The Safe Machine: A New Specification Construct for B

Steve Dunne

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

Abstract. We compare the role of state invariants in Z and other state-based formalisms with that of abstract machine invariants in B. We argue a case for bringing B into line with the other formalisms in its use of invariants, and show how this can be achieved by one small extension to B's underlying semantics concerning the multiple composition operator, which has in any case already been proposed by others from different motivations. We illustrate the utility of our proposal with a small specification example, our Electronic Thesaurus.

1 Introduction

The role of the state invariant in a Z specification of a sequential system is surely familiar to all; it has been amply described by many authors; the source most frequently cited is perhaps [8]. The Z specifier conventionally incorporates $\Delta State$ in his description of each of his system's operations, and $State$ (or $State'$) in his description of his system's initialisation. We can regard the state invariant as embodying all common required safe aspects of behaviour factored out from the individual operations, allowing those operations to be expressed all the more succinctly. Whatever else he asserts in their descriptions, as long as he adheres to this convention our Z specifier's operations and initialisation will necessarily respect the state invariant: his specifications are inherently "safe". In asserting some further specific effect of an operation the Z specifier may of course unwittingly contradict the state invariant, but the result of doing so is not thereby to compromise the operation's safety: rather, it is implicitly to precondition the operation so as to render it inapplicable wherever the contradiction obtains. We will call this use of the state invariant in specification a *constructive* one. It means the state invariant is an integral part of the specification. Other formalisms which employ state invariants in the same constructive manner within specification include VDM [6] and the Refinement Calculus [7].

The role of the machine invariant in B is somewhat different from the constructive one described above: it plays no direct part in the specifier's expression of his abstract machine's initialisation or operations, but rather serves as a safety template against which those operations and initialisation must be checked *a posteriori*. Weakening an abstract machine's invariant, even to the ultimate extent where it serves merely to type the machine's variables, does not change the

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 472–489, 1999.

meaning of the machine *per se*, though confidence in its validity as an expression of its user's requirements would be diminished if the consistency of its operations and initialisation has been checked only against this weakened invariant. The B specifier's only incentive to ensure his machine's invariant is as tight as possible is such a concern for validity. Indeed, one might even imagine –perish the thought– an unscrupulous specifier deliberately underspecifying his state invariant so as to avoid some tedious consistency proof obligations.

In this paper we set out to explore how the invariant of a B abstract machine might be conscripted to play a more constructive part in the specification of that machine's operations and initialisation. Having first identified an appropriate state invariant for his machine we would like the B specifier to be able to utilise it in specifying its initialisation and operations. This is in contrast to the current situation where, by obliging him to formulate his initialisation and operations without reference to his machine invariant but in a way which ensures they are consistent with that invariant, B might be said to be coercing the specifier into engaging in what is really a proto-implementation activity rather than one of pure specification. In any realm of engineering-description a consistency-checking need is usually symptomatic of some underlying duplication of description: the best way to avoid inconsistency is to describe everything just once. The proper place for consistency checking is *between* successive steps of design on the path to implementation, but not *within* any one design stage where it just betokens descriptive redundancy. We will look to Z for inspiration as to how such consistency checking might be eliminated in B: that is, we seek a means by which our B user can specify B abstract machines which are inherently safe. We will show that the key to this lies in a natural extention of the notion of multiple composition of generalised substitutions which already exists in B.

2 Some B Preliminaries

In the B literature generalised substitutions are usually understood in the context of an abstract machine with known state variables and a known invariant. A substitution therefore tends to be identified with the operation it characterises within that machine. We take a somewhat different standpoint here: our generalised substitutions will have an existence independent of any particular machine context; or, rather, we will regard a generalised substitution as inducing its own primitive context we call its *frame*. After explaining frames, we use them to describe a useful general notion of *refinement* between arbitrary pairs of substitutions. We also introduce the ideas of "totalising" a partly feasible substitution by formulating its *weakest feasible completion*, and of "normalising" a non-deterministic choice substitution, both of which will be needed later.

