Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines — UMR 8530 CNRS

Recherche Opérationnelle et Informatique

Note de Recherche

n° 15 / 2004 / LAMIH-ROI

A natural extension of B substitutions : postconditions

Samuel Colin, Georges Mariano, Vincent Poirriez

Soumis le 21/06/2004
FSTTCS 2004 (Foundations of Software Technology and Theoretical Computer Science)

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT-CAMBRÉSIS
LE MONT HOUY — F-59313 VALENCIENNES CEDEX 9 — FRANCE
Secrétariat : Mme Aureggi – phone: +33 (0)3 27 51 19 41
`www.univ-valenciennes.fr/LAMIH`

A natural extension of *B* substitutions : postconditions

Samuel COLIN[1,2] Georges MARIANO[1], Vincent POIRRIEZ[2]

[1] INRETS*, 20, rue Elisée RECLUS, BP 317 F-59666 Villeneuve d'Ascq Cedex, France
{First name}.{Last name}@inrets.fr
[2] LAMIH**, UMR CNRS 5830, Le Mont Houy, 59313 Valenciennes Cedex 9, France
{First name}.{Last name}@univ-valenciennes.fr

Abstract. New approaches of programming and new extensions of the B method seem to make appear the lack in the ***B*** method of postconditions. This paper presents a possible way for adding postconditions to the ***B*** method without breaking its foundations and shows some of the interesting consequences of this addition, with the help of examples.

1 Introduction

The ***B*** method is a formal method based on four paradigms :

– Predicate calculus with set theory, which is both a mean of expressing properties and checking the correctness of ***B*** machines
– Substitutions to describe the dynamics of the state of the variables
– Modularity, which allows the sharing and the reuse of ***B*** developments
– Refinement, which allows the gradual progression from the abstract, mathematical specification, to the final computer code

Roughly speaking, a ***B*** machine is said correct if its invariant hold. The invariant is a predicate describing the state of the machine before and after the evaluation of a substitution[3].

Since it was presented in [Abr96], numerous extensions have been proposed, some of which adding or expressing the need for postconditions in ***B*** operations :

– ***event B***[Cle01], postconditions are introduced as a mean to express the state of the variables of the machine w.r.t. their state before the corresponding event was triggered
– [Pet03] introduces postconditions for two purposes :
  • Making ***B*** a contract-based development method, the precondition and the postcondition of an operation being normally to describe the behaviour of the substitution of this operation.
  • Making the final computer code more trustworthy : in that case, the postcondition is an additional assertion that can be embedded into the, say, final C program, to ease the debugging of tests.
– The author itself uses postconditions as a support predicate for the generation of temporal logic formulas. This work has not yet been published, but the interested reader can see [CPM03] for a presentation of this temporal logic and the properties of its implementation in a general-purpose theorem-prover.

Thus, the need for postconditions is flagrant. The main problem is, while postconditions are well integrated into ***event B***, what is needed to have them into the "classical" ***B*** method ? That is why we present in this paper a possible definition of postconditions in the ***B*** method, by using the inner foundations of the formalism to do so.

In the first section, we remind the reader of the properties and the mechanisms of the substitutions of the ***B*** method, then we describe the actual definition of postconditions by using this mechanism, and illustrate the impacts of this definition to the way ***B*** machines are verified. Then we give some examples of the use of postconditions, and conclude with the new insights brought by this extension.

* Institut National de REcherche sur les Transports et leur Sécurité
** Laboratoire d'Automatique, de Mécanique, et d'Informatique industrielles et Humaines
[3] Note that in other formalisms, substitutions are rather called "state transition"

2 The *B* method

This section first reminds the reader of the ***B*** method, then presents a typical development with the ***B*** method with the help of examples, and finally describes the mechanisms of the basic substitutions of ***B*** as well as their properties.

2.1 Presentation of the *B* method

The ***B*** method allows to generate computer code, safe and prove to be correct with respect to mathematical formal expressions. These mathematical expressions are gathered into a ***B*** component called *abstract machine*. They are then refined by one or several components called *refinements*. After all these successive refinement steps, we obtain the last component called *implementation*. All these components compose a ***B*** *module*. A module can depend on several other modules, in different ways. This set of modules then forms a ***B*** project.

A ***B*** abstract machine is a description of a state that can evolve according to transitions described by the operations of the machine. The main clauses of a ***B*** machine are :

MACHINE describing the name of the machine

SETS introducing the sets of the machine. Those sets are either abstract, or defined by the enumeration of their elements.

VARIABLES defining the state of the machine with the help of so-called variables. The variables are the names referred to when an update or a description of the state of the machine is needed, and can be only modified by the operations of the machine.

INVARIANT is a predicate stating the state of the machine between operation updates. This clause must be present if the VARIABLES clause is specified.

INITIALISATION is a ***B*** substitution defining the initial state of the machine, by specifying the values of the machine's variables.

OPERATIONS is a clause composed of different substitutions, whose role is to describe the dynamical behaviour of the machine (i.e. the possible state transitions of the machine).

2.2 An example of *B* development

The MACHINE clause can be replaced by a REFINEMENT or IMPLEMENTATION one : in that case, an additional REFINES clauses indicating which machine is refined must be provided. Now, for a ***B*** machine to be correct, one has to ensure that its initialisation establishes the invariant, and that the invariant is kept when updating the state of the machine after an operation. A ***B*** refinement (or implementation) is correct if its operations are not inconsistent with those of the machine it refines, i.e. if called within the same state of variables, an operation yields at least the same result (or a more precise one) than the operation it refines.

