Interpreting the B-Method in the Refinement Calculus

Yann Rouzaud

LSR-IMAG, Grenoble, France
Laboratoire Logiciels Systèmes Réseaux - Institut IMAG (CNRS - INPG - UJF)
BP 72, F-38402 Saint-Martin d'Hères Cedex
Tel +33 4 76827219 - Fax +33 4 76827287
Yann.Rouzaud@imag.fr
"Stream: Foundations and Methodology"
"Mini-Track: B-Method"

Abstract. In this paper, we study the B-Method in the light of the theory of refinement calculus. It allows us to explain the proof obligations for a refinement component in terms of standard data refinement. A second result is an improvement of the architectural condition of [PR98], ensuring global correctness of a B software system using the SEES primitive.

1 Introduction

The B-Method ([Abr96]) is a methodology for formal software development. It has industrial tools (B-toolkit, Atelier B) and has been successfully applied to the MTOR system, equipping the new line of subway in Paris ([Met]).

It is well known that the refinement theory of the B-Method is based on weakest precondition semantics ([Dij76]). However, curiously, no attempt has be made to establish a formal link between the theory of B, as exposed in the B-Book, and the now standard theory of refinement calculus. The primary goal of this paper is a study of this relationship.

Refinement calculus was first investigated by Back ([Ba78]), and further independently rediscovered by Morgan ([Mo88]) and Morris ([Mor87]). A good starting point to the subject is [BW98]. Its aim is to formalize the development of programs by stepwise refinement ([Wi71]). By removing the healthiness conditions of [Dij76] (which are necessary for an executable program), a continuum is obtained between specifications and programs. Indeed, specifications are programs (and programs are specifications). The theory is now mature; potential applications in the B-Method are discussed in the conclusion of this paper.

Our paper is organized as follows. Section 2 introduces the refinement calculus, defining the notions of predicate transformers and refinement, and the standard notion of data refinement. Section 3 presents another notion of data refinement. It is called data refinement through an invariant. It is this notion which is used in the B-Method, for the proof obligations of a refinement component. The relationship between the two notions of data refinement is established.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 411–430, 1999.

Section 4 describes the primitive language of substitutions of the B-Method. Substitutions are interpreted in terms of predicate transformers. Section 5 describes the notion of refinement in the B-Method, and relates it to standard data refinement. This section ends up with an example of valid data refinement, which is not expressible within the B-Method. Section 6 begins with an example showing that the SEES primitive of the B-Method can break the global correctness of a B-software. Using our analysis of the B-Method in terms of the refinement calculus, we exhibit an architectural condition ensuring global correctness. This analysis corrects and extends the work done in [PR98]. Finally, Section 7 concludes this work, by giving some directions of future work.

2 Predicate Transformers

Refinement calculus can be modelized using set transformers, or, equivalently, using predicate transformers. We choose the second model, because it explicitly deals with variables, which will be useful when dealing with common variables. This section is taken from [Wri94] and [BW90]. It reminds the main results and fixes the notation for the sequel.

2.1 States, Predicates, Commands, and Refinement

In programs, each variable x is associated with a set of values D_x. For any set of variables v, a state is a function mapping every x in v to some value in D_x, and the state space Σ_v is the set of states on v.

A predicate on v is a function from Σ_v to $Bool$, where $Bool = \{ff, tt\}$ is the ordered (with the order $ff < tt$) set of boolean values. $Pred_v$, the predicate space on v, is the complete boolean lattice of predicates on v, with the implication order $\leq$: $P \leq Q$ holds iff $P \Rightarrow Q$ holds universally (*false* is the bottom element and *true* is the top element).

A predicate on v can be extended to a predicate on v, w by adding new variables w. In this paper, we do not use explicit notation for this change of view. Substitutions $[d/v]$ in Σ_v are extended to $Pred_v$ in the usual way. If $Q \in Pred_{u,v}$, the renaming of v by w in Q, $Q[w/v]$, denotes the equivalent predicate in $Pred_{u,w}$.

$$\bigwedge_{i \in I} Q_i = \bigwedge \{Q_i \cdot i \in I\} \quad \text{and} \quad \bigvee_{i \in I} Q_i = \bigvee \{Q_i \cdot i \in I\}$$

are respectively the meet and the join of the family $(Q_i)_{i \in I}$. Quantified predicates are defined as follows:

$$\forall v \cdot P = \bigwedge_{d \in D_v} P[d/v] \quad \text{and} \quad \exists v \cdot P = \bigvee_{d \in D_v} P[d/v].$$

A predicate transformer is a monotonic function from predicates to predicates. $Mtran_{u \to v}$ is the complete lattice of predicate transformers from $Pred_v$ to $Pred_u$ ([Wri94] denotes it $Mtran_{u \leftarrow v}$). A predicate transformer $S \in Mtran_{u \to v}$ is identified with a command which executes from an initial state in Σ_u and, if it terminates, which gives a state in Σ_v (this is the weakest precondition approach,

due to [Dij76]). In this paper, we shall indifferently use the terms command and predicate transformer.

For commands $S \in \mathit{Mtran}_{u\to v}$ and $T \in \mathit{Mtran}_{v\to w}$, sequential composition $S;T \in \mathit{Mtran}_{u\to w}$ is such that $(S;T)(Q) = S(T(Q))$.

For each lattice $\mathit{Mtran}_{u\to v}$, commands *magic* and *abort* are top and bottom elements: $\mathit{magic}(Q) = \mathit{true}$ and $\mathit{abort}(Q) = \mathit{false}$. For $\mathit{Mtran}_{u\to u}$, *skip* is the identity command: $\mathit{skip}(Q) = Q$.

For two commands $S, T \in \mathit{Mtran}_{u\to v}$, the refinement order $S \leq T$ holds iff $S(Q) \leq T(Q)$ holds for all $Q \in \mathit{Pred}_v$.

2.2 Properties of Commands

Let $S \in \mathit{Mtran}_{u\to v}$ be a command, P, Q be predicates on v, and $(Q_i)_{i\in I}$ be a non-empty family of predicates on v. We have:

1. monotonicity: $P \leq Q \Rightarrow S(P) \leq S(Q)$
2. and-distributivity: $S(\bigwedge_{i\in I} Q_i) \leq \bigwedge_{i\in I} S(Q_i)$
3. or-distributivity: $\bigvee_{i\in I} S(Q_i) \leq S(\bigvee_{i\in I} Q_i)$

The following definitions are standard:

1. S is non-miraculous if $S(\mathit{false}) = \mathit{false}$
2. S is always terminating if $S(\mathit{true}) = \mathit{true}$
3. S is conjunctive if $S(\bigwedge_{i\in I} Q_i) = \bigwedge_{i\in I} S(Q_i)$
4. S is disjunctive if $S(\bigvee_{i\in I} Q_i) = \bigvee_{i\in I} S(Q_i)$

For instance, substitutions in the B-Method are conjunctive commands. See [BW92] for a study of sublanguages and their relationship.