2.1 Substitutions and Frames

We call a collection of variables a *frame*. If a frame comprises only one variable, then the frame and its single constituent variable are synonymous. If u and v are frames, then

u, v

denotes the new frame obtained by merging u and v, and

$u \setminus v$

denotes the residual frame obtained by removing from u any variables it shares with v. In particular, note that

$u, v \;=\; v, u \qquad$ and $\qquad u, (v \setminus u) \;=\; u, v$

We call the collection of variables on which a generalised substitution acts its *active frame*. If S is a generalised substitution we denote its active frame by $\mathit{frame}(S)$. A substitution may make passive reference to variables outside its active frame. For example, the active frame of $x := y$ is just x although it makes passive reference to y too. The active frame may be empty as in *skip*, or as in $x < 7 \mid \mathit{skip}$ which makes only passive reference to x. We distinguish between *skip* and $x := x$ since they have different active frames. We also have that

$\mathit{frame}(S \parallel T) \;=\; \mathit{frame}(S), \mathit{frame}(T)$

When one abstract machine is included in another the operations of the former can be invoked as substitutions in the latter. The active frame of an invoked operation comprises *all* the variables of that operation's native machine, not just the active frame of its operation body therein. If *opbody* denotes that body and r denotes the residual native machine variables outside *opbody*'s active frame, the invoked operation is therefore equivalent to

$\mathit{opbody} \parallel r := r$

By incorporating the residual $r := r$ above we depart from the orthodox interpretation of operation invocation, which simply syntactically replaces the operation name by *opbody*. We contend our interpretation is preferable since it provides referential transparency over the way an operation is defined. For example, in a given machine we would recognise the two operations defined respectively with bodies *skip* and $x := x$ as having the same meaning, and therefore we would desire that their invocations in any including machine have the same meaning too.

2.2 Refinement of Substitutions

A generalised substitution S with active frame s is said to be *refined* by another generalised substitution T with active frame t, written

$S \;\sqsubseteq\; T$

if for any predicate Q without free variables in $t \setminus s$

$[S]Q \;\Rightarrow\; [T]Q$

Essentially T must be able to establish anything that S can establish. T's active frame will usually encompass that of S. One obvious exception is the rather pathological case of *magic* (i.e. $\mathit{false} \Longrightarrow \mathit{skip}$) which though having an empty active frame refines anything. If T has a wider active frame than S we can interpret this as T employing its own "local" variables outside the scope of S. The free-variables constraint we imposed on Q above reflects the local nature within T of these variables of $t \setminus s$.

2.3 Weakest Feasible Completion of a Substitution

In the Z literature a Z operation schema is always interpreted as a totally feasible operation, even when it has a restricted domain of before-states on which it can be applied. Indeed, the actual restriction predicate extracted from the schema property by hiding primed and !-decorated variables is called the *precondition* of the operation. The operational interpretation is that the operation is always invocable, although unsafe to invoke outside its precondition because there the effect will be unpredictable, perhaps even abortive. There is thus no concept of restricted feasibility for a Z operation, but only one of restricted safe applicability.

In hindsight we can see Z would probably have been better served by an interpretion of its schema preconditions as feasibility guards. Certainly, Z's current interpretation of schema preconditions makes a reasonable intuitive operational interpretation of other Z constructs unfortunately problematic: the schema disjunction of two operation schemas cannot in general be operationally interpreted as a non-deterministic choice between those two operations; nor can the schema sequential composition of two operation schemas always be operationally interpreted as a sequential composition of those operations. Interpreting schema preconditions as feasibility guards would resolve such anomalies at a stroke.

Happily, the syntax of generalised substitutions, in common with VDM and the Refinement Calculus, does distinguish between preconditions and guards. We can still impose a Z-like "total" interpretation of any substitution by explicitly preconditioning it by its own feasibility guard. We call this derivation the *weakest feasible completion* of the substitution. We define the weakest feasible completion of S as

$$\mathit{fis}(S) \mid S$$

where $\mathit{fis}(S)$ is the feasibility guard of S, defined as $\neg\ [S]\mathit{false}$. Interpreting our substitutions as weakest precondition predicate transformers, the above expresses a substitution which concurs with S where the latter is feasible but cannot be guaranteed to establish anything when applied where S is infeasible –unlike S itself which by definition can establish anything at all in those circumstances. If S is already total $\mathit{fis}(S)$ reduces to *true* and therefore, as one would expect, S is its own weakest feasible completion. The weakest feasible completion of *magic* is *abort*. Extracting the feasibility guard of a generalised substitution is a purely syntactic manipulation, so deriving the weakest feasible completion of a substitution is straightforward.