Figure 1 shows a toy example (taken from [Mar97]) of a ***B*** development.

The machine *LittleExample* defines a variable y representing a set of natural numbers $y \in \mathbb{F}(\mathbb{N}^*)$, initialised as an empty set $y := \emptyset$, and whose possible evolutions are described by the operation *read* (which update the variable) and *maximum* (which returns the maximum value of the set). Intuitively, the role of this machine is to return the maximum integer among those it has been feed with.

Then comes its refinement *LittleExample1*, introducing a new variable z. This variable is initialised at the value 0, and its evolution is also described by operations named as in the machine *LittleExample*. The so-called *gluing* invariant of *LittleExample1* allows to describe the relationship between z and y. Intuitively, this refinement behaves the same way as the machine it refines : it keeps the maximum of the integers it has been feed with through *read*, and sends it through the operation *maximum*.

The ***B*** method allows to check that the invariant of *LittleExample* is preserved by both the initialisation and the operation, and that *LittleExample1* can replace *LittleExample* without machines depending on it noticing (which is another way of stating what a refinement is).

From the machines, proof obligations are generated according to specific rules. They are logical formulas whose proof helps ensure that the machines are correct. For instance, example 1 illustrates the proof obligation of the *read* operation : $[y := y \cup \{n\}]$ is a substitution that is applied to the invariant in order to obtain the weakest precondition (that is, the requirement) so as to ensure that the hypotheses (the invariant and the precondition) fulfil this precondition. The rules for obtaining a weakest precondition from a substitution and a predicate are presented in section 2.3. In example 1, after applying the substitution, one can see that the formula is trivially correct.

MACHINE LittleExample **VARIABLES** y **INVARIANT** y ∈ 𝔽(NAT1) **INITIALISATION** y :=∅ **OPERATIONS** read(n) = **PRE** n ∈ NAT1 **THEN** y :=y ∪{n} **END**; m ⟵maximum = **PRE** y ≠∅ **THEN** m :=max(y) **END**	**REFINEMENT** LittleExample1 **REFINES** LittleExample **VARIABLES** z **INVARIANT** z = max(y ∪{0}) **INITIALISATION** z :=0 **OPERATIONS** read(n) = **PRE** n ∈ NAT1 **THEN** z :=max(z,n) **END**; m ⟵maximum = **PRE** z ≠0 **THEN** m :=z **END**

Fig. 1. Machine LittleExample and its refinement

Example 1.

$$y \in \mathbb{F}(\mathbb{N}^*) \wedge n \in \mathbb{N}^* \Rightarrow [y := y \cup \{n\}](y \in \mathbb{F}(\mathbb{N}^*))$$

After application of the substitution :

$$y \in \mathbb{F}(\mathbb{N}^*) \wedge n \in \mathbb{N}^* \Rightarrow y \cup \{n\} \in \mathbb{F}(\mathbb{N}^*)$$

Example 2 presents an example of refinement proof obligation : the particular form of this formula expresses that the *maximum* operation should not establish a state where the refined one is not defined.

Example 2.

$$y \in \mathbb{F}(\mathbb{N}^*) \wedge z = max(y \cup \{0\}) \wedge y \neq \emptyset \Rightarrow z \neq 0 \wedge [[m := m']m := z]\neg[m := max(y)]\neg(z = max(y \cup \{0\}) \wedge m = m')$$

2.3 Generalised substitutions : definition, properties

The *generalised substitutions*[4] are the core mechanism of the ***B*** method for the description of the evolution of the machines' state. Other substitutions presented in [Abr96] are actually syntactic sugar for GSL.

Figure 2 presents the basic substitutions of ***B***. Note there are more elaborate ones, like the *WHILE* loop, but we don't present them in the figure 2 because there are not necessary to understand how GSL are used.

Let us describe more precisely the role of each of those substitutions :

skip does nothing. It is used to replace a dynamic behaviour one wants only to specify in a refinement, or to build more complex substitutions (the "IF THEN ELSE" one, for instance).

$x := E$ is a simple assignment : the x variable now corresponds to the E value.

$P|S$ specifies that the predicate P should be checked statically before the S substitution can be applied. Note that this substitution doesn't state anything about the behaviour of the substitution in the case it is "executed" out of the precondition.

[4] abbreviated GSL from now on

GSL	$[GSL]P$	description
$skip$	P	"Do nothing" substitution
$x := E$	$P[E/x]$	All the occurrences of x are replaced by E
$P \mid S$	$P \wedge [S]P$	Precondition
$P \Longrightarrow S$	$P \Rightarrow [S]P$	Guard
$S [\!] T$	$[S]P \wedge [T]P$	Bounded choice
$@x.S$	$\forall x [S]P$	Unbounded choice

Fig. 2. Calculus of the weakest precondition

$P \Longrightarrow S$ is the dual of the precondition : it states that the substitution S is applied only if the guard P proves to be true at execution.

$S [\!] T$ states that either S or T is to be "executed". This substitution is non-deterministic, i.e. we don't know which of its inner substitutions will be chosen at runtime. This substitution can be made more deterministic with the help of guards (see below).