2.3 Adjoint Commands

Let $S \in \mathit{Mtran}_{u\to v}$ be an always terminating disjunctive command. Then there exists a unique always terminating conjunctive command $S^{\mathtt{r}} \in \mathit{Mtran}_{v\to u}$ such that $S; S^{\mathtt{r}} \leq \mathit{skip}$ and $\mathit{skip} \leq S^{\mathtt{r}}; S$. $S^{\mathtt{r}}$ is the right adjoint of S. Dually, for each always terminating conjunctive command $S \in \mathit{Mtran}_{u\to v}$, there exists a unique always terminating disjunctive command $S^{\mathtt{l}} \in \mathit{Mtran}_{v\to u}$ such that $S^{\mathtt{l}}; S \leq \mathit{skip}$ and $\mathit{skip} \leq S; S^{\mathtt{l}}$. We have:

$$S^{\mathtt{r}}(P) = \bigvee\{Q \cdot S(Q) \leq P\} \quad \text{and} \quad S^{\mathtt{l}}(P) = \bigwedge\{Q \cdot P \leq S(Q)\}$$

Let R be a relation $R(u, v)$ and let α, β be commands such that $\forall Q$ *on* v, $\alpha(Q) = \exists v \cdot (R \wedge Q)$ and $\forall Q'$ *on* u, $\beta(Q') = \forall u \cdot (R \Rightarrow Q')$. Then we have: $\alpha^{\mathtt{r}} = \beta$ and $\beta^{\mathtt{l}} = \alpha$.

2.4 Data Refinement

Data refinement is like refinement, with a supplementary change of variable space. The following definition is now standard; [GM93] (where it is called cosimulation) proved that it is a complete method when refining modules. The identifiers a, c, g stand respectively for the set of abstract, concrete and global variables (supposed distinct).

Let $S \in \mathit{Mtran}_{a,g \to a,g}$, $T \in \mathit{Mtran}_{c,g \to c,g}$, and $\alpha \in \mathit{Mtran}_{c,g \to a,g}$ be commands, then $S \leq_\alpha T$ (S is refined by T through α, see Fig. 1) iff $\alpha; S \leq T; \alpha$.

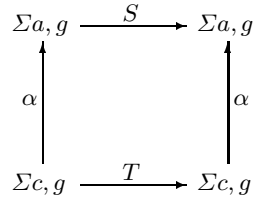

Fig. 1. $S \leq_\alpha T$ iff $\alpha; S \leq T; \alpha$.

Various command constructors are monotonic through $\leq_\alpha$. In particular,

if $S_1 \leq_\alpha T_1$ and $S_2 \leq_\alpha T_2$ then $S_1; S_2 \leq_\alpha T_1; T_2$.

Another property is transitivity: if $S_1 \leq_\alpha S_2$ and $S_2 \leq_\beta S_3$ then $S_1 \leq_{\beta;\alpha} S_3$.
An important case of data refinement is *forward data refinement*. It occurs when $\alpha(Q) = \exists a \cdot (R \wedge Q)$, where R is a predicate on a, c, g, so we have:

$$S \leq_\alpha T \text{ iff } \forall c, g \cdot (\exists a \cdot (R \wedge S(Q)) \Rightarrow T(\exists a \cdot (R \wedge Q))) \text{ for all } Q \in \mathit{Pred}_{a,g}$$

that is: $\forall a, c, g \cdot (R \wedge S(Q) \Rightarrow T(\exists a \cdot (R \wedge Q)))$. In the case of forward data refinement, the four conditions of Fig. 2 are equivalent (they correspond to previous definitions of data refinement).

(1)	$\alpha; S \leq T; \alpha$
(2)	$\alpha; S; \alpha^{\mathrm{r}} \leq T$
(3)	$S; \alpha^{\mathrm{r}} \leq \alpha^{\mathrm{r}}; T$
(4)	$S \leq \alpha^{\mathrm{r}}; T; \alpha$

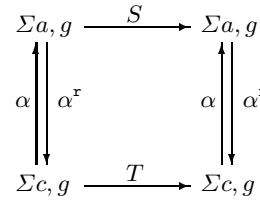

Fig. 2. Equivalent definitions of forward data refinement.

Forward data refinement is incomplete, but it suffices for most practical cases. As we shall see, refinement in the B-Method is based on forward data refinement, and its exposure in the B-Book uses the third condition of Fig. 2.

We end this section with the notion of *abstraction lattice*. Let R be a relation $R(a, c, g)$ such that $\forall c, g \cdot \exists a \cdot R$. The set of commands α such that:

$$\forall a \cdot (R \Rightarrow Q) \leq \alpha(Q) \leq \exists a \cdot (R \wedge Q)$$

is a complete non-empty lattice, called the abstraction lattice for (a, c, R). It is a singleton iff R is functional.

Now S is data refined by T through R, $S \leq_R T$, if there exists a command α in the abstraction lattice such that $S \leq_\alpha T$. If $\alpha(Q) = \forall a \cdot (R \Rightarrow Q)$, data refinement is called backward data refinement. An example of backward data refinement is given in Section 5.

3 Another Formulation of Data Refinement

In this section, we consider another formulation of data refinement, due to [GP85]. We call it *data refinement through an invariant*. It is also used in the B-Method. The equivalence of forward data refinement and data refinement through an invariant is proved in [CU89], for conjunctive commands with no common variables. We restate this proof, dealing with non-conjunctive commands, and allowing common variables. Forward data refinement is the "right" definition and data refinement through an invariant is an "operational" one, more adequate for use in theorem provers.

3.1 Extension of a Command

Like the B-Method, [GP85] and [CU89] deal with formulae transformers, which can be seen as "generic" predicate transformers, able to work with any variable space. We reformulate their works with commands, so we must be able to "extend" the set of variables involved in a command. This operation looks like the "embed" operator of [BB98].

First, we note that if Q is a predicate on v, w, we can find two families $(w_i)_{i \in I}$ and $(Q_i)_{i \in I}$, such that:

1. $\bigcup_{i \in I} \{w_i\} \subseteq D_w$,
2. each Q_i is a predicate on v (it may be *false*),
3. Q can be written as $Q = \bigvee_{i \in I} (Q_i \wedge w = w_i)$.

Definition 1. Let $S \in \mathit{Mtran}_{u \to v}$ be a command and w be a set of variables, distinct from v. $S_w \in \mathit{Mtran}_{u,w \to v,w}$ is called the extension of S to the context w, and is defined as follows: $S_w(Q) = \bigvee_{i \in I} (S(Q_i) \wedge w = w_i)$.

Property 1. Let $S \in \mathit{Mtran}_{u \to v}$ be a command, w be a set of variables, distinct from v, P be a predicate on v, w and Q be a predicate on w. Then:

1. $S(\forall w \cdot P) \leq \forall w . S_w(P)$.
2. $S(\forall w \cdot P) = \forall w . S_w(P)$ if S is conjunctive.
3. $S_w(Q) = S(\mathit{true}) \wedge Q$.
4. $S_w(P \vee Q) = S_w(P) \vee (S(\mathit{true}) \wedge Q)$.

Proof.