2.4 Normalising a Non-deterministic Choice

Consider the non-deterministic choice substitution $x := 3 \,[]\, y := 4$. Its two component substitutions have different frames x and y. We can rewrite such a choice as

$$x, y := 3, y \,[]\, x, y := x, 4$$

without changing its meaning, so that the two components now have identical active frames. This is what we term *normalising* a non-deterministic choice. More generally if S and T are two generalised substitutions, let u denote $\mathit{frame}(S) \setminus \mathit{frame}(T)$ and let v denote $\mathit{frame}(T) \setminus \mathit{frame}(S)$; then we can express $S[]T$ in normalised form as

$$(S \parallel v := v) \,[]\, (T \parallel u := u)$$

2.5 Classical Multiple Composition in B

In [1] Abrial defines the multiple composition $S \parallel T$ of generalised substitutions S and T only when S and T have disjoint active frames. He also forbids the multiple composition $op1 \parallel op2$ of two operations $op1$ and $op2$ from the same native machine, though the pathological example he exhibits on page 317 of [1] provides only a pragmatic illustration of the need for the prohibition, not a deep explanation for it. Our new interpretation of operation invocation allows us to infer the same prohibition as a particular consequence of his general edict on the disjointness of frames. This is because we now recognise that $op1$ and $op2$, by being from the same native machine, have the same active frame.

On page 308 of [1] Abrial gives these definitions of $\mathit{trm}(S \parallel T)$ and $\mathit{prd}(S \parallel T)$, where S and T act respectively on disjoint frames x and y:

$\mathit{trm}(S \parallel T)$	$\mathit{trm}(S) \wedge \mathit{trm}(T)$
$\mathit{prd}_{x,y}(S \parallel T)$	$\mathit{prd}_x(S) \wedge \mathit{prd}_y(T)$

The above *prd* definition is flawed. To see this we note that for any generalised substitution U, $\mathit{prd}(U) \vee \mathit{trm}(U)$ is a tautology.

Proof: Let U's active frame be u. We assert the following true disjunction:

$u = u' \vee \mathit{true}$

whence $\{a \vee b = \neg\, a \Rightarrow b\}$

$u \neq u' \Rightarrow \mathit{true}$

whence {monotonicity of $[U]$}

$[U]u \neq u' \Rightarrow [U]\mathit{true}$

whence {definition of $\mathit{trm}(U)$}

$[U]u \neq u' \Rightarrow trm(U)$
whence $\{a \Rightarrow b = \neg\, a \vee b\}$
$\neg\, [U]u \neq u' \vee trm(U)$
whence {definition of $prd(U)$}
$prd(U) \vee trm(U)$ QED

Abrial's above definitions of $trm(S \parallel T)$ and $prd(S \parallel T)$ do not uphold the tautology. For example, they give us

$$trm(abort \parallel magic) = false$$

$$prd(abort \parallel magic) = false$$

which cannot be the case for any generalised substitution. We can correct the flaw by amending the definition of $prd(S \parallel T)$ to

$prd_{x,y}(S \parallel T)$	$(trm(T) \Rightarrow prd_x(S)) \wedge (trm(S) \Rightarrow prd_y(T))$

We hasten to reassure B users this flaw in Abrial's definition of $prd(S \parallel T)$ is benign, since it in no way invalidates the rest of the treatment of multiple composition in [1]. Our correction therefore, though technically justified, is of little practical significance.

