty \ \wedge \ [abox := full \ \ || \ \ acat :\in CATSTATE]\,(abox, acat) \in u\\
&\ = \{ \text{ rewrite } \ || \ \}\\
&[\mathsf{rep}]\,(abox = empty \ \wedge\\
&\qquad [abox, acat := full, alive \sqcap abox, acat := full, dead]\,(abox, acat) \in u\\
&\ = \{ \text{ apply substitution, logic } \}\\
&[\mathsf{rep}]\,(abox = empty \ \wedge \ (full, alive) \in u \ \wedge \ (full, dead) \in u)\\
&\ = \{ \text{ body of } \mathsf{rep} \}\\
&[\text{IF ... END}]\,(abox = empty \ \wedge \ (full, alive) \in u \ \wedge \ (full, dead) \in u)\\
&\ = \{ \text{ apply IF ... END, logic } \}\\
&bbox = empty \ \wedge \ (full, alive) \in u \ \wedge \ (full, dead) \in u \qquad \textbf{(3)}
\end{aligned}
$$

whereas

$$
\begin{aligned}
&\mathrm{cod}(\mathsf{bput}\,;\mathsf{rep})\\
&\ = \{ \text{ defn of cod } \}\\
&[\mathsf{bput}\,;\mathsf{rep}]\,(abox, acat) \in u\\
&\ = \{ \text{ defn of } ; \}\\
&[\mathsf{bput}]\,[\mathsf{rep}]\,(abox, acat) \in u\\
&\ = \{ \text{ body of } \mathsf{rep} \}\\
&[\mathsf{bput}]\,[\text{IF ... END}]\,(abox, acat) \in u\\
&\ = \{ \text{ apply IF ... END } \}\\
&[\mathsf{bput}]\,(bbox = empty \Rightarrow (empty, alive) \in u \ \vee \ (empty, dead) \in u) \ \wedge\\
&\qquad\qquad (bbox = full \Rightarrow (full, alive) \in u \ \wedge \ (full, dead) \in u)\\
&\ = \{ \text{ body of } \mathsf{bput} \}\\
&[bbox = empty \mid bbox := full]\\
&\qquad\qquad (bbox = empty \Rightarrow (empty, alive) \in u \ \vee \ (empty, dead) \in u) \ \wedge\\
&\qquad\qquad (bbox = full \Rightarrow (full, alive) \in u \ \wedge \ (full, dead) \in u)\\
&\ = \{ \text{ apply substitution, logic } \}\\
&bbox = empty \ \wedge \ (full, alive) \in u \ \wedge \ (full, dead) \in u \qquad \textbf{(4)}
\end{aligned}
$$

whence it can be seen that **(3)** = **(4)**

□

The take **Operation** To differentiate the abstract and concrete versions of take we label the former as atake and the latter as btake. Since they share the output variable rr this appears in both their frames. We have to prove that

$$\mathrm{cod}(\mathsf{rep}\,;\mathsf{atake}) \Rightarrow \mathrm{cod}(\mathsf{btake}\,;\mathsf{rep})$$

We note that the relevant frame tuple u here is $(abox, acat, rr)$.

Proof:

$\mathrm{cod}(\mathsf{rep}\,;\mathsf{atake})$
$= \{$ defn of cod $\}$
$[\mathsf{rep}\,;\mathsf{atake}]\,(abox, acat, rr) \in u$
$= \{$ defn of ; $\}$
$[\mathsf{rep}]\,[\mathsf{atake}]\,(abox, acat, rr) \in u$
$= \{$ body of atake $\}$
$[\mathsf{rep}]\,[abox = full \mid abox, rr := empty, acat]\,(abox, acat, rr) \in u$
$= \{$ apply substitution $\}$
$[\mathsf{rep}]\,(abox = full \;\wedge\; (empty, acat, acat) \in u)$
$= \{$ body of rep $\}$
$[\text{IF ... END}]\,(abox = full \;\wedge\; (empty, acat, acat) \in u)$
$= \{$ apply IF ... END, logic $\}$
$$bbox = full \;\wedge\; (empty, alive, alive) \in u \;\wedge\; (empty, dead, dead) \in u \qquad \textbf{(5)}$$