1. We write $P = \bigvee_{i \in D_w} (P_i \wedge w = i)$.
$$\begin{aligned}
S(\forall w \cdot P) &= S(\textstyle\bigwedge_{d \in D_w} P[d/w]) && \text{definition of quantifiers}\\
&\leq \textstyle\bigwedge_{d \in D_w} S(P[d/w]) && \text{and-distributivity}\\
&= \textstyle\bigwedge_{d \in D_w} S((\bigvee_{i \in D_w} (P_i \wedge w = i))[d/w]) && \text{definition of } P\\
&= \textstyle\bigwedge_{d \in D_w} S(P_d) && \text{calculus}\\
&= \textstyle\bigwedge_{d \in D_w} (\bigvee_{i \in D_w} (S(P_i) \wedge w = i))[d/w]\\
&= \textstyle\bigwedge_{d \in D_w} S_w(P)[d/w] && \text{definition of } S_w\\
&= \forall w \cdot S_w(P) && \text{definition of quantifiers}
\end{aligned}$$
2. Since S is conjunctive, the inequality in the demonstration above becomes an equality.
3. Q can be written as $Q = (\bigvee_{j \in J} w = w_j) = \bigvee_{j \in J} (\mathit{true} \wedge w = w_j)$.
$$\begin{aligned}
S_w(Q) &= \textstyle\bigvee_{j \in J} (S(\mathit{true}) \wedge w = w_j) && \text{definition of } Q\\
&= S(\mathit{true}) \wedge \textstyle\bigvee_{j \in J} w = w_j && \text{factorisation}\\
&= S(\mathit{true}) \wedge Q && \text{definition of } Q
\end{aligned}$$
4. $P = \bigvee_{i \in I} (P_i \wedge w = w_i)$ and $Q = (\bigvee_{j \in J} w = w_j) = \bigvee_{j \in J} (\mathit{true} \wedge w = w_j)$. So $P \vee Q = \bigvee_{k \in K} (R_k \wedge w = w_k)$, where:
   (a) $K = I + J = \{(0, i) \cdot i \in I\} \cup \{(1, j) \cdot j \in J\}$
   i.e. K is the disjoint union of I and J;
   (b) if $k = (0, i)$ then $w_k = w_i$ and $R_k = P_i$;
   (c) if $k = (1, j)$ then $w_k = w_j$ and $R_k = \mathit{true}$.
$$\begin{aligned}
S_w(P \vee Q) &= \textstyle\bigvee_{k \in K} (S(R_k) \wedge w = w_k)\\
&= \textstyle\bigvee_{i \in I} (S(P_i) \wedge w = w_i) \vee \bigvee_{j \in J} (S(\mathit{true}) \wedge w = w_j)\\
&= S_w(P) \vee (S(\mathit{true}) \wedge Q)
\end{aligned}$$

3.2 Data Refinement through an Invariant

Definition 2. Let a, c be disjoint sets of variables, $S \in \mathit{Mtran}_{a \to a}$ and $T \in \mathit{Mtran}_{c \to c}$ be commands and I be a predicate $I(a, c)$. We denote by $S \leq_I T$ the fact that S is data refined by T through I. Then
$$S \leq_I T \;\text{ iff }\; I \wedge S(\mathit{true}) \Rightarrow T_a(\neg S_c(\neg I)).$$

This definition must be adapted when common variables occur:

Definition 3. Let a, c, g be disjoint sets of variables, $S \in \mathit{Mtran}_{a,g \to a,g}$ and $T \in \mathit{Mtran}_{c,g \to c,g}$ be commands, and I be a predicate $I(a, c, g)$.
$$S \leq_I T \;\text{ iff }\; I \wedge g = g' \wedge S(\mathit{true}) \Rightarrow T'_{a,g}(\neg S_{c,g'}(\neg(I \wedge g = g'))$$
where g' are fresh variables (i.e. new and different from a, c, g) and $T' \in \mathit{Mtran}_{c,g' \to c,g'}$ is such that $T'(Q) = T(Q[g/g'])[g'/g]$.

For instance, let $D_g = D_{g'} = \mathbb{N}$, $S(Q) = Q[g + 1/g]$, $T(Q) = Q[g + 1/g]$, and $I = \mathit{true}$. For simplicity, we assume that sets a and c are empty.

1. $S_{c,g'}(\neg(I \wedge g = g')) = S_{c,g'}(\bigvee_{d \in D_{g'}}(g \neq d \wedge g' = d))$
$= \bigvee_{d \in D_{g'}}(S(g \neq d) \wedge g' = d)$
$= \bigvee_{d \in D_{g'}}(g + 1 \neq d \wedge g' = d) \;=\; (g + 1 \neq g')$.
2. So $\neg S_{c,g'}(\neg(I \wedge g = g')) = (g + 1 = g') = \bigvee_{d \in D_g}(d + 1 = g' \wedge g = d)$.
3. $T'_{a,g}(\neg S_{c,g'}(\neg(I \wedge g = g')) = \bigvee_{d \in D_g}(T'(d{+}1{=}g') \wedge g{=}d)$
$= \bigvee_{d \in D_g}(T(d{+}1{=}g)[g'/g] \wedge g{=}d)$
$= \bigvee_{d \in D_g}((d{+}1{=}g+1)[g'/g] \wedge g{=}d)$
$= \bigvee_{d \in D_g}(d{+}1{=}g'+1 \wedge g{=}d){=}(g{+}1{=}g'{+}1)$

3.3 Equivalence of Data Refinement Definitions

Property 2. Let u, v be disjoint sets of variables, $S \in \mathit{Mtran}_{u \to u}$ be a command, Q_1 be a predicate on u, v and Q_2 be a predicate on v. Then:

$$\forall u \cdot S(\forall v \cdot (Q_1 \Rightarrow Q_2)) \leq \forall u, v \cdot (\neg S_v(\neg Q_1) \Rightarrow Q_2)$$

Proof.

$$\begin{aligned}
\forall u \cdot S(\forall v \cdot (Q_1 \Rightarrow Q_2)) &\leq \forall u, v \cdot S_v(Q_1 \Rightarrow Q_2) && \text{property 1.1}\\
&= \forall u, v \cdot S_v(\neg Q_1 \vee Q_2)\\
&= \forall u, v \cdot (S_v(\neg Q_1) \vee (S(\mathit{true}) \wedge Q_2)) && \text{property 1.4}\\
&= \forall u, v \cdot (\neg S_v(\neg Q_1) \Rightarrow S(\mathit{true}) \wedge Q_2)\\
&\leq \forall u, v \cdot (\neg S_v(\neg Q_1) \Rightarrow Q_2)
\end{aligned}$$

Property 3. Let a, c be disjoint sets of variables, $S \in \mathit{Mtran}_{a \to a}$ and $T \in \mathit{Mtran}_{c \to c}$ be commands, I be a predicate on a, c, and $\alpha \in \mathit{Mtran}_{c \to a}$ be $\alpha(Q) = \exists a \cdot (I \wedge Q)$. Then $S \leq_I T \Rightarrow S \leq_\alpha T$.

Proof. Let Q be a predicate on c.

1. $\forall a, c \cdot (I \wedge S(\forall c \cdot (I \Rightarrow Q))) \leq \forall a, c \cdot S(\forall c \cdot (I \Rightarrow Q))$ predicate calculus
$\leq \forall a, c \cdot (\neg S_c(\neg I) \Rightarrow Q)$ property 2
$\leq \forall a, c \cdot (T_a(\neg S_c(\neg I)) \Rightarrow T_a(Q))$ monotonicity
2. Assume $S \leq_I T$, that is $\forall a, c \cdot (I \wedge S(\mathit{true}) \Rightarrow T_a(\neg S_c(\neg I)))$. So we have to prove $S \leq_\alpha T$. Because α has the right form, we may use the second of the four equivalent formulations of forward data refinement: $\alpha; S; \alpha^{\mathrm{r}} \leq T$, that is: $\forall a, c \cdot (I \wedge S(\forall c \cdot (I \Rightarrow Q)) \Rightarrow T(Q))$.
3. Given a, c, assume $I \wedge S(\forall c \cdot (I \Rightarrow Q))$.
4. $\forall c \cdot (I \Rightarrow Q) \Rightarrow \mathit{true}$, so $S(\mathit{true})$ holds by monotonicity, so does $T_a(\neg S_c(\neg I))$.
5. Using step 1, $T_a(Q)$ holds.
6. $T_a(Q) = T(Q)$, because Q is a predicate on c.