3 Generalising Multiple Composition

It is a fact that the characterisation of $S \parallel T$ in terms of *trm* and *prd* described in the previous section remains perfectly sound even if we dispense with Abrial's requirement that the active frames of S and T be disjoint. We are by no means the first to make this observation: Bert *et al* [4] characterise exactly such an operator, which they call their "new composition of substitutions" and denote by $S \otimes T$; in [5] Chartier describes how in his formalisation of the Generalised Substitution Language in Isabelle/HOL it seems more natural to define $\parallel$ without any explicit disjoint frames requirement. Both authors observe that their more general operators coincide exactly with Abrial's $\parallel$ whenever the frames of the two participating substitutions do happen to be disjoint. We propose to follow Chartier by retaining the symbol $\parallel$ to represent such a generalisation of Abrial's multiple composition, justifying this by representing that we haven't changed $\parallel$'s meaning, but merely extended its domain of application. We will indulge ourselves, however, with a change in terminology: we will refer to our generalised $\parallel$ as a *parallel composition* of substitutions.

3.1 Re-write Rules for ||

In section 7.1.2 of [1] Abrial gives a comprehensive set of re-write rules by which the multiple composition operator || can always be systematically eliminated from any generalised substitution expression. The existence of such a set of rules reveals that || is not, after all, a fundamental operator of the Generalised Substitution Language. In essence, it is really no more than a convenient syntactic shorthand for expressing substitutions which could always be alternatively expressed without it. Our extension of Abrial's multiple composition into our parallel composition doesn't change ||'s status as an operator which can always be eliminated by application of re-write rules. We do, though, have to consider carefully the impact of our extension on those re-write rules: it turns out we have to augment Abrial's existing battery of re-write rules for || with one more; we also have to qualify one of his existing rules, as we will show below.

A New Re-write Rule for ||. This new rule[1] captures the meaning of applying two simple substitutions in parallel on the same variable:

Syntax	Definition
$x := E \parallel x := F$	$E = F \Longrightarrow x := E$

It expresses that a parallel composition of two simple substitutions on the same variable is only feasible if those substitutions are assigning identical values.

Distributing || through Non-deterministic Choice. One of Abrial's existing re-write rules, expressed as

$$(S \,[]\, T) \;\parallel\; U \;=\; (S \parallel U) \,[]\, (T \parallel U)$$

allows us to distribute multiple composition through non-deterministic choice unconditionally. With our more general parallel composition we have to be a little more circumspect about such a distribution: it is in fact only valid once the non-deterministic choice concerned has been normalised. To see why this is so, consider the following parallel composition:

$$(x := 3 \,[]\, \mathit{skip}) \;\parallel\; x := 7$$

If we just unconditionally distribute the || through the [] we obtain

$$(x := 3 \parallel x := 7) \,[]\, (\mathit{skip} \parallel x := 7)$$

which reduces to $\mathit{magic} \,[]\, x := 7$ and thence $x := 7$. But if we directly calculate the *prd* of our parallel composition we have

[1] It was originally suggested by Andy Galloway.

$prd((x := 3 \ [] \ skip) \ || \ x := 7)$

$= \{$defn of $prd(S \ || \ T)\}$

$prd(x := 3 \ [] \ skip) \ \wedge \ prd(x := 7)$

$= \{$defn of $prd\}$

$\neg \, [x := 3 \ [] \ skip]x \neq x' \ \wedge \ \neg \, [x := 7]x \neq x'$

$= \{$apply substitutions and simplify$\}$

$(x' = 3 \vee x = x') \ \wedge \ x' = 7$

$= \{$boolean algebra$\}$

$(x' = 3 \wedge x' = 7) \ \vee \ (x = x' \wedge x' = 7)$

$=$

$false \ \vee \ x = x' = 7$

$=$

$x = x' = 7$

which is not $prd(x := 7)$ but rather $prd(x = 7 \Longrightarrow x := 7)$.

We try again, but this time we will normalise the non-deterministic choice first before we distribute the ||. Since *skip* has an empty frame, the normalised form of $x := 3 \ [] \ skip$ is just $x := 3 \ [] \ (skip \ || \ x := x)$ which reduces to $x := 3 \ [] \ x := x$. So we have

$(x := 3 \ [] \ skip) \ || \ x := 7$

$= \{$normalise the non-deterministic choice$\}$

$(x := 3 \ [] \ x := x) \ || \ x := 7$

$= \{$now distribute || through $[]\}$

$(x := 3 \ || \ x := 7) \ [] \ (x := x \ || \ x := 7)$

$= \{3 = 7$ is false, so the lefthand choice collapses to magic$\}$

$magic \ [] \ (x := x \ || \ x := 7)$

$= \{$magic disappears in non-deterministic choice$\}$

$x := x \ || \ x := 7$

$= \{$applying our new re-write rule$\}$

$x = 7 \Longrightarrow x := 7$

which accords completely with our *prd* calculation above.

