

Module OMGL - UE Méthode B

La méthode B

pour la construction rigoureuse de logiciels

J. Christian Attiogbé

Novembre 2008, maj 11/2009, 10/2010



Plan de ce cours cours

La Méthode B : une introduction

- Introduction : qu'est-ce que c'est
 - Aperçu rapide
 - Exemple de spécification
 - Contrôle Lumière
- Comment développer avec B
 - Analyse système
 - Structuration : Machines abstraites
 - Modélisation données
 - Modélisation opérations
 - Raffinements
 - Implantation



Exemples de développement

- Exemples
 - PGCD, division euclidienne,
 - Tri
- Les concepts à la base de la méthode
 - Modélisation de la partie statique (données)
 - Modélisation de la partie dynamique (opérations)
 - Preuves de cohérence
 - Raffinement
 - Preuves de raffinement
- Etudes de cas (avec AtelierB, B4free, Rodin)

Méthode B

- (..1996) Méthode pour **spécifier, concevoir et développer** des systèmes informatiques **séquentiels**
- (1998..) Event B ... systèmes **distribués, concurrents**
- (...) méthode en constante évolution, mais les outils ne suivent pas au même rythme ;-(

Applications industrielles

- Applications sécuritaires dans le Transport
Freinage, portes palières (ligne 13, metro Paris),
- KVS, Metro de Calcutta, Caire
- INSEE (recensement)
- Meteor RATP : pilotage automatique, généralisation portes palières
- Cartes à puce : sécurisation, ...
- Peugeot
- etc

👉 compétences très recherchées par certaines entreprises.

Méthodes en génie logiciel

Méthode formelle =

- Langage de spécification/modélisation formelle
- Système formel (de raisonnement)

Méthode B =

- Langage de spécification
 - Logique, théorie des ensembles : langage de données
 - Langage des substitutions généralisées : langage des opérations
- Système formel (de raisonnement)
 - Prouveur de théorèmes

Développement formel

Développement formel de logiciel =

- Transformation systématique d'un modèle mathématique en code exécutable.
 - = Transformation de l'abstrait en concret
 - = Passage des structures mathématiques aux structures informatiques
 - = Raffinement jusqu'au code dans un langage de progr.

B : Méthode formelle

+ théorie de raffinement (de machines abstraites)

⇒ méthode de développement formel

Notion d'états et d'espaces d'états

- Observons une **variable** dans un modèle logique ;
- Elle peut avoir **différentes valeurs** au cours du temps, ou encore plusieurs états au cours du temps ;
- Par exemple une **variable entière** I : on peut observer logiquement $I = 2, I = 6, I = 0, \dots$ à condition que I soit modifiée... ;
- Suite à une modification I change d'états ;
- Le changement d'état peut être modélisé par une action qui substitue une nouvelle valeur à celle de la variable ;
- De façon générale, pour I entier, il y a potentiellement tout l'espace des entiers comme états que I peut avoir : on parle d'**espace d'états**.
- On généralise à plusieurs variables $\langle I, J \rangle, \langle V1, V2, V4, \dots \rangle$

Approche de développement

Les approches Z, TLA, B, ... sont dites : **orientées modèle** (à états)

- Décrire un **espace d'états**
- Décrire des opérations qui parcourent cet espace
- Système de **transition entre états**

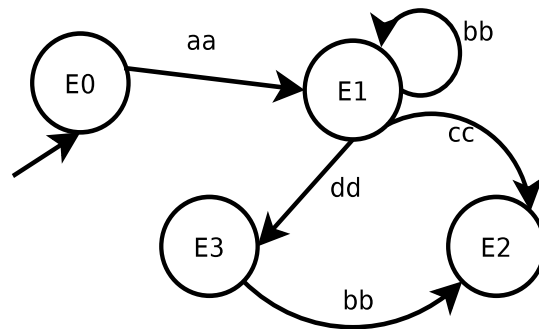


Figure: Evolution d'un système logiciel

Approche de spécification

- Un nuplet de **variables** décrit **un état**

$$\langle mode = jour, lumiere = eteint, temp = 20 \rangle$$

- Un **prédicat** (sur les variables) décrit **un espace d'états**

$$lumiere = eteint \wedge mode = jour \wedge temp > 12$$

- Une **opération** qui affecte les variables **change l'état**

$$mode := jour$$

Spécifier en B = Modéliser, décrire un système de transition
(avec une approche logique)