Property 4. Let a, c be disjoint sets of variables, $S \in \mathit{Mtran}_{a \to a}$ and $T \in \mathit{Mtran}_{c \to c}$ be commands, I be a predicate on a, c, and $\alpha \in \mathit{Mtran}_{c \to a}$ be $\alpha(Q) = \exists a \cdot (I \wedge Q)$. If S is conjunctive then $S \leq_\alpha T \Rightarrow S \leq_I T$.

Proof (the idea of P_1 and P_2 comes from [CU89]). Assume $S \leq_\alpha T$: for any a, c such that $I \wedge S(true)$, we have to prove $T_a(\neg S_c(\neg I))$. Let $a_0 \in D_a$, $P_1 = \neg S_c(\neg I)[a_0/a]$ and $P_2 = \forall c \cdot (I \Rightarrow P_1)$.

1. $S(P_2) = \forall c \cdot S_c(I \Rightarrow P_1)$ property 1.2
   $= \forall c \cdot S_c(\neg I \vee P_1)$ predicate calculus
   $= \forall c \cdot (S_c(\neg I) \vee (S(true) \wedge P_1))$ property 1.4
2. Using predicate calculus:
   $S(P_2)[a_0/a] = (\forall c \cdot (S_c(\neg I) \vee (S(true) \wedge P_1)))[a_0/a]$
   $= \forall c \cdot (S_c(\neg I)[a_0/a] \vee (S(true)[a_0/a] \wedge P_1[a_0/a]))$
   $= \forall c \cdot (S_c(\neg I)[a_0/a] \vee (S(true)[a_0/a] \wedge \neg S_c(\neg I)[a_0/a]))$
   $= \forall c \cdot (S_c(\neg I)[a_0/a] \vee S(true)[a_0/a])$
   $= (\forall c \cdot (S_c(\neg I) \vee S(true)))[a_0/a]$
   So $S(P_2) = \forall c \cdot (S_c(\neg I) \vee S(true))$.
3. We have $S(true) \Rightarrow S(P_2)$, so $I \wedge S(true) \Rightarrow I \wedge S(P_2)$. Hence $T(\exists a \cdot (I \wedge P_2))$ because $S \leq_\alpha T$.
4. Given any c, let a be such that $I \wedge P_2$ holds; we have $I \wedge P_2 = I \wedge \forall c \cdot (I \Rightarrow P_1)$, so $I \Rightarrow P_1$, so P_1 holds. Hence $\forall c \cdot (\exists a \cdot (I \wedge P_2) \Rightarrow \neg S_c(\neg I)[a_o/a])$.
5. By monotonicity, $\forall c \cdot (T(\exists a \cdot (I \wedge P_2)) \Rightarrow T(\neg S_c(\neg I)[a_o/a]))$.
6. We have: $T(\exists a \cdot (I \wedge P_2)) = T(\exists a \cdot (I \wedge P_2))[a_0/a]$
   and $T(\neg S_c(\neg I)[a_0/a]) = T_a(\neg S_c(\neg I))[a_0/a]$.
   So $T(\exists a \cdot (I \wedge P_2))[a_0/a] \Rightarrow T_a(\neg S_c(\neg I))[a_0/a]$,
   hence: $T(\exists a \cdot (I \wedge P_2)) \Rightarrow T_a(\neg S_c(\neg I))$.
7. Finally, $T_a(\neg S_c(\neg I))$ holds.

Property 4 does not hold for non-conjunctive commands. For example, let $D_a = D_c = \mathbb{N}$, $S(Q) = Q[1/a] \vee Q[2/a]$, $T(Q) = Q[1/c] \vee Q[2/c]$, and $I = (a = c)$. Then $\alpha(Q) = \exists a \cdot (a = c \wedge Q) = Q[c/a]$ and $S \leq_\alpha T$ holds.
But $T_a(\neg S_c(\neg I)) = T_a(\neg((1 \neq c) \vee (2 \neq c))) = T_a(false) = false$.

Theorem 1. Let a, c, g be disjoint sets of variables, $S \in Mtran_{a,g \to a,g}$ and $T \in Mtran_{c,g \to c,g}$ be commands, I be a predicate on a, c, g, and $\alpha \in Mtran_{c,g \to a,g}$ be $\alpha(Q) = \exists a \cdot (I \wedge Q)$. Then:

(1)	$S \leq_I T \Rightarrow S \leq_\alpha T$
(2)	$S \leq_I T \Leftrightarrow S \leq_\alpha T$ if S is conjunctive

Proof. Let g' be fresh variables, $T' \in Mtran_{c,g' \to c,g'}$ be a command such that $T'(Q) = T(Q[g/g'])[g'/g]$, and $\beta \in Mtran_{c,g' \to c,g}$ be $\beta(Q) = \exists g \cdot (g = g' \wedge Q) = Q[g'/g]$. So $\beta; \alpha \in Mtran_{c,g' \to a,g}$.

1. We have : $T'; \beta(Q) = T'(Q[g'/g]) = T(Q[g'/g][g/g'])[g'/g] = T(Q)[g'/g] = \beta; T(Q)$. So $\beta; T = T'; \beta$ and $T \leq_\beta T'$.
2. By transitivity: $S \leq_{\beta;\alpha} T' \Leftrightarrow S \leq_\alpha T$
3. But $\beta; \alpha(Q) = \exists g \cdot (g = g' \wedge \exists a \cdot (I \wedge Q)) = \exists a, g \cdot (I \wedge g = g' \wedge Q)$.
4. Now the results hold by applying properties 3 and 4 with S, T', $I \wedge g = g'$ and $\beta; \alpha$ (the variable spaces are disjoint).

4 The Language of Substitutions

In this section, we present the substitutions of the B-Method, and we interpret them as commands.

The primitive language of substitutions is given in Fig. 3, where x is a list of variables, E is a list of expressions, P is a formula, S and T are substitutions. Substitutions have a formula transformer semantics. Let Q be a formula and S be a substitution: $[S]Q$ is the weakest precondition establishing Q. For the unbounded choice substitution, we assume that Q has no free occurrence of x (otherwise, a renaming can be done). The B-Book, page 287, establishes that substitutions are monotonic (through implication) and conjunctive.

S	$[S]Q$	name of substitution	$U(S)$	$M(S)$
skip	Q	do nothing	$\emptyset$	$\emptyset$
$x := E$	$Q[E/x]$	simple	$Var(E)$	$\{x\}$
$P \mid S$	$P \wedge [S]Q$	precondition	$Var(P) \cup U(S)$	$M(S)$
$S \parallel T$	$[S]Q \wedge [T]Q$	bounded choice	$U(S) \cup U(T)$	$M(S) \cup M(T)$
$P \Longrightarrow S$	$P \Rightarrow [S]Q$	guard	$Var(P) \cup U(S)$	$M(S)$
$@x \cdot S$	$\forall x \cdot [S]Q$	unbounded choice	$U(S) - \{x\}$	$M(S) - \{x\}$

Fig. 3. Definition of substitutions; used and modified variables.

Let $Var(E)$ and $Var(P)$ be the sets of free variables of an expression E and of a formula P. In Fig. 3, $U(S)$ is the set of used variables in the substitution S and $M(S)$ is the set of modified variables.