3.2 Fusion

When both operands share the same active frame, or when one operand's active frame is completely contained within that of the other, our generalised || corresponds to what Back and Butler [2, 3] call, within the more general context of all monotonic predicate transformers [2], their *fusion* operator.

If we think of S and T as embodying two separate operational requirements on the same state-space, then their "fusion" is simply the logical conjunction of those requirements. Fusion therefore reduces demonic non-determinism. For example

$$x : (5 < x) \;||\; x : (x < 10)$$

is equivalent to

$$x : (5 < x < 10)$$

If S and T represent contradictory requirements then their fusion becomes infeasible. For example, as we saw earlier,

$$x := 3 \;||\; x := 7$$

collapses to *magic*. Our || is more general than either fusion or Abrial's multiple composition because, unlike either of them, its operands can have arbitrarily overlapping frames. For example

$$x, y := 1, 2 \;||\; y, z := z, y$$

is meaningful. It reduces, in fact, to

$$z = 2 \Longrightarrow x, y, z := 1, 2, y$$

3.3 Abrial's Pathological Example Revisited

We now consider the pathological example of a multiple composition given on page 317 of [1] to which we alluded earlier. It consists of a machine with frame x, y and invariant

$$x \in NAT \wedge y \in NAT \wedge x \leq y$$

with the following pair of operations

$$incx \mathrel{\widehat{=}} x < y \mid x := x + 1$$
$$decy \mathrel{\widehat{=}} x < y \mid y := y - 1$$

The parallel composition $inx \;||\; decy$ is now expressible, noting our introduction of a residual $r := r$ in each operation body, as

[2] Generalised substitutions model only the positively conjunctive subclass of such transformers.

$(x < y \mid x := x + 1 \mid\mid y := y) \mid\mid (x < y \mid y := y - 1 \mid\mid x := x)$

thence

$(x < y \mid x, y := x + 1, y) \mid\mid (x < y \mid x, y := x, y - 1)$

thence

$x < y \mid x = x + 1 \wedge y = y - 1 \Longrightarrow x, y := x, y$

thence

$x < y \mid \mathit{false} \Longrightarrow x, y := x, y$

thence

$x < y \mid \mathit{magic}$

Although entirely infeasible inside its precondition $x < y$, we conclude that *incx* || *decy* is certainly safe: that is to say, wherever it is applicable it (miraculously) preserves the machine's invariant. So the parallel composition of two invoked operations from the same machine can now safely be permitted. It is interpreted as the fusion of the two operations concerned. Often, as in this case, it will simply yield an infeasible substitution.

3.4 A Categorical Definition of Parallel Composition

Since the ultimate destiny of most generalised substitutions in B is refinement into implementation code, the following alternative characterisation of || directly in terms of refinement is apt:

> For generalised substitutions S and T with respective active frames x and y, $S \mid\mid T$ is the unique generalised substitution with active frame x, y satisfying the following universal property for any generalised substitution U:
>
> $$S \mid\mid T \sqsubseteq U \quad \Leftrightarrow \quad \mathit{trm}(T) \mid S \sqsubseteq U \;\wedge\; \mathit{trm}(S) \mid T \sqsubseteq U$$

That is to say, in the lattice of generalised substitutions ordered by refinement, $S \mid\mid T$ is the refinement supremum of the pair of substitutions

$$\mathit{trm}(T) \mid S \qquad \text{and} \qquad \mathit{trm}(S) \mid T$$

In other words, it is the "least-refined" co-refinement of these two substitutions. This alternative characterisation is important for builders of tools to support the B method since it indicates how we can directly "refine away" a parallel composition instead of having to eliminate it from a specification using the rewrite rules.

4 Intrinsically Safe Abstract Machines

Having developed our parallel composition operator for substitutions we can now employ it to formulate our desired new notion of a safe abstract machine. In order to do this we need first to coin some more B terminology –the "characteristic" operation and "characteristic" initialisation of an abstract machine– as explained in the next subsection.