All these simple substitutions can then be used to build more complex control structures. For instance, the conditional statement is defined as in example 3. The corresponding one-branch conditional statement (namely "IF THEN") is obtained by making T equal to $skip$.

Example 3.

$$\text{IF } P \text{ THEN } S \text{ ELSE } T \equiv P \Longrightarrow S [\!] \neg P \Longrightarrow T$$

Substitutions are also given a mechanism to calculate their weakest precondition with respect to a given predicate : the obtained formula is then a predicate describing the minimal (weakest) required state for the substitution to establish the predicate it was given. In fact, this is exactly the way ***B*** machines are checked : by proving that the invariant is established by any operation of the machine (as well as the initialisation of the machine). The rules to calculate the weakest precondition of a substitution with respect to a predicate are presented in figure 2.

Then, after seeing all these formal definitions, the question arises : is there a way to characterise more formally GSL ? The answer is affirmative, is is also presented in [Abr96]. The common, underlying shape of GSL is :

Definition 1.

$$S \equiv P \mid @x'.(Q \Longrightarrow x := x')$$

where P and Q are predicates and where x' is a variable distinct from x having no free occurrence in P. The predicate Q depends on x and x'.

Then comes another question : what are those predicates, precisely ? Before [Abr96] answers this question, it introduces formal definitions for reasoning and characterising substitutions. These definitions are showed in figure 3.

GSL	$trm(GSL)$	$prd_x(GSL)$
$skip$	True	True
$x := E$	True	True
$P \mid S$	$P \wedge \mathbf{trm}(S)$	$P \Rightarrow \mathbf{prd}_x(S)$
$P \Longrightarrow S$	$P \Rightarrow \mathbf{trm}(S)$	$P \wedge \mathbf{prd}_x(S)$
$S [\!] T$	$\mathbf{trm}(S) \wedge \mathbf{trm}(T)$	$\mathbf{prd}_x(S) \vee \mathbf{prd}_x(T)$
$@z.S$	$\forall z.\mathbf{trm}(S)$	$\exists z.\mathbf{prd}_x(S)$
$P \mid @x'.(Q \Longrightarrow x := x')$	P	$P \Rightarrow Q$
$x : P$	True	$[x, x_0 := x', x]P$

Fig. 3. Termination and before-after predicate of substitutions

Let us precise what do the concept of *termination* and *before-after predicate* mean :

Termination is the minimal required condition for a substitution to be able to define a predicate. It is calculated for the substitution S by applying the formula : $[S](x = x)$.

The before-after predicate prd_x describes the value of the variable x when it has been "changed" by the substitution. Of course, if x doesn't appear in the substitution, then the obtained formula describes exactly that x has not changed, as expected. The before-after predicate is calculated by (for the variable x) : $\neg[S](x' \neq x)$. The x' represents the state of x w.r.t its before-value, x in the formula.

The example given in [Abr96] is :

Example 4. $prd_x((x := x+1) [] (x := x-1)) \Leftrightarrow (x' = x+1) \vee (x' = x-1)$

, which describes accurately the dynamics of the substitution.

We have also added the termination and before-after predicate for the general substitution rule and the *predicate statement*. The former is given for references purposes, but the latter is an interesting substitution : it defines the state of a variable in comprehension. For instance, $x : (x = x_0 + 1)$ is such a substitution, and states that the new value of x is actually the *before* value of x plus one. The $_0$ index refers to the value of the variable before the substitution is applied. This *predicate statement* is actually a shortcut for a more complex definition, as showed in figure 4. We mention this substitution, because we will refer to it in section 3.

$x : P$	$@x'.$ $[x, x_0 := x', x]P$ $\Longrightarrow$ $x := x'$

Fig. 4. Definition of the predicate statement

Note that in figure 4, x' could have been named differently : it is just a helper to remind of the state variable it represents.

Then, with the help of termination and before-after predicate, [Abr96] makes the proof that the general shape of substitutions is actually :

Definition 2.

$$S \equiv trm(S) | @x'.(prd_x(S) \Longrightarrow x := x')$$

Thus any substitution can be easily defined, provided that the termination and the before-after predicate are known. This mechanism is used in [Abr96] to define the parallel composition of substitutions.

2.4 A note on variables' renamings

The section 3.3 will illustrate the use of postconditions from the point of view of another (including) machine. That's why we need to remind the reader of the problem of variables' renaming when replacing an operation call with the actual body of the called operation, and replacing the operation parameters with those of the call.

This problem is solved by the definition of substitution rules on substitutions : the rules are trivial, except for substitutions on substitutions that might result in an ill-formed substitution. These rules are described in [Abr96, appendice E.1].

Example 5. Let us suppose $x : (P(x,y))$ is the body of the operation $x \longleftarrow Oper(y)$, and that $P(x,y)$ is a predicate where no x_0 appears (which is not possible as x is an output parameter). What happens if the operation is called with $z \longleftarrow Oper(z)$?

$$\begin{aligned}
& [x,y := z,z](x \longleftarrow Oper(y)) \\
\Rightarrow\ & [x,y := z,z](x : P(x,y)) \\
\Rightarrow\ & [x,y := z,z](@x'.[x,x_0 := x',x]P(x,y) \Longrightarrow x := x') \\
\Rightarrow\ & [x,y := z,z](@x'.P(x',y) \Longrightarrow x := x') \\
\Rightarrow\ & @x'.[x,y := z,z]P(x',y) \Longrightarrow [x,y := z,z](x := x') \\
\Rightarrow\ & @x'.P(x',z) \Longrightarrow (z := x')
\end{aligned}$$

That is, z is updated with a value referring to itself, and verifying the predicate P, which the expected behaviour. Notice the replacement of x with z in the substitution $x := x'$: such a replacement is allowed only if the result in the left-hand side of an assignment is a variable.