Now, let $Var(S) = U(S) \cup M(S)$. First, we note that the notation $[S]Q$ can be reused for a predicate $Q \in Pred_{Var(S)}$, because the involved operators in formulae have homolog ones in predicates. So a substitution S is obviously interpreted as a predicate transformer $[\![S]\!]$ in $Mtran_{Var(S) \to Var(S)}$: $[\![S]\!](Q) = [S]Q$ for all $Q \in Pred_{Var(S)}$.

In practice, there are many other substitutions involved in B-components (machine, refinement, implementation) with another (verbose) syntax. However, all these substitutions are reducible to the primitive language of substitutions. For instance, the substitution PRE P THEN S END is equivalent to $P \mid S$. Moreover, these substitutions are used under the context of the variables v of the B-component. For such substitutions S, we must consider their extension $[\![S]\!]_w$, where $w = v - Var(S)$.

Some substitution constructors (for instance, the parallel operator "$\|$" and the sequence operator ";") are indirectly defined, using the normalized form theorem[1]: any substitution S can be written as:

[1] Its demonstration in the B-Book is flawed, but it can be easily repaired, by explicitly dealing with the set of involved variables.

$$S = \mathsf{trm}(S) \mid @v' \cdot (\mathsf{prd}_v(S) \Longrightarrow v := v')$$

where $v = Var(S)$, v' are fresh variables, $\mathsf{trm}(S) = [S](v = v)$ (it is $[\![S]\!](true)$), and $\mathsf{prd}_v(S) = \neg[S](v' \neq v)$. So, it suffices to define a substitution by giving its trm and prd, with the requirement: $\neg\ \mathsf{trm}(S) \Rightarrow \mathsf{prd}_v(S)$, where S is the newly defined substitution and $v = Var(S)$ (see [Du97, Du99]; see also property 6.4.1, page 297 of the B-Book). For instance, for the parallel operator[2], we have: $\mathsf{trm}(S\|T) = \mathsf{trm}(S) \wedge \mathsf{trm}(T)$, and $\mathsf{prd}_{v,w}(S\|T) = (\mathsf{trm}(S) \wedge \mathsf{trm}(T) \Rightarrow \mathsf{prd}_v(S) \wedge \mathsf{prd}_w(T))$, where $v = Var(S)$ and $w = Var(T)$ are disjoint sets of variables.

5 Refinement in the B-Method

In this section, we describe the notion of refinement, as defined in the B-Book, and we relate it to the results of Section 3.

5.1 Refinement Component and Proof Obligations.

A refinement component is defined as a differential to be added to a component. A refinement component can have proper variables which are linked to variables of the refined component by a *gluing invariant*. Moreover, refined operations must be stated on the new variables.

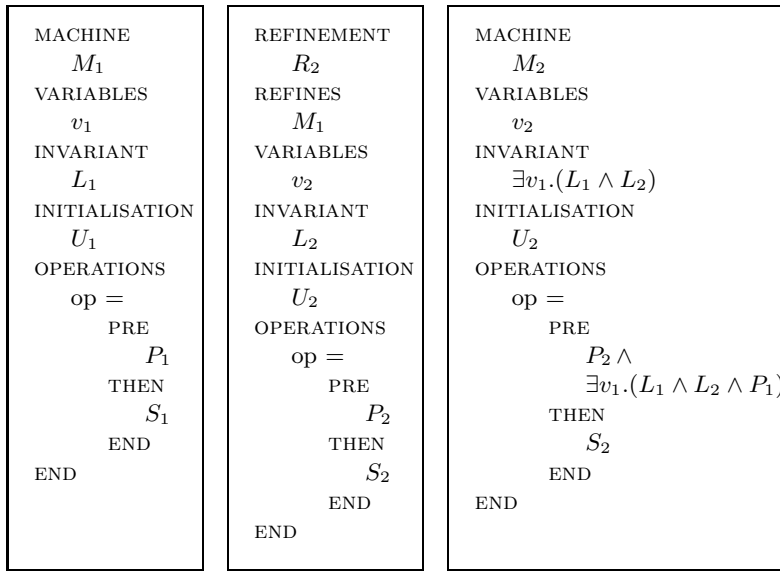

Fig. 4. Refinement R_2 of M_1, seen as an independent machine M_2.

[2] We use the definition of [Du97].

The proof obligations for machine M_1 of Fig. 4 are:

1. Initialisation: $[U_1]L_1$
2. Operation op: $L_1 \wedge P_1 \Rightarrow [S_1]L_1$

The proof obligations for refinement R_2 of Fig. 4 are, provided that there are no common variables (B-Book, p. 530):

1. Initialisation: $[U_2]\neg[U_1]\neg L_2$
2. Operation op: $L_1 \wedge L_2 \wedge P_1 \Rightarrow P_2 \wedge [S_2]\neg[S_1]\neg L_2$

In the most general case, there is a chain of refinements $M_1, R_2, \ldots, R_n$ to be considered. The proof obligation for an operation of R_n is, provided that M_1 and its refinements have no common variables:

$$L_1 \wedge L_2 \wedge \ldots \wedge L_n \wedge P_1 \wedge \ldots \wedge P_{n-1} \Rightarrow P_n \wedge [S_n]\neg[S_{n-1}]\neg L_n \ .$$

5.2 Refinement in B and Data Refinement

In the B-Book, the theory of refinement is presented in a set-transformer fashion. Then sufficient conditions of previous subsections are established. To do that, Abrial introduces a "substitution" W (the quotation marks come from the B-Book, pages 513 and 517): let a and c be the sets of abstract and concrete variables, supposed distinct, with corresponding sets of values D_a, D_c, and let R be a total relation from D_c to D_a (i.e $\forall c \exists a \cdot R(c,a)$); then $str(W)(q) = \overline{R[\overline{q}]}$, for all $q \subseteq D_c$ (in fact, Abrial defines W in a context of "external" variables, then drops them, page 519 ; we decided to directly present W without those variables).

W is not a true substitution, because initial and final state spaces are distinct. But it corresponds to a command $[\![W]\!] \in Mtran_{a\to c}$: $[\![W]\!](Q) = \forall c \cdot (R \Rightarrow Q)$ for all $Q \in Pred_c$. So, $[\![W]\!]$ is the right adjoint of the command $\alpha \in Mtran_{c\to a}$: $\alpha(Q) = \exists a \cdot (R \wedge Q)$, for all $Q \in Pred_a$. Hence, refinement in B is forward data refinement, and its presentation in the B-Book follows the third of the four equivalent formulations: $S \leq_\alpha T \ \Leftrightarrow \ S; \alpha^{\mathtt{r}} \leq \alpha^{\mathtt{r}}; T$.

Page 528, Property 11.2.5 shows that data refinement through an invariant implies forward data refinement. So, the proof obligations of the B-Method correspond to data refinement through an invariant. They are simpler, because it is assumed that the refined component is already proved (see Property 11.2.3, page 526 of the B-Book).

5.3 A Data Refinement Inexpressible in the B-Method

The example of Fig. 5 is taken from [GM93].