4.1 Characteristic Operation and Initialisation

In conventional Z, $\Delta State$ represents the least-deterministic operation on *State* which maintains the invariant of *State*. Conventionally any other operation specified on *State* will incorporate $\Delta State$, and so can be deconstructed as a conjunction of operation $\Delta State$ with a second, usually unsafe operation. Now consider a B abstract machine with variable x and invariant *Inv*. We will use the names *Establish* for the generalised substitution

$$x : Inv$$

and *Preserve* for the generalised substitution

$$Inv \mid x : Inv$$

Establish expresses the least-deterministic safe initialisation of our machine. We call it our machine's *characteristic* initialisation. *Preserve* characterises an operation preconditioned by *Inv* and guaranteeing, providing the before-state is within *Inv*, to deliver an after-state also satisfying *Inv*. It is, in short, the least-deterministic safe operation for our machine –the exact B counterpart of Z's $\Delta State$ operation. We call it our machine's *characteristic* operation.

4.2 Making an Abstract Machine Safe

An arbitrary initialisation of our machine will be made safe by parallel composition with its characteristic initialisation *Establish*. Such a composite initialisation will inherit from *Establish* the certainty of establishing *Inv*. Similarly, an arbitrary operation body of our machine will be made safe by parallel composition with its characteristic operation *Preserve*.

We give a small example. Consider the following machine:

MACHINE
M

VARIABLES
x, y

INVARIANT
$x \in \mathbb{N} \wedge$
$y \in \mathbb{N} \wedge$
$x + y = 100$

INITIALISATION
$x := 0$
OPERATIONS
$incx \mathrel{\widehat{=}} x := x + 1$;
$incy \mathrel{\widehat{=}} y := y + 1$
END

Of course, M is manifestly unsafe since its initialisation and its operations all flout its invariant. *Establish* for M is the substitution

$$x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100)$$

and *Preserve* is the substitution

$$x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100 \mid x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100)$$

or equivalently in Abstract Machine Notation

PRE
$x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100$
THEN
$x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100)$
END

Now we will "repair" M by parallel-composing its initialisation with *Establish* and its operations with *Preserve*. We obtain the following machine:

MACHINE
SafeM
VARIABLES
x, y
INVARIANT
$x \in \mathbb{N} \wedge$
$y \in \mathbb{N} \wedge$
$x + y = 100$
INITIALISATION
$x := 0 \quad ||$
$x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100)$
OPERATIONS
safe_incx $\mathrel{\widehat{=}}$
$x := x + 1 \quad ||$
PRE
$x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100$
THEN
$x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100)$
END ;

$$
\begin{array}{l}
\quad safe\_incy \mathrel{\widehat{=}} \\
\qquad\qquad y := y + 1 \quad || \\
\qquad\qquad \mathrm{PRE} \\
\qquad\qquad\quad x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100 \\
\qquad\qquad \mathrm{THEN} \\
\qquad\qquad\quad x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100) \\
\qquad\qquad \mathrm{END} \\
\mathrm{END}
\end{array}
$$

Our "repaired" initialisation reduces to

$$x, y := 0, 100$$

The "repaired" operations *safe_incx* and *safe_incy* are infeasible, since they both contain non-trivial guards. For example, *safe_incx* reduces to

$$
\begin{array}{l}
\mathrm{PRE} \\
\quad x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100 \\
\mathrm{THEN} \\
\quad \mathrm{SELECT} \\
\qquad x < 100 \\
\quad \mathrm{THEN} \\
\qquad x, y := x + 1, y - 1 \\
\quad \mathrm{END} \\
\mathrm{END}
\end{array}
$$