Machine abstraite

variables

prédicat

opération

```

MACHINE ...
SETS ...
VARIABLES
...
INVARIANT
... predicat
INITIALISATION
...
OPERATIONS
...
END
  
```

Machine abstraite

```

MACHINE ReguLum
SETS
MODEJ = {jour, nuit}
; ETATLUM = {eteint, allume}
  
```

- Toute machine abstraite a un nom
- La **clause SETS** permet de considérer des ensembles abstraits ou énumérés ; Ces ensembles seront utilisés pour **typer** les variables
- les ensembles prédéfinis sont : NAT, INTEGER, BOOL, etc

Machine abstraite

VARIABLES

```
mode
, lumiere
, temp
```

INVARIANT

```
mode : MODEJ
& lumiere : ETATLUM
& temp : ENTIER
```

- La **clause VARIABLES** permet de lister les variables dont on va se servir dans la spécification
- La **clause INVARIANT** permet de donner le prédicat décrivant les **propriétés invariantes** de la machine abstraite ; ce qui **doit toujours être vrai et vérifié**
- les deux clauses vont de paire

Machine abstraite

INITIALISATION

```
mode := jour
|| temp := 20
|| lumiere := eteint
```

- Dans une machine abstraite, on doit donner un état initial du système spécifié. Il faut que cet état initial vérifie les propriétés invariantes.
La **clause INITIALISATION** permet d'initialiser TOUTES les variables listées dans la machine. L'initialisation par substitution, se fait **simultanément** pour toutes les variables.
On pourra ensuite les modifier avec les opérations.

Machine abstraite

OPERATIONS

```

changerMode =
CHOICE mode := jour
OR mode := nuit
END
;
allumer =
lumiere := allume
;
eteindre =
lumiere := eteint
;
baisserTemp = temp := temp - 1
;
monterTemp = temp := temp +1
END

```

- Sous la **clause OPERATIONS** on liste toutes les opérations voulues pour la machine abstraite. Les opérations modélisent les **changements d'état des variables** par des **substitutions logiques** (notées :=). Les substitutions sont généralisées pour plus d'expressivité. Les opérations ont une **PREcondition** (la POST est implicitement l'invariant)

Machine abstraite : exemple Regul. lumière

MACHINE ReguLum

SETS

```

MODEJ = {jour, nuit}
; ETATLUM = {eteint, allume}

```

VARIABLES

```

mode
, lumiere
, temp

```

INVARIANT

```

mode : MODEJ
& lumiere : ETATLUM
& temp : ENTIER

```

INITIALISATION

```

mode := jour || temp := 20
|| lumiere := eteint

```

OPERATIONS

```

changerMode =
CHOICE mode := jour
OR mode := nuit
END
;
allumer =
lumiere := allume
;
eteindre =
lumiere := eteint
;
baisserTemp = temp := temp - 1
;
monterTemp = temp := temp +1
END

```


Machine abstraite : offre des opérations

Une machine abstraite offre des opérations qui sont appelables à partir d'autres opérations/programmes externes.

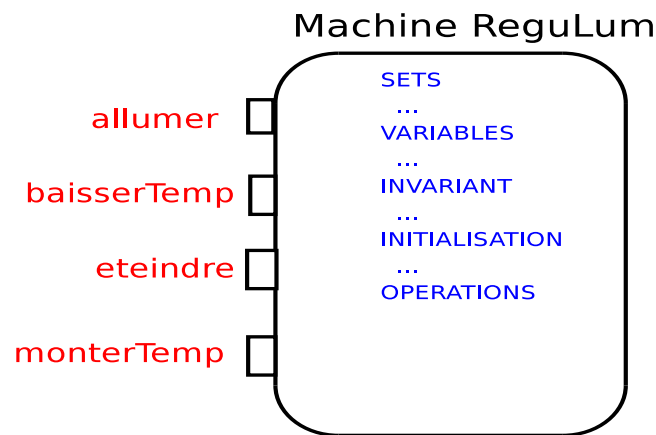


Figure: Les opérations sont appelables de l'extérieur

☞ Une opération d'une machine ne peut pas en appeler une autre

Interface des opérations

(avec/sans paramètres entrée/sortie)