In machine M_1, variable I_2 determines how many times the operation *incr* is active (in [GM93], I_2 is called a prophecy variable). Machine M_1 is data refined by machine M_2, through the command α:

$$\alpha(Q) = \forall I_2 \cdot ((B_2 = \mathsf{true} \Rightarrow I_2 = 0) \wedge I_1 + I_2 \leq 10 \Rightarrow Q)$$

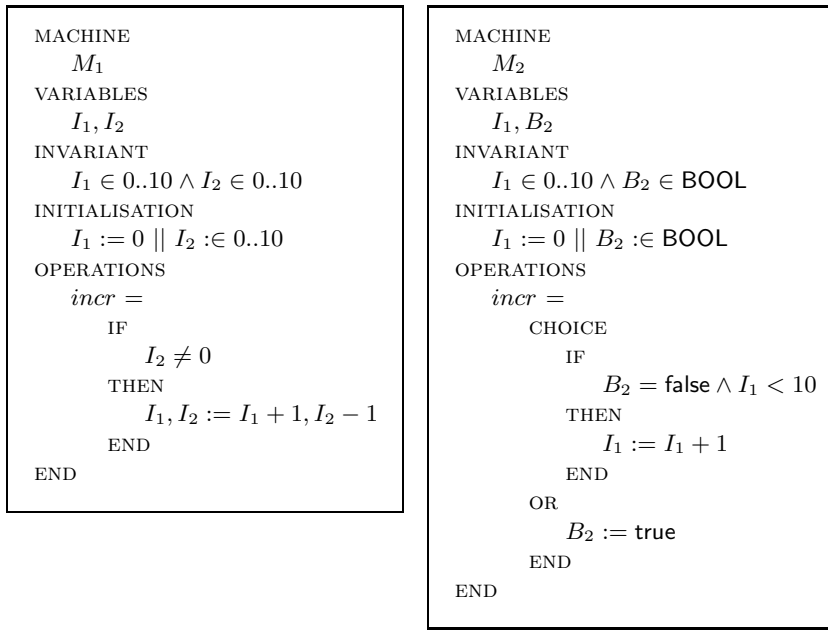

Fig. 5. A backward data refinement.

This is a backward data refinement ([Wri94]): the non-deterministic choice in the initialisation of I_2 in M_1 is made later in M_2 (it is done in the operation *incr*).

This data refinement cannot be established in the B-Method, where only forward data refinement is allowed.

6 The SEES Primitive

In this section, we exhibit an example of an incorrect B software system, where all components are locally correct. The problem relies upon a misuse of the composition primitives IMPORTS and SEES. We show that those primitives can cause aliasings between variables. Unsoundness of the construction can be determined at the command interpretation level. Then, we exhibit an architectural condition, ensuring global correctness.

First, the semantics of IMPORTS and SEES primitives of the B-Method are reminded :

- The IMPORTS primitive links implementations to abstract machines, and is used to build the state of a machine on to the state of imported machines. This primitive does not introduce sharing, so it allows to build a layered software.

– The SEES primitive allows the sharing of an abstract machine. This primitive can be used in a machine, a refinement or an implementation. Variables of a seen machine can only be consulted, and must not be modified by the seeing components. Notice that a SEES primitive must be preserved in a development: if a machine is seen at a given level, it must be seen in subsequent refinements.

In [PR98], we exhibited an architectural condition, ensuring global correctness of a B software system, when SEES occurrences are only in implementations. We extend this work, dealing with SEES occurrences at all levels.

6.1 An Example

The example given in Fig. 7 exhibits an architecture (Fig. 6) which is incorrect, in the sense that local proofs of correctness do not guarantee global correctness (this example was accepted by Atelier B [AtB], up to version 3.2: proof obligations discharged and architecture not rejected). Note that the conclusion is identical if machines A_2 and A_3 are seen by the implementation of A_1 instead of being imported.

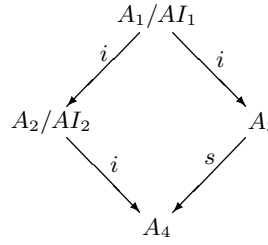

Fig. 6. Architecture of the example.

All machines and implementations are locally correct. But, the effective code of the operation Op_1 (Fig. 8) can be built by replacement, as defined for the meaning of the operation call (B-Book page 314-316 and 556), and extended for passing through a chain of refinements in [PR98]. In this case, the substitution reduces to rr := FALSE, what is obviously incorrect.

6.2 An Aliasing Problem

In this subsection, we go back to monotonicity through $\leq_\alpha$, as explained in Section 2. We show that the lack of global correctness of a B software system using SEES primitive occurs because monotonicity cannot be applied.

Let $C\langle X\rangle$ be a command, built by using X and some monotonic (through $\leq_\alpha$) command constructors, where X is a variable denoting an unknown command.

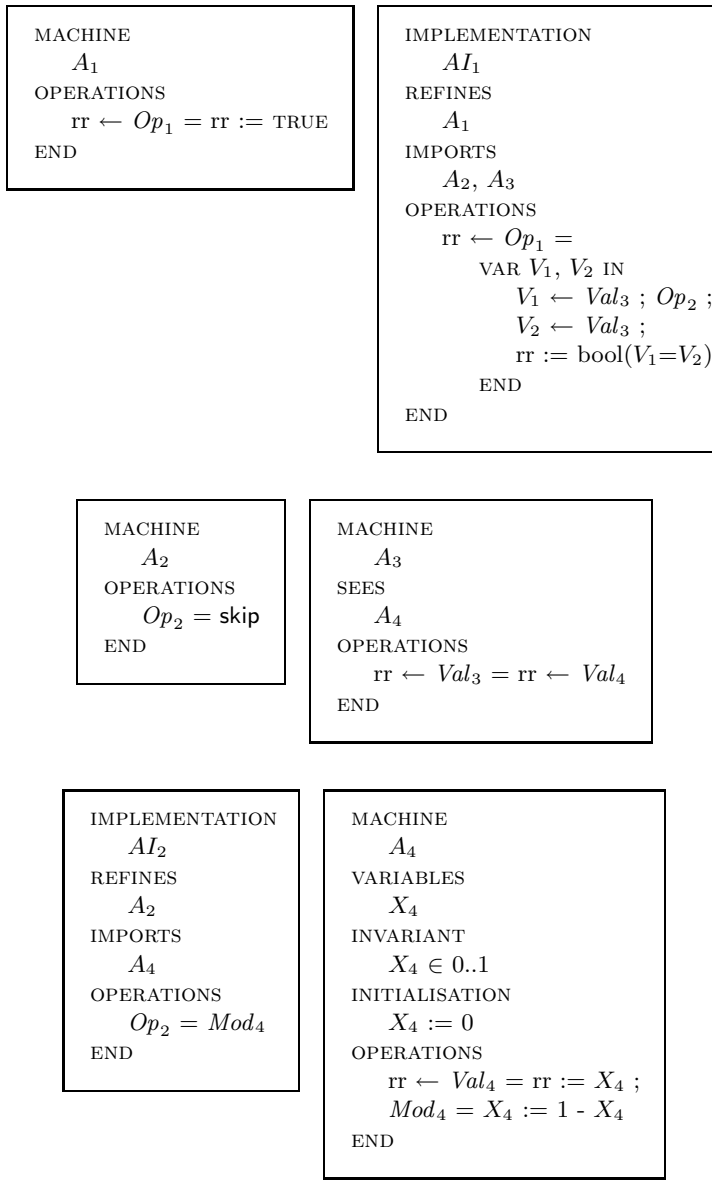

Fig. 7. A complete example

```
rr ← Op1 =
    VAR V1, V2 IN
        V1 := X4 ; X4 := 1-X4 ; V2 := X4 ;
        rr := bool(V1=V2)
    END
```

Fig. 8. Code of the Op_1 operation

We assume that $X \in \mathit{Mtran}_{a \to a}$ and that C ranges over variables g, distinct from a, so $C\langle X\rangle \in \mathit{Mtran}_{a,g \to a,g}$. Hence X must be extended to X_g to be used in C. For instance, $C\langle X\rangle = [1/g]; X_g$, so that $C\langle X\rangle(Q) = X_g(Q)[1/g]$.