Our goal is to reflect faithfully in B all the recognised Z conventions in operation specification, so it remains for us to "totalise" our repaired operations *safe_incx* and *safe_incy* by taking their weakest feasible completions. In practice this means changing guards into preconditions. Thus we finally arrive at the following definitions of our total operations *tot_incx* and *tot_incy*:

$$
\begin{array}{l}
\quad tot\_incx \mathrel{\widehat{=}} \\
\qquad \mathrm{PRE} \\
\qquad\quad x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100 \wedge \\
\qquad\quad x < 100 \\
\qquad \mathrm{THEN} \\
\qquad\quad x, y := x + 1, y - 1 \\
\qquad \mathrm{END}
\end{array}
$$

and

$$
\begin{array}{l}
\quad tot\_incy \mathrel{\widehat{=}} \\
\qquad \mathrm{PRE} \\
\qquad\quad x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y = 100 \wedge \\
\qquad\quad y < 100 \\
\qquad \mathrm{THEN} \\
\qquad\quad x, y := x - 1, y + 1 \\
\qquad \mathrm{END}
\end{array}
$$

4.3 A New Specification Construct for B

We have just seen how an arbitrary abstract machine can be made feasible and safe after the manner of conventional Z, by

1. parallel-composing its actual and characteristic initialisations;
2. parallel-composing each of its actual operations with its characteristic operation;
3. "totalising" the resulting composite operations by taking their weakest completions.

This leads us to propose a new syntactic category for B which we will call a *SAFE_MACHINE*. Each safe machine can be syntactically expanded to an equivalent safe ordinary abstract machine by the systematic process we have just described. For example, suppose we have

SAFE_MACHINE
SM
VARIABLES
x
INVARIANT
Inv
INITIALISATION
U
OPERATIONS
$op1 \mathrel{\widehat{=}} S$;
$op2 \mathrel{\widehat{=}} T$
END

We can syntactically expand SM into the abstract machine

MACHINE
SM
VARIABLES
x
INVARIANT
Inv
INITIALISATION
$U \parallel (x : Inv)$
OPERATIONS
$op1 \mathrel{\widehat{=}} (Inv \wedge fis(S \parallel x : Inv)) \mid (S \parallel x : Inv)$;
$op2 \mathrel{\widehat{=}} (Inv \wedge fis(T \parallel x : Inv)) \mid (T \parallel x : Inv)$
END

At this point the machine SM can be further reduced by applying our elimination rules for $\parallel$, and so be expressed as a classical B machine. Alternatively we can by utilising our characteristic refinement rules for $\parallel$ choose to refine SM directly into a conventional B refinement or implementation.

4.4 How Safe Is a Safe Machine?

It could be argued that –paradoxically– a Safe Machine would be far from safe for another developer to introduce into his own development, since its operations' real preconditions are not explicit in its text. This would be quite true if indeed its author simply published the unqualified source text of his Safe Machine into his organisation's software component catalogue. Clearly we must insist that the translation tool which translates his Safe Machine into its corresponding conventional B abstract machine must also extract the real precondition of each operation in an explicit form. This should then be used to annotate the text of the Safe Machine which will ultimately appear in the component catalogue.

5 The Electronic Thesaurus

We now discuss a more significant application which illustrates the expressive power of our new safe machine construct. A thesaurus is essentially a grouping of the words of a language such that all the words in any given group are associated in meaning. Some words will appear in more than one group because they have several different meanings. Suppose we wish to specify an electronic thesaurus. We will assume the given sets $GROUP$ and $WORD$. We could model the thesaurus simply as the relation

$$contains : GROUP \leftrightarrow WORD$$

or alternatively as the function

$$comprises : GROUP \nrightarrow \mathbb{F}_1\ WORD$$

Some of the maintenance operations we wish to specify will be easier to express in terms of the relation *contains* while others will be easier to express in terms of the function *comprises*. For example, establishing a new group with its inaugural set of words is best expressed using *comprises*. Removing an individual thesaurus entry, on the other hand, is best expressed using *contains*, since when we remove the last or only word from a group we must also dis-establish the group itself. If we were specifying our thesaurus as a classical B abstract machine we would have to commit ourselves to either one representation or the other. Inevitably the choice we are forced to make will result in some of our operations being difficult to express. Using a definition to represent the omitted component would be to no avail, because defined identifiers cannot be actively updated in machine operations.