3 Adding postconditions to *B*

3.1 The existing definitions of postcondition

Postconditions have been introduced in [Cle01]. They allow the developer to express more precisely the state of the variables after the corresponding event has been triggered. Then one can state more complex properties of ***event B*** machines, such as deadlock-freeness for instance.

The postconditions also have been introduced in [Pet03, 4.3.3] for the ***B*** method. They are used to allow a contract-based development method, where the contracts are represented by the preconditions and the postconditions of these operations. Furthermore, these contracts can be embedded in the target language if it permits it. For example, in [Pet03], the contracts from the ***B*** model can be transformed into assertions if the target language is OCaml.

```
remove_element =
  BEGIN
    ANY
      element
    WHERE
      element ∈ set ∧enable_remove =true
    THEN
      set :=set −{ element }
    || enable_remove :=bool ( set ≠∅)
    END
  POST
    card(set) <card(set$0)
  END
```

Fig. 5. An example of postcondition, taken from [Cle01]

Let us remember the predicate statement in section 2.3 : this substitution seems ideal to achieve the effect we are looking for. Actually, this substitution has some drawbacks preventing us to use it as a postcondition :

– The scope of its indiced variables is not enough to embrace a complex substitution construction (see example 6) : the only way to have this effect would be to use a refinement step, where the abstract operation is the lone specification statement, and the refinement operation the substitution that actually establishes this statement.
– Except when using the refinement step indicated above, the predicate statement does not allow the hiding of properties that would be irrelevant for including machines. Indeed, it can appear anywhere in an operation, not specifically at the end.

Example 6. Let $P(set)$ be a predicate about the set variable, S the body of the operation in figure 5, and $Post$ its postcondition. Let us suppose we want to know the weakest precondition so that the operation $remove\_element$ establishes $P(set)$. Then, using the rules and notations from [Abr96], we have :

$$
\begin{aligned}
[S; set : Post](P(set)) &= [S; @set'.([set_0, set := set, set']Post \Longrightarrow set := set')](P(set)) && (1)\\
&= [S](\forall set'.card(set') < card(set) \Rightarrow P(set') && (2)\\
&= \forall element.element \in set \wedge enable\_remove = TRUE && (3)\\
&\quad \Rightarrow (\forall set'.card(set') < card(set \setminus \{element\}) \Rightarrow P(set')) && (4)
\end{aligned}
$$

We see that intuitively, the result is not what was expected : the $set \setminus \{element\}$ should have appeared in the left-hand part of the inequation.

That is why we need to define formally, *à la BBook*, how we can handle postconditions.

3.2 Defining the POST substitution

In [Abr96, 6.3], we see that any substitution can be characterised completely by its termination (trm) and its before-after predicate (prd_x, x representing the variables of the machine). Then the author show how to use this mechanism to define formally the multiple generalised substitution ($||$).

For readability purposes, we will note the postcondition substitution in the GSL[5] as $S \rhd P$, where S is a substitution and P a predicate (the actual postcondition).

Termination of a postcondition What does it mean for $S \rhd P$ to terminate ? The termination of a substitution S, as stated in [Abr96], is "the predicate that holds when the substitution S 'terminates'". In other words, it is the predicate stating all the needed conditions for S to establish something. Assuming x is a modified variable in S, let us define $trm(S \rhd P)$ as :

$$trm(S \rhd P) \equiv [x_0 := x]([S]P)$$

This formula reads as : "Assuming that S is a substitution and P a predicate, a postcondition built on top of these terminates (i.e. is able to establish something) if we can verify that P is established by S". In other words, knowing that we mainly rely on the postcondition at the proof stage, we want S to reflect and establish the properties we state in its postcondition.

Before-after predicate of a postcondition Let us assume that x is a modified variable in S. We then define the before-after predicate of $S \rhd P$ w.r.t. the variable x as :

$$prd_x(S \rhd P) \equiv [x_0, x := x, x']P$$

As indicated in section 2.3, the before-after predicate reflects the possible states of the variables after a substitution has been applied. However, we want the postcondition to be an expression of what we want for the state of the machine w.r.t. its state before the substitution was applied. Thus, the before-after predicate of a postcondition is no more, no less than the postcondition itself (with the appropriate variables' renaming).

Formal definition of the postcondition As indicated in [Abr96, section 6.3.3], the predicates $trm(S)$ and $prd_x(S)$ characterise completely the generalised substitution S. Thus we can give the final definition of the postcondition :

Definition 3. *Assuming that x represents the modified variables of S, then we have :*

$$S \rhd P \equiv [x_0 := x]([S]P) | @x'.([x_0, x := x, x']P \Longrightarrow x := x')$$

Then, to give an illustration of the use of postconditions, and to ensure that the definition is correctly written, let us check that we fall back on $trm(S \rhd P)$ and $prd_x(S \rhd P)$ by using their actual definition.