Let $S \in \mathit{Mtran}_{a \to a}$, $T \in \mathit{Mtran}_{c \to c}$ and $\alpha \in \mathit{Mtran}_{c \to a}$ be commands, such that a, c, g are distinct and $S \leq_\alpha T$ holds. Obviously, $S_g \leq_{\alpha_g} T_g$, so, by monotonicity: $C\langle S\rangle \leq_{\alpha_g} C\langle T\rangle$.

Now, let $S \in \mathit{Mtran}_{a \to a}$, $T \in \mathit{Mtran}_{c,g \to c,g}$ and $\alpha \in \mathit{Mtran}_{c,g \to a}$ be commands, such that $S \leq_\alpha T$. For instance, $S = \mathit{skip}$, $T = [2/g]$, and $\alpha = [c/a]$. In this case, there is no refinement relation between the extended predicate transformers: $S_g \not\leq_{\alpha_g} T_g$, and monotonicity cannot be applied[3]. As counterexample, we have: $C\langle S\rangle = [1/g]; \mathit{skip} = [1/g]$, and $C\langle T\rangle = [1/g]; [2/g] = [2/g]$.

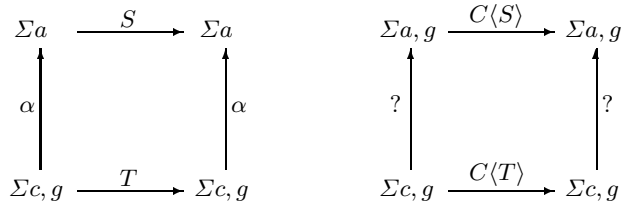

Fig. 9. An aliasing problem: concrete variable g occurs in the context $C\langle X\rangle$.

So the problem clearly comes from an aliasing between the variables of the using context of a command and the concrete variables of its refinement (see Fig.9).

We can now explain the example in the light of this analysis: in implementation AI_1, the operation Op_2 of machine A_2 is used in the context X_4, because AI_1 imports A_3, which sees A_4 (where X_4 is defined). In the implementation AI_2, the concrete variable is X_4, because AI_2 imports A_4; hence, an alias occurs.

Prohibiting this aliasing phenomenon by rejecting every aliasing architecture would be a too drastic solution. Such an architecture is globally correct if we can ensure that, when an aliasing phenomenon appears, the refined operation does not modify the global variables. In terms of commands: let $S \in \mathit{Mtran}_{a \to a}$,

[3] In [PR98], monotonicity was mistakenly applied.

$T \in Mtran_{c,g \to c,g}$ and $\alpha \in Mtran_{c,g \to a}$ be commands, such that $S \leq_\alpha T$. Let $C\langle X \rangle \in Mtran_{a,g \to a,g}$ be a command constructor.

If for all $Q \in Pred_g$ and for all $d \in D_g$, $T(Q[d/g]) = T(Q)[d/g]$,
then $S_g \leq_{\alpha_g} T_g$, so $C\langle S \rangle \leq_{\alpha_g} C\langle T \rangle$.

Applying this condition to the B-Method can be done either by a fine analysis of operations, or by simply ensuring, by architectural restrictions, that the refined operations *cannot* modify global variables. In the following, we investigate the latter approach.

6.3 Dependency Relations

Definition 4.

1. C_1 *s* M_2 iff the component C_1 (a machine, a refinement or an implementation) sees the machine M_2.
2. M_1 *sees* M_2 iff the implementation of M_1 sees the machine M_2.
3. M_1 *imports* M_2 iff the implementation of M_1 imports the machine M_2.
4. M_1 *uses* M_2 iff the implementation of M_1 sees or imports M_2:
   $uses = sees \cup imports$.
5. M_1 *depends_on* M_2 iff the implementation of M_1 is built by using M_2:
   $depends\_on = uses^+$.
6. M_1 *can_consult* M_2 iff the implementation of M_1 can consult the variables of the code of M_2:
   $can\_consult = (uses^*; sees)$.
7. M_1 *can_alter* M_2 iff the implementation of M_1 can modify the variables of the code of M_2:
   $can\_alter = (uses^*; imports)$.

Relational notation is the one of the B-Method: transitive closure ($^+$), reflexive and transitive closure (*) and composition (;).

6.4 Framework Hypotheses

Our analysis is based on the following assumptions, stated in the B-Book:

1. The dependency graph has no cycle: $depends\_on \cap id = \emptyset$, where id is the identity relation.
2. If a machine M sees a machine N, refinements of M must see N: $s \subseteq sees$.
3. A machine is imported only once: $imports^{-1}$ is a (partial) function.
4. An implementation cannot see and import the same machine:
   $sees \cap imports = \emptyset$.
5. Variables of two distinct components (at clause VARIABLES) are distinct. So common variables in components can only occur when a SEES primitive is used. Moreover, in a refinement chain, a variable cannot disappear, then reappear at a lower level.
6. If a component C sees a machine N, variables of N cannot be referenced in the invariant of C, and only consulting operations of N can be called in C.

6.5 Ensuring Monotonicity

Let M_1 be an abstract machine, and let I_1 be its implementation.

1. Assume I_1 imports a machine M_2. Since a machine can be imported only once, variables of M_2 cannot be modified elsewhere: this case is safe.
2. Assume I_1 sees M_2 (this case covers the case when M_1 sees M_2). To ensure that variables of M_2 are not modified elsewhere, it suffices to have:
   $can\_alter \cap sees = \emptyset$.
3. A similar analysis can be done when M_2 is indirectly seen by I_1. For example, I_1 sees or imports a machine M_3, which sees M_2 (a longer chain of SEES between M_3 and M_2 may be allowed). In this case, I_1 may call an operation of M_3 which consults a variable of M_2. To ensure that the variable cannot be modified elsewhere:
   $can\_alter \cap (uses; sees^+) = \emptyset$.
4. These two cases are treated by the condition:
   $can\_alter \cap ((imports; sees^+) \cup (sees^+)) = \emptyset$.
5. This condition is too restrictive in the special case where global correctness is a consequence of local correctness: when the modification of the variable is explicit in the operation. For example, I_1 imports M_3 which sees M_1 (the chain of sees may be longer), and I_1 also imports M_1. In this case, local proof obligations cannot be satisfied. So it suffices to consider:
   $(uses; can\_alter) \cap ((imports; sees^+) \cup (sees^+)) = \emptyset$.
6. We now analyse the fact that variables of a seen machine cannot appear in the invariant of the seeing component. For example, in the architecture of Fig. 10, machine B_4 is only seen in the implementation BI_3 of B_3. Variables of B_3 are necessarily independent of variables of B_4, so they cannot be modified by an operation of B_2. So the condition becomes:
   $(uses; can\_alter) \cap ((imports; s^+) \cup (sees; s^*) = \emptyset$.

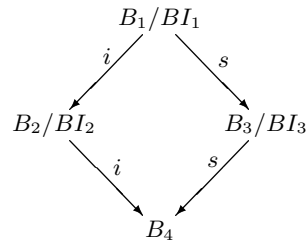

Fig. 10. A correct architecture (SEES are only in implementations).