In contrast, by specifying our thesaurus as a safe machine we gain the best of both worlds. We declare the state of the thesaurus to consist of both the relation and the function. We link these together via the machine invariant thus:

SAFE_MACHINE

Thesaurus

SETS

$GROUP$;
$WORD$

VARIABLES

$contains, comprises$

INVARIANT

$contains \in GROUP \leftrightarrow WORD \wedge$
$comprises \in GROUP \nrightarrow \mathbb{F}_1\ WORD \wedge$
$comprises = \lambda\, gg\ .\ (gg \in \mathrm{dom}\ contains \mid contains[\{gg\}])$

INITIALISATION

$contains := \{\}$

OPERATIONS

$new\_group(words) \mathrel{\widehat{=}}$

PRE
$words \in \mathbb{F}_1\ WORD \wedge$
$\mathrm{dom}\ comprises \neq GROUP$
THEN
ANY
gg
WHERE
$gg \in GROUP - \mathrm{dom}\ comprises$
THEN
$comprises(gg) := words$
END
END ;

$remove\_entry(gg, ww) \mathrel{\widehat{=}}$

PRE
$gg \in GROUP \wedge ww \in WORD \wedge$
$gg \mapsto ww \in contains$
THEN
$contains := contains - \{gg \mapsto ww\}$
END

END

The extra complexity of the *Thesaurus* safe machine's invariant is amply compensated by the ease with which its operations are subsequently expressed. A full *Thesaurus* would of course involve many more operations than the two we have presented above, so the benefit would be correspondingly magnified.

6 On Framing

There are no completely free lunches, and so there is a price to pay by the B developer for being relieved of the obligation to take explicit steps to ensure his operations preserve his machine's invariant. When he is writing an operation of a Safe Machine he can no longer be certain that any machine state variable which he hasn't explicitly changed will in fact remain unchanged. Some will doubtless argue that this robs us of so convenient a facet of our B operations as to render the whole concept of a Safe Machine impractical.

We would counter that the concern can adequately be addressed by appropriate framing –that is, encapsulating of variables inside subsidiary machines. If two variables can be changed independently we should question why they are framed in the same machine, rather than being encapsulated in separate subsidiary machines included by the original one.

7 Conclusion

We contrasted B's current use of machine invariants in a purely passive verification role during specification, against Z's more constructive use of state invariants in specification. We argued that, in the context of specification, the Z usage is to be preferred, since it gives more expressive leverage to the specifier and frees him from the obligation to demonstrate his specifications are consistent, because they will automatically be so. We have shown how a single straightforward and very natural extension to just one of the constructors of B's Generalised Substitution Language is the only change needed in B's underlying semantics to provide us with a means of writing B specifications which constructively use their machine invariants. We have seen how the $\Delta State$ construction so familiar to Z users has a direct analogy in B as the characteristic operation of an abstract machine which we have called *Preserve*. We have shown how fusing *Preserve* with the particular body of an abstract machine operation is guaranteed to produce a safe operation which respects the machine invariant.

Along the way we have made various proposals to complete the theory of the multiple composition operator in B. We have also introduced a number of other subsidiary new constructions, which we hope may themselves prove to be of value in helping to promote a deeper appreciation of some of B's underlying concepts.

Acknowledgements

I am indebted to my Teesside colleagues Bill Stoddart and Richard Shore, and to Andy Galloway of York, for countless fruitful discussions on many of the ideas in this paper. I also wish to thank my anonymous referees for their encouraging comments.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Moller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 128–158. Springer Verlag, 1995.

[3] R.J.R. Back and M.J. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35(11):921–940, 1998.

[4] D. Bert, M.-L. Potet, and Y. Rouzaud. A study on components and assembly primitives in B. In H. Habrias, editor, *Proceedings of the First B Conference*, pages 47–62. IRIN, Nantes, 1996.

[5] P Chartier. Formalisation of B in Isabelle/HOL. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method; Proceedings of the Second International B Conference, Montpellier, France*, number 1393 in Lecture Notes in Computer Science, pages 66–82. Springer Verlag, 1998.

[6] C.B. Jones. *Systematic Software Development Using VDM (2nd edn)*. Prentice-Hall, 1990.

[7] C. Morgan. *Programming from Specifications (2nd edn)*. Prentice Hall International, 1994.

[8] J.M. Spivey. *The Z Notation: a Reference Manual (2nd edn)*. Prentice Hall International, 1992.