Example 7.

$$\begin{array}{lr}
\quad trm(S \rhd P) & \\
\Leftrightarrow [S \rhd P](x = x) & \text{def. of } trm \\
\Leftrightarrow [[x_0 := x]([S]P) | @x'.([x_0, x := x, x']P \Longrightarrow x := x')](x = x) & \text{def. of } S \rhd P \\
\Leftrightarrow [x_0 := x]([S]P) \wedge \forall x'.([x_0, x := x, x']P \Rightarrow [x := x'](x = x))) & \text{rules for GSL} \\
\Leftrightarrow [x_0 := x]([S]P) \wedge \forall x'.([x_0, x := x, x']P \Rightarrow x' = x')) & \text{def. of affectation} \\
\Leftrightarrow [x_0 := x]([S]P) & \text{predicate logic}
\end{array}$$

Example 8.

$$\begin{array}{lr}
\quad prd_x(S \rhd P) & \\
\Leftrightarrow \neg[S \rhd P](x' \neq x) & \text{def. of } prd_x \\
\Leftrightarrow \neg([[x_0 := x]([S]P) | @x''.([x_0, x := x, x'']P \Longrightarrow x := x''](x' \neq x))) & \text{def. of } S \rhd P\text{, } \alpha\text{-renaming} \\
\Leftrightarrow \neg([x_0 := x]([S]P) \wedge \forall x''.([x_0, x := x, x'']P \Rightarrow [x := x''](x' \neq x))) & \text{rules for GSL} \\
\Leftrightarrow (\neg[x_0 := x]([S]P)) \vee \exists x''.\neg([x_0, x := x, x'']P \Rightarrow (x' \neq x'')) & \text{def. of affectation, predicate logic} \\
\Leftrightarrow False \vee \exists x''.\neg(\neg[x_0, x := x, x'']P \vee (x' \neq x'')) & \text{assumption of } trm(S \rhd P)\text{, predicate logic} \\
\Leftrightarrow \exists x''.([x_0, x := x, x'']P \wedge x' = x'') & \text{predicate logic} \\
\Leftrightarrow [x_0, x := x, x']P & \text{predicate logic}
\end{array}$$

[5] Generalised Substitution Language

Examples 7 and 8, as expected, show us that our definition is sound.

3.3 Impact on proof obligations

To illustrate the influence of postconditions on proof obligations, let us first define a notation to refer to the different parts of a ***B*** project. This notation is presented in figure 6. All indiced identifiers refer to a machine in a refinement sequence (with the abstract machine corresponding to the index 1). Including machines are represented by another letter (in figure 6, N is a machine that includes M_1). u and w identifiers represent the output and the input parameter of an operation respectively, v represents the variables of the machine. Now the identifiers representing more complex structures (predicates or substitutions) :

- I represents the invariant.
- P and Q represent the precondition and the postcondition of an operation respectively.
- S represents the body of an operation.
- OP represents the name of an operation.

MACHINE M_1 **VARIABLES** v_{M_1} **INVARIANT** I_{M_1} **OPERATIONS** $u_{M_1} \longleftarrow OP_{M_1}(w_{M_1}) =$ **PRE** P_{M_1} **THEN** S_{M_1} **POST** Q_{M_1} **END**	**MACHINE** N **INCLUDES** M_1 **VARIABLES** v_N **INVARIANT** I_N **OPERATIONS** $u_N \longleftarrow OP_N(w_N) =$ **PRE** P_N **THEN** S_N $;\ x_N \longleftarrow OP_{M_1}(y_N)$ $;\ T_N$ **END**	**REFINEMENT** M_2 **REFINES** M_1 **VARIABLES** v_{M_2} **INVARIANT** IM_2 **OPERATIONS** $u_{M_2} \longleftarrow OP_{M_2}(w_{M_2}) =$ **PRE** P_{M_2} **THEN** S_{M_2} **END**

Fig. 6.

Then, let us see how the use of a postcondition influences proof obligations. The proof obligation for the machine M_1 of figure 6 is as follows :

$$I_{M_1} \wedge P_{M_1} \Rightarrow [S_{M_1} \rhd Q_{M_1}]I_{M_1} \tag{5}$$

Note that we removed the unnecessary clauses for readability reasons. After expanding (5), we obtain :

$$I_{M_1} \wedge P_{M_1} \Rightarrow \begin{cases} [v_{M_1 0} := v_{M_1}]([S_{M_1}]Q_{M_1}) \\ \wedge \forall v_{M_1}', u_{M_1}'.([v_{M_1 0}, v_{M_1}, u_{M_1 0}, u_{M_1} := v_{M_1}, v_{M_1}', u_{M_1}, u_{M_1}']Q_{M_1} \Rightarrow [v_{M_1}, u_{M_1} := v_{M_1}', u_{M_1}']I_{M_1}) \end{cases} \tag{6}$$

Now let us see how, assuming the proof obligations for the machine have been validated, it influences the proof obligations depending on this machine :

Influence on included machines Here is the proof obligation for a machine including other machines :

$$I_N \wedge P_N \wedge I_{M_1} \Rightarrow [S_N; x_N \leftarrow OP_{M_1}(y_N); T_N]I_N \tag{7}$$

Let us define first a more convenient notation to ease the readability of coming formulas.

Definition 4.

$$\begin{aligned} PostRen1_x &\equiv [x, x_0 := x', x] \\ PostRen2_x &\equiv [x := x'] \\ InstRen_x(y) &\equiv [x := y] \end{aligned}$$

PostRen is the renaming created by the postcondition, and *InstRen* is the renaming caused by the instantiations of operations' parameters.