- Aucun paramètre : $\text{nomOperation} = \dots$
- Que des paramètres d'entrée : $\text{nomOperation}(p_1, p_2, \dots) = \dots$
- Paramètres de sortie $r_1, r_2, \dots \leftarrow \text{nomOperation} = \dots$
- Entrée et sortie $r_1, r_2, \dots \leftarrow \text{nomOperation}(p_1, p_2, \dots) = \dots$

Systeme de régulation de lumière

Etude

Exigences :

- La lumière ne doit pas être allumée s'il fait jour
- La température ne doit pas dépasser 29 degrés quand il fait jour
- ...

⇒ trouver et formaliser les propriétés dans l'invariant.

Machine abstraite : exemple de la jauge

```

MACHINE NomMach
VARIABLES
jauge
INVARIANT
jauge : NAT
& jauge >= 2
& jauge <= 45
INITIALISATION
jauge := 1 // !! quoi ?

```

```

OPERATIONS
decrementer1 =
PRE jauge > 2
THEN jauge := jauge - 1
END
; decrementer(pas) =
PRE pas : NAT
& jauge - pas >= 2
THEN
jauge := jauge - pas
END
...
incrementer ...
...
END

```

Machine abstraite : exemple des ressources

MACHINE

Resrc

SETS

RESC

CONSTANTS

maxRes // un paramètre

PROPERTIES

maxRes : NAT & maxRes > 1

VARIABLES

rsc

INVARIANT

rsc <: RESC
& card(rsc) <= maxRes // bornée

INITIALISATION

rsc := {}

OPERATIONS

addRsc(rr) = // ajout

PRE

rr : RESC & rr /: rsc &
card(rsc) < maxRes

THEN

rsc := rsc \\/ {rr}

END

;

rmvRsc(rr) = // retrait

PRE

rr : RESC & rr : rsc

THEN

rsc := rsc - {rr}

END

END

Bases de la construction de programmes corrects

Considérons des opérations sur un compte en banque :

- un retrait de `somDemandee`

```
begin
  compte := compte - somDemandee
end
```

- approvisionnement du compte avec `nouvSomme`

```
begin
  compte := compte + nouvSomme
end
```

☞ ces opérations ne sont pas satisfaisantes, elles ne tiennent pas compte des contraintes (les seuils à ne pas dépasser).

Bases de la construction de programmes corrects

- un retrait de `somDemandee`

```
retrait(compte, somDemandee)=
pre
  compte - somDemandee >= 0 //découvert non autorisé
begin
  compte := compte - somDemandee
end
```

☞ Avant d'appeler l'opération de retrait on s'assure qu'elle n'occasionne pas de découvert.

```
Si retraitPossible(compte, somDemandee)
  alors retrait(compte, somDemandee)
Finsi
```

Bases de la construction de programmes corrects

Soient deux entiers `nombreN` et `nombreD`.
Que se passe-t-il avec l'instruction suivante ?

```
res := nombreN / nombreD
```

Ce qu'il fallait faire :

```
Si (nombreD /= 0)
  Alors res := nombreN / nombreD
Finsi
```

En effet l'opération de **division a une précondition : (denom /= 0)**

B : le principe de la méthode

L'invariant d'un système (ou logiciel)

- on modélise l'**espace des états corrects par une propriété** (une conjonction de propriétés).
- Tant que le système est dans un de ces états, **il est fonctionne bien ; il faut l'y maintenir !**
- Donc il faut éviter qu'il sorte de cet espace d'états.
- Donc s'assurer de l'état résultant, **avant de faire les changements d'états.**

Exemples : trajectoire d'un robot (éviter les points de collision).

☞ Les opérations qui changent l'état ont une précondition.

B : Approche logique

Originalité de B : tout en logique (données et opérations)

- Espace d'état : Invariant : Prédicat : $P(x, y, z)$
Un état : une **valuation des variables**
 $x := v_x \quad y := v_y \quad z := v_z \quad \text{dans } P(x, y, z)$
⇒ **Substitution logique**
- Une opération : transforme un état (correct) en un autre état.
Transformer un état = **transformer le prédicat** (invariant)
Opération = transformateur de prédicat = Substitution
Effets autre que affectation ⇒ **