whereas

$\mathrm{cod}(\mathsf{btake}\,;\mathsf{rep})$
$= \{$ defn of cod $\}$
$[\mathsf{btake}\,;\mathsf{rep}]\,(abox, acat, rr) \in u$
$= \{$ defn of ; $\}$
$[\mathsf{btake}]\,[\mathsf{rep}]\,(abox, acat, rr) \in u$
$= \{$ body of rep $\}$
$[\mathsf{btake}]\,[\text{IF ... END}]\,(abox, acat, rr) \in u$
$= \{$ apply IF ... END $\}$
$[\mathsf{btake}]\,((bbox = empty \Rightarrow (empty, alive, rr) \in u \;\vee\; (empty, dead, rr) \in u) \;\wedge$
$\quad (bbox = full \Rightarrow (full, alive, rr) \in u \;\wedge\; (full, dead, rr) \in u))$
$= \{$ body of btake $\}$
$[bbox = full \mid bbox := empty \;\;||\;\; rr :\in CATSTATE]$
$\quad ((bbox = empty \Rightarrow (empty, alive, rr) \in u \;\vee\; (empty, dead, rr) \in u) \;\wedge$
$\quad (bbox = full \Rightarrow (full, alive, rr) \in u \;\wedge\; (full, dead, rr) \in u))$
$= \{$ rewrite $||$ $\}$
$[bbox = full \mid bbox, rr := empty, alive \sqcap bbox, rr := empty, dead]$
$\quad ((bbox = empty \Rightarrow (empty, alive, rr) \in u \;\vee\; (empty, dead, rr) \in u) \;\wedge$
$\quad (bbox = full \Rightarrow (full, alive, rr) \in u \;\wedge\; (full, dead, rr) \in u))$
$= \{$ apply substitution, logic $\}$

$$bbox = full \ \wedge \ ((empty, alive, alive) \in u \ \vee \ (empty, dead, alive) \in u) \ \wedge \\ ((empty, alive, dead) \in u \ \vee \ (empty, dead, dead) \in u) \qquad \mathbf{(6)}$$

whence it can be seen that **(5)** $\Rightarrow$ **(6)**

□

7 Comparison with Single Complete Refinement in Z

In [4] Derrick gives a single complete refinement rule for Z, which he expresses within an appropriate relational framework although it is inspired by the older technique of *possibility mappings* first proposed in [15]. In place of a simple retrieve relation between abstract and concrete states, his rule employs a *powersimulation*, *i.e.* a relation from sets of abstract states to individual concrete states. There is in fact a close correspondence between Derrick's method and ours: specifically, his powersimulation when inverted should yield the power co-predicate of our cosimulation as embodied by our representation operation.

7.1 Derrick's Example Translated into B

The single complete rule in [4] is illustrated there on an example refinement which is neither a forward nor backward one, and therefore unamenable to a direct single-step proof using alone either the forward refinement rules or backward refinement rules in [16], although of course since these rules are jointly complete it would be possible to prove this as indeed any valid refinement by using them in combination via an intermediate refinement.