After expanding the operation call, we obtain :

$$I_N \wedge P_N \wedge I_{M_1} \Rightarrow [S_N] \begin{cases} [InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1} \\ \wedge[[InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)][v_{M_1 0} := v_{M_1}][S_{M_1}]]Q_{M_1} \\ \wedge[[InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)](\forall v_{M_1}{}'.([PostRen1_{v_{M_1}, u_{M_1}}]Q_{M_1} \\ \qquad \Rightarrow [PostRen2_{v_{M_1}, u_{M_1}}]]([T_N]I_N))) \end{cases} \tag{8}$$

Let us now assume we have proved the subgoal : $[S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1})$. We know by predicate logic that we have :

$$\forall P, Q, R, (P \Rightarrow Q \wedge R) \Rightarrow (P \wedge Q \Rightarrow R) \tag{9}$$

Thus, we can introduce $[S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1})$ (namely, the precondition of the called operation) in the hypotheses. Additionally, by monotony of the substitution application and by the fact I_{M_1} does not contain any variable modified by the substitution $[S_N][InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]$, we have :

$$\begin{aligned} & I_{M_1} \wedge [S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1}) \\ & \Rightarrow \\ & [S_N]([[InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)][S_{M_1} \rhd Q_{M_1}]]I_{M_1}) \end{aligned} \tag{10}$$

What happens if we expand (10) a little, and keep the subgoal *not* depending on I_{M_1} ? We obtain :

$$\begin{aligned} & I_{M_1} \wedge [S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1}) \\ & \Rightarrow \\ & [S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]([v_{M_1 0} := v_{M_1}]([S_{M_1}]Q_{M_1}))) \end{aligned} \tag{11}$$

Thus, assuming $I_N \wedge P_N \wedge I_{M_1} \Rightarrow [S_N]([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1})$ from (8), and by (9) and (11), we then obtain for the proof obligation for an including machine :

Theorem 1. *If the operation of a machine contains an operation call to an included machine, and the called operation contains a postcondition, like in figure 6, then the proof obligation for this operation has the following shape :*

$$I_N \wedge P_N \wedge I_{M_1} \Rightarrow [S_N] \begin{cases} [InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)]P_{M_1} \\ \wedge \forall v_{M_1}{}', u_{M_1}{}'.([InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)][PostRen1_{v_{M_1}, u_{M_1}}]Q_{M_1} \\ \quad \Rightarrow [[InstRen_{u_{M_1}, w_{M_1}}(x_N, y_N)][PostRen2_{v_{M_1}, u_{M_1}}]]([T_N]I_N))) \end{cases}$$

Intuitively, all this demonstration shows that if we are able to prove that the precondition of the called operation is verified, then we can remove from the subgoals of the proof obligation the part where we have to show that the body of the called operation establishes its postcondition. Luckily, the proof obligation already requires us to prove the precondition of the called operation.

On a more pragmatic note, this demonstration states that we do not need to know the actual body of the called operation, only its pre- and postcondition. Also note there is no obligation for the final tool to remove the verification from the including machine's proof obligation that the body establishes the postcondition, because any well-designed theorem prover, knowing about the proof obligations of the included machines, can deduce that by itself.

Influence on refinements Now, let us see how the use of a postcondition in an abstract machine can have an influence on its refinements. Note that, due to the appearance of a primed u_{M_1}, the definitions of *PostRen*1 and *PostRen*2 are slightly changed : instead of priming once the variables, they prime them twice (we simply apply the rules to avoid a conflict between variables' names). The proof obligation for a refinement has the following shape :

$$I_{M_1} \wedge I_{M_2} \wedge P_{M_1} \Rightarrow P_{M_2} \wedge [[u_{M_1} := {u_{M_1}}']S_{M_2}]\neg[S_{M_1} \rhd Q_{M_1}]\neg(I_{M_2} \wedge u_{M_1} = {u_{M_1}}') \tag{12}$$

If we expand formula (12), we obtain :

$$\begin{array}{l} I_{M_1} \wedge I_{M_2} \wedge P_{M_1} \Rightarrow \\ \begin{cases} \quad P_{M_2} \\ \wedge\ [[u_{M_1} := {u_{M_1}}']S_{M_2}]\neg[S_{M_1} \rhd Q_{M_1}]\neg(I_{M_2} \wedge u_{M_1} = {u_{M_1}}') \end{cases} \end{array} \tag{13}$$

After expanding $S_{M_1} \rhd Q_{M_1}$, we have the following formula :

$$\begin{array}{l} I_{M_1} \wedge I_{M_2} \wedge P_{M_1} \Rightarrow \\ \begin{cases} \quad P_{M_2} \\ \wedge\ [[u_{M_1} := {u_{M_1}}']S_{M_2}]\neg(\\ \qquad\qquad [v_{M_1 0} := v_{M_1}]([S_{M_1}]Q_{M_1}) \\ \qquad\qquad \wedge \forall {v_{M_1}}'', {u_{M_1}}''.([PostRen1_{v_{M_1},u_{M_1}}]Q_{M_1} \Rightarrow [PostRen2_{v_{M_1},u_{M_1}}](\neg(I_{M_2} \wedge u_{M_1} = {u_{M_1}}')))) \end{cases} \end{array} \tag{14}$$