So the sufficient architectural condition ensuring global correctness is:

$$\boxed{(uses; can\_alter) \cap ((imports; s^+) \cup (sees; s^*)) = \emptyset}$$

6.6 Simplifying Architectural Condition

Architectural condition can be reduced, if we use the following hypothesis, which is not stated in the B-Book:

> *If a machine M is indirectly seen through a chain of* SEES *primitives from a component C, possibly via an* IMPORTS *primitive, then M must be directly in the scope of a* SEES *or* IMPORTS *primitive in C.*

Using dependency relations, this hypothesis becomes:

1. For a machine or a refinement: $s^+ \subseteq s$,
2. For an implementation: $s^+ \subseteq s \cup imports$,
3. $imports; s^+ \subseteq sees \cup imports$.

This hypothesis is consistent with the non-transitivity of the SEES primitive. It imposes that each variable which can appear in a proof obligation of a component C is in the direct scope of its definition. With this hypothesis, architectural condition can be simplified, because chains of SEES primitives no more need to be considered. We have:

1. $imports; s^+ \subseteq sees \cup imports$.
2. $sees; s^* = sees \cup sees; s^+ \subseteq sees; s \cup sees; imports \subseteq sees \cup imports \cup sees; imports$.
3. So architectural condition is ensured by:
   $(uses; can\_alter) \cap (sees \cup imports \cup sees; imports) = \emptyset$.
4. Because a machine is imported only once, $(uses; can\_alter) \cap (imports \cup sees; imports) = \emptyset$, so condition becomes: $(uses; can\_alter) \cap sees = \emptyset$.
5. Because $can\_alter = imports \cup (uses; can\_alter)$ and $imports \cap sees = \emptyset$ (an implementation cannot see and import the same machine), we have: $(uses; can\_alter) \cap sees = can\_alter \cap sees$.

In consequence, architectural condition becomes:

$$\boxed{can\_alter \cap sees = \emptyset}$$

7 Conclusion

We have studied the B-Method in the light of the refinement calculus. The main results are:

1. an explanation of the proof obligations of a B-refinement in terms of standard refinement (this is Theorem 1);
2. an explanation of the SEES problem ([PR98]) in terms of abusive use of monotonicity, in presence of aliasing;
3. a sufficient architectural condition ensuring global correctness of a B-software.

An immediate perspective would be the definition of a sharing primitive, allowing several machines to see and modify the same machine. To be useful, variables of shared machines should be allowed in invariants of sharing components, so we should enforce the architectural condition. It is probable that this architectural condition would be too restrictive, so a fine analysis of operations would be necessary, to precisely know where the variables are modified.

We think that our kind of investigation is useful for studying evolution of the B-Method, because known results of refinement calculus can be directly applied, and current research can be a food for thought. The following list is not exhaustive:

1. New substitution constructors could be defined, using the results of [BB98] (the authors note that the B parallel operator $\|$ is equivalent to their derived product, for conjunctive commands). In [BPR96], we introduced a $\otimes$ operator, extending $\|$ when variables are shared; it looks like the fusion operator of [BB98].
2. Object-oriented extension could be studied, in the light of [MS97].
3. A current research in B is its application to distributed systems ([AM98]). A comparative study with action systems ([BK83], referenced in [AM98]) would be useful. This work has already begun ([BuW96], [WS96]).

Acknowledgment

We thank Pierre Berlioux for his careful reading of Section 3 and for useful suggestions.

References

[Abr96] J-R. Abrial, *The B-Book*, Cambridge University Press, 1996.

[AM98] J-R. Abrial, L. Mussat, *Introducing Dynamic Constraints in B*, In Second B International Conference, D. Bert editor, LNCS 1393, 83–128, Springer, 1998.

[AtB] Steria Méditerranée, *Le Langage B. Manuel de référence version 1.5* , S.A.V. Steria, BP 16000, 13791 Aix-en-Provence cedex 3, France.

[Ba78] R. J. R. Back, *On the Correctness of Refinement Steps in Program Development*, Report A-1978-4, Dept. of Computer Science, University of Helsinki, 1978.

[BB98] R. J. R. Back, M. Butler, *Fusion and Simultaneous Execution in the Refinement Calculus*, Acta Informatica 35, 921–949, 1998.

[BK83] R. J. R. Back, R. Kurki-Suonio, *Decentralisation of Process Nets with Centralized Control*, In 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 131–142, 1983.

[BW90] R. J. R. Back, J. von Wright, *Refinement calculus I: Sequential Nondeterministic Programs*, In Stepwise Refinement of Distributed Systems, J. W. deBakker, W. P. deRoever, G. Rozenberg, editors, LNCS 430, 42–66, Springer, 1990.

[BW92] R. J. R. Back, J. von Wright, *Combining Angels, Demons and Miracles in Program Specifications*, Theoretical Computer Science 100, 365–383, 1992.

[BW98] R. J. R. Back, J. von Wright, *Refinement Calculus - A Systematic Introduction*, Springer, 1998.

[BPR96] D. Bert, M-L. Potet, Y. Rouzaud, *A Study on Components and Assembly Primitives in B*, In First Conference on the B Method, 47–62, H. Habrias editor, 1996.

[BuW96] M. Butler, M. Walden, *Distributed System Development in B*, In First Conference on the B Method, 155–168, H. Habrias editor, 1996.

[CU89] W. Chen, J. T. Udding, *Towards a Calculus of Data Refinement*, In Mathematics of Program Construction, J. L. A. van de Snepsheut editor, LNCS 375, 197–218, Springer, 1989.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Du97] S. Dunne, *Parallel Composition*, In B-Talk and BUG mailing-lists, http://estas1.inrets.fr:8001/ESTAS/BUG/WWW/MailArchives/, 97-12/msg00003.html and 98-01/msg00003.html.

[Du99] S. Dunne, *The Safe Machine: a new specification construct for B*, In Proceedings of FM'99, World Congress on Formal Methods, Toulouse, 1999.

[GM93] P. H. B. Gardiner, C. Morgan, *A Single Complete Rule for Data Refinement*, Formal Aspects of Computing, 5, 367–392, 1993. Reprinted in [Mo92].

[GP85] D. Gries, J. Prins, *A New Notion of Encapsulation*, in Proc. of Symp. on Languages Issues in Programming Environments, SIGPLAN, 131–139, 1985.

[Met] Line 14: Meteor operation, http://www.ratp.fr (available in english).

[MS97] A. Mikhajlova, E. Sekerinski, *Class Refinement and Interface Refinement in Object-Oriented Programs*, In FME'97, J. Fitzgerald, C. Jones, P. Lucas editors, LNCS 1313, 82–101, Springer, 1997.

[Mo88] C. Morgan, *The Specification Statement*, ACM TOPLAS 10(3), July 1988. Reprinted in [Mo92].

[Mo92] C. Morgan, *On the Refinement Calculus*, Springer-Verlag, 1992.

[Mor87] J. Morris, *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, Science of Computer Programming 9, 287–306, 1987.

[PR98] M-L. Potet, Y. Rouzaud, *Composition and Refinement in the B-Method*, In Second B International Conference, D. Bert editor, LNCS 1393, 46–65, Springer, 1998.

[WS96] M. Walden, K. Sere, *Refining Action Systems within B-Tool*, In FME'96, M-C. Gaudel, J. Woodcock editors, LNCS 1051, 84–103, Springer, 1996.

[Wi71] N. Wirth, *Program Development by Stepwise Refinement*, CACM 14(4), 221–227, 1971.

[Wri94] J. von Wright, *The Lattice of Data Refinement*, Acta Informatica 31, 105–135, 1994.