We applied our method to the same example, after first translating this from Z to B to obtain the following pair of machines:

MACHINE $\quad Amach$

VARIABLES $\quad xx$

INVARIANT $\quad xx \in 0..5$

INITIALISATION $\quad xx := 0$

OPERATIONS

$\mathsf{one} \ \widehat{=} \ $ PRE $\ xx = 0 \vee xx = 1$
THEN $\ xx = 0 \Longrightarrow xx := 1 \ \sqcap \ xx = 1 \Longrightarrow xx := 0$
END ;

$\mathsf{two} \ \widehat{=} \ $ PRE $\ xx = 0\ $ THEN $\ xx := 2 \ \sqcap \ xx := 3\ $ END ;

$\mathsf{three} \ \widehat{=} \ $ PRE $\ xx = 2 \vee xx = 3$
THEN $\ xx = 2 \Longrightarrow xx := 4 \ \sqcap \ xx = 3 \Longrightarrow xx := 5$
END

END

$$
\begin{array}{l}
\text{MACHINE} \quad Cmach \\
\text{VARIABLES} \quad yy \\
\text{INVARIANT} \quad yy \in \{0,2,4,5\} \\
\text{INITIALISATION} \quad yy := 0 \\
\text{OPERATIONS} \\
\quad \mathsf{one} \;\widehat{=}\; \text{PRE} \;\; cc = 0 \;\; \text{THEN} \;\; yy := 0 \;\; \text{END} \;\; ; \\
\quad \mathsf{two} \;\widehat{=}\; \text{PRE} \;\; yy = 0 \;\; \text{THEN} \;\; yy := 2 \;\; \text{END} \;\; ; \\
\quad \mathsf{three} \;\widehat{=}\; \text{PRE} \;\; yy = 2 \;\; \text{THEN} \;\; yy := 4 \;\sqcap\; yy := 5 \;\; \text{END} \\
\text{END}
\end{array}
$$

7.2 Verification of Derrick's Refinement Example

Our experience of verifying Derrick's refinement example was interesting. First, we constructed the following representation operation rpn corresponding directly to the powersimulation given by Derrick in [4] for the same example:

$$
\begin{array}{lll}
\mathsf{rpn} & \widehat{=} & (yy = 0 \mid (xx := 0 \sqcap xx := 1) \sqcup xx := 0 \sqcup xx := 1) \\
 & & \sqcup \;\; (yy = 4 \lor yy = 5 \mid (xx := 4 \sqcap xx := 5))
\end{array}
$$

We were then unexpectedly perplexed to find that this rpn was ineffective for proving $Amach \sqsubseteq Cmach$ by our method. On the other hand, we found *were* able to verify this refinement by means of a different representation operation rpr, where

$$
\begin{array}{lll}
\mathsf{rpr} & \widehat{=} & (yy = 0 \mid (xx := 0 \sqcup xx := 1)) \\
 & & \sqcup \;\; (yy = 2 \mid (xx := 2 \sqcap xx := 3)) \\
 & & \sqcup \;\; (yy = 4 \lor yy = 5 \mid (xx := 4 \sqcap xx := 5))
\end{array}
$$

We omit here the proofs involved, which are similar to those already given for Schrődinger's cat. We note that our representation operation rpr corresponds to the powersimulation r, defined in Z terms by

$$
\begin{array}{|l}
r : \mathbb{P}\, Astate \leftrightarrow Cstate \\
\hline
r = \{\{\langle xx \leadsto 0\rangle\} \mapsto \langle yy \leadsto 0\rangle, \\
\qquad \{\langle xx \leadsto 1\rangle\} \mapsto \langle yy \leadsto 0\rangle, \\
\qquad \{\langle xx \leadsto 2\rangle, \langle xx \leadsto 3\rangle\} \mapsto \langle yy \leadsto 2\rangle, \\
\qquad \{\langle xx \leadsto 4\rangle, \langle xx \leadsto 5\rangle\} \mapsto \langle yy \leadsto 4\rangle, \\
\qquad \{\langle xx \leadsto 4\rangle, \langle xx \leadsto 5\rangle\} \mapsto \langle yy \leadsto 5\rangle\}
\end{array}
$$

rather than the r defined in [4]. We subsequently alerted [4]'s author to this discrepancy between his and our powersimulations. He obliged us by undertaking his own investigation which resulted in his diagnosing a printer's error in [4]; moreover, he confirmed that the correct powersimulation for the example is indeed our r above rather than that given in [4]. We take this as a significant vindication of our single refinement method for B: not only has it proved effective in independently verifying this refinement example; it also directly led us to detect a previously unsuspected mistake in the original powersimulation given in [4] for verifying the same example by Derrick's rule.