Then, we apply the negation :

$$\begin{array}{l} I_{M_1} \wedge I_{M_2} \wedge P_{M_1} \Rightarrow \\ \begin{cases} \quad P_{M_2} \\ \wedge\ [[u_{M_1} := {u_{M_1}}']S_{M_2}](\\ \qquad\qquad \neg[v_{M_1 0} := v_{M_1}]([S_{M_1}]Q_{M_1}) \\ \qquad\qquad \vee \exists {v_{M_1}}'', {u_{M_1}}''.([PostRen1_{v_{M_1},u_{M_1}}]]Q_{M_1} \wedge [PostRen2_{v_{M_1},u_{M_1}}](I_{M_2} \wedge u_{M_1} = {u_{M_1}}'))) \end{cases} \end{array} \tag{15}$$

The expanded formula (15) shows us an apparently strange predicate. Let us first comment about the second part of the disjunction : $[[u_{M_1} := {u_{M_1}}']S_{M_2}]$ must establish :

- $[PostRen1_{v_{M_1},u_{M_1}}]Q_{M_1}$, i.e. it must establish the postcondition of the refined operation
- $[PostRen2_{v_{M_1},u_{M_1}}]I_{M_2}$, i.e. it must establish the invariant of the refinement
- $u_{M_1} = {u_{M_1}}'$, i.e. the values returned by the operation must be the same as the ones returned by the refined operation.

In other words, the second part of the disjunction describes the usual steps required to prove a refinement is sound.

Then, what does the first part of the disjunction mean ? It means that the proof obligation is checked if the operation can establish a state where the refined operation does not terminate. After a look at figure 3 in section 2.3, we see that many substitutions have a termination that reduces to the *TRUE* predicate : that means that, when calculating a refinement proof obligation, the first part of the disjunction actually reduces to *FALSE* most of the times.

Let us have a look at the informal description of refinement in [Abr96, section 11.1.1], where *T* is the refinement and *S* the refined substitution : "*T* also "does more" in that it might terminate if started in situations where *S* would not have terminated. The fact that *T* "does more" in this case is no problem since we use *T* "as if it were *S*" (within the termination conditions of *S*). Consequently, we will never notice that *T* may do things that *S* is unable to do itself : we are merely using a refinement which is too sophisticated with respect to the corresponding abstraction."

Thus, the seemingly strange first part of the disjunction is actually another facet of the refinement in ***B*** : it states that a refinement is correct when it terminates in all cases where the refined operation did not terminate.

Another property of the postcondition in the case of a refinement is the *strengthening* : this property is confirmed by our definition of the postcondition, because, in the cause the operation of the refinement has a postcondition (say, Q_{M_2}), then the generated proof obligation has the following shape (we replaced all the renamings with [...] to focus on the shape of the formula) :

Proposition 1.

$$\begin{array}{l} I_{M_1} \wedge I_{M_2} \wedge P_{M_1} \Rightarrow \\ \begin{cases} \quad P_{M_2} \\ \wedge\ [[...]S_{M_2}]Q_{M_2} \\ \wedge\ @x''.[...]Q_{M_2} \Longrightarrow \\ \qquad\qquad \neg[...]([S_{M_1}]Q_{M_1}) \\ \qquad\qquad \vee \exists {v_{M_1}}'.([...]Q_{M_1} \wedge [...](I_{M_2} \wedge u_{M_1} = {u_{M_1}}'))) \end{cases} \end{array}$$

We can clearly see, by the guard, that Q_{M_2} has to be at least as strong as Q_{M_1}.

4 Examples of use

4.1 Example from [Cle01]

Let us generate the proof obligation of the operation *remove_element* (see figure 5) and attempt to prove it. Let :

- $\mathbb{N}$ be the set of naturals
- $\mathcal{P}(S)$ the set of all the subsets of S
- $\mathbb{B}$ the set of booleans
- $I \equiv set \in \mathcal{P}(\mathbb{N}) \wedge enable\_remove \in \mathbb{B}$ the invariant of the hypothetical machine of our example
- S the body of the operation without the postcondition
- $P \equiv card(set) < card(set_0)$ the postcondition of the operation

Then we must check the following proof obligation for the operation to be correct :

$$
\begin{aligned}
& I \Rightarrow [S \rhd P]I \\
\Leftrightarrow\ & I \Rightarrow ([set_0 := set]([S]P) \wedge \forall set'.([set_0, set := set, set']P \Rightarrow [set := set']I)) && \text{def. of } S \rhd P\text{, rules for GSL} \\
\Leftrightarrow\ & I \Rightarrow (\forall element.(element \in set \wedge enable\_remove = TRUE \Rightarrow card(set \setminus \{element\}) < card(set)) \\
& \wedge \forall set'.(card(set') < card(set) \Rightarrow set' \in \mathcal{P}(\mathbb{N}) \wedge enable\_remove' \in \mathbb{B}) && \text{rules for GSL} \\
\Leftrightarrow\ & I \Rightarrow (\forall set'.(card(set') < card(set) \Rightarrow set' \in \mathcal{P}(\mathbb{N}) \wedge enable\_remove' \in \mathbb{B}) && \text{predicate logic}
\end{aligned}
$$

We see that we miss a predicate in the postcondition ($set \in \mathcal{P}(\mathbb{N}) \wedge enable\_remove \in \mathbb{B}$) to be able to prove that the operation establishes the invariant. This is no surprise, as the definition of postconditions in [Cle01] is different from ours, and is used for ***event B***, whereas our definition of postcondition is applied in the context of the "classical" ***B*** method.

4.2 Example from [Pet03, appendice A]

MACHINE example **VARIABLES** x,y,z **INVARIANT** x ∈ *NAT1*∧y ∈ *NAT1*∧z ∈ *NAT1*∧(x +y) ≤z **OPERATIONS** illustration(param) = **PRE** param ∈ *NAT1*∧(param +z) ∈ *NAT1* **THEN** z :=z +param ; y ⟵mini(y, param) ; x :=x +1 **POST** **true** **END**...	... r ⟵mini(a,b) = **PRE** a ∈ *NAT1*∧b ∈ *NAT1* **THEN** *hidden body* **POST** r ∈ *NAT1*∧((r =a ∧a ≤b) ∨(r =b ∧a >b)) **END**...

Fig. 7. Example of an operation call with a postcondition

With our definition of postcondition, the proof obligation for the *illustration* operation does not hold : the added postcondition, $TRUE$, will result in a $TRUE \Rightarrow Invariant$ formula, which can not be proved. Thus, for the postcondition to be correct, it should have contained a formula at least as strong as the machine's invariant. Nevertheless, the example is meant to be a toy example, thus we will assume there is no postcondition (i.e. that the *illustration* operation has the shape of OP_n as in figure 6).

Thus, the proof obligation for the *illustration* operation is :

$$I_N \wedge P_N \wedge I_{M_1} \Rightarrow [S_N] \begin{cases} [InstRen_{u_{M_1},w_{M_1}}(x_N, y_N)]P_{M_1} \\ \wedge \forall v_{M_1}{}'.([InstRen_{u_{M_1},w_{M_1}}(x_N, y_N)][PostRen1_{v_{M_1},u_{M_1}}]Q_{M_1} \\ \quad \Rightarrow [[InstRen_{u_{M_1},w_{M_1}}(x_N, y_N)][PostRen2_{v_{M_1},u_{M_1}}]]([T_N]I_N))) \end{cases}$$

$$\begin{array}{l} x,y,z \in \mathbb{N}^* \wedge x+y \leq z \wedge param \in \mathbb{N}^* \wedge (param+z) \in \mathbb{N}^* \\ \Rightarrow [z := z + param] \begin{cases} [r,a,b := y,y,param](a,b \in \mathbb{N}^*) \\ \wedge \forall r'.([r,a,b := y,y,param][r_0, r := r, r'](r \in \mathbb{N}^* \wedge ((r = a \wedge a \leq b) \vee (r = b \wedge a > b))) \\ \quad \Rightarrow [[r,a,b := y,y,param][r := r']]([x := x+1](x,y,z \in \mathbb{N}^* \wedge x+y \leq z)))) \end{cases} \end{array}$$

After simplification, we obtain the following proof obligation :

$$\begin{array}{l} x,y,z \in \mathbb{N}^* \wedge x+y \leq z \wedge param \in \mathbb{N}^* \wedge (param+z) \in \mathbb{N}^* \\ \Rightarrow \begin{cases} (y, param \in \mathbb{N}^*) \\ \wedge \forall r'.(r' \in \mathbb{N}^* \wedge ((r' = y \wedge y \leq param) \vee (r' = param \wedge y > param)) \\ \quad \Rightarrow ((x+1, r', z+param \in \mathbb{N}^* \wedge x+1+r' \leq z+param))) \end{cases} \end{array}$$

Note that we put together some variables having the same domain in order to make the formula more readable. The obtained formula is easily proved (the proof is left as an exercise for the interested reader). There is even an unnecessary hypothesis $param + z \in \mathbb{N}^*$, because it can be deduced from $z, param \in \mathbb{N}^*$ (appearing in the hypotheses) and from the fact that the addition preserves the domain (the addition of two natural numbers is a natural number).

5 Conclusion, perspectives

Our initial goal was to provide the ***B*** method with the expression of postconditions, with the following properties :

- The postcondition substitution should be as well-founded as the other basic substitutions.
- The postcondition must help establish the invariant of the machine.
- When generating the proof obligation for an operation containing an operation call, if the called operation has a postcondition, then its body is not needed to verify the proof obligation.
- The postcondition of an operation should be stronger than the postcondition of the refined operation.

All these properties have been established so far. Then, all works (see section 1) relying on postconditions will be able to base their foundations on the definition of postcondition we gave in this paper, namely :

- The translation of postconditions from ***event B*** to “classical” ***B***.
- The contract-based approach for ***B***, i.e. allowing to define ***B*** components by their contracts (pre- and postconditions for the operations).
- The embedding of additional assertions in the computer code generated from the ***B*** machines.
- The description of temporal properties of operations by using the postconditions to generate temporal formulas.

References

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Cle01] ClearSy. Event B reference manual, June 2001.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. Thoughts about the implementation of the duration calculus with coq. In *4th International Workshop on the Implementation of Logics*, volume Technical report ULCS-03-018. University of Liverpool, september 2003. http://www.csc.liv.ac.uk/research/techreports/.

[Mar97] Georges Mariano. *Évaluation de logiciels critiques développés par la méthode B : une approche quantitative*. Thèse de doctorat, Universitée de Valenciennes et du Hainaut-Cambrésis, Dec 1997.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.