8 Related Work and Conclusions

In [5] model-checking is employed to generate retrieve relations for both forward and backward refinements. Presumably this technique could be extended to generate powersimulations for arbitrary refinements, although this is not discussed in [5]. On the other hand [14] does describe automatic verification of arbitrary refinements in B using the ProB model checker [13]. That technique uses ProB to construct a relation from concrete states to sets of abstract states which is in effect the power co-predicate of a cosimulation for the refinement, so this complements our refinement proof method rather well.

Our single refinement method is applicable to classical B and Event-B alike. In particular, Event-B's characteristic introduction of new events during refinement raises no particular issues for the new method. The key to our method is the construction of an effective monotonic representation operation. Our experience indicates that the flexibility afforded by the extended substitution language's syntax to arbitrarily interleave demonic and angelic choices greatly assists the developer in such an exercise.

The example refinements on which we have demonstrated our single refinement method are necessarily rather trivial, although they do nevertheless illustrate all the principles of the method so we hope that they may have served sufficiently to demonstrate that our method is amenable to the sort of mechanisation provided by both the classical B and Event-B development support environments. Indeed we hope to explore the possible provision of a suitable plug-in for the Rodin platform for the generation of the proof obligations of our method. Two extensions to core B are needed by our method, namely support for extended substitutions and also for arbitrary tuples. Fortunately, we believe neither of these should pose any particular difficulty for support tool implementors.

Acknowledgements

We are grateful for the points raised by the anonymous reviewers, which we have endeavoured to address in this final version of the paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, 1998.
3. R. Banach and S. Fraser. Retrenchment and the B-Toolkit. In H. Treharne, S. King, M.C. Henson, and S. Schneider, editors, *ZB2005: Formal Specification and Development in Z and B*, number 3455 in Lecture Notes in Computer Science, pages 203–221. Springer, 2005.
4. J. Derrick. A single complete refinement rule for Z. *Journal of Logic and Computation*, 10(5):663–675, 2000.

5. J. Derrick and Smith G. Using model checking to automatically find retrieve relations. In *International Refinement Workshop (Refine 2007)*, number 201 in Electronic Notes in Theoretical Computer Science, pages 155–175. Elsevier, 2008.
6. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
7. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Berlin, 1990.
8. S.E. Dunne. A theory of generalised substitutions. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, number 2272 in Lecture Notes in Computer Science, pages 270–290. Springer-Verlag, 2002.
9. S.E. Dunne. Introducing backward refinement into B. In D. Bert, J.P. Bowen, S. King, and M. Walden, editors, *ZB2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users*, number 2651 in Lecture Notes in Computer Science, pages 178–196. Springer-Verlag, 2003.
10. S.E. Dunne. Chorus Angelorum. In Jacques Julliand and Olga Kouchnarenko, editors, *B2007: Formal Specification and Development in B*, number 4355 in Lecture Notes in Computer Science, pages 19–33. Springer, 2007.
11. P.H.B. Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5:367–382, 1993.
12. C.B. Jones. *Systematic Software Development Using VDM (2nd edn).* Prentice-Hall, 1990.
13. M. Leuschel and M.J. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, number 2805 in Lecture Notes in Computer Science, pages 855–874. Springer-Verlag, 2003.
14. M. Leuschel and M.J. Butler. Automatic refinement checking for B. In K. Lau and R Banach, editors, *Formal Methods and Software Engineering: ICFEM 2005*, number 3785 in Lecture Notes in Computer Science, pages 345–359. Springer-Verlag, 2005.
15. N.A. Lynch. Multivalued possibility mappings. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, number 430 in Lecture Notes in Computer Science, pages 519–543. Springer-Verlag, 1990.
16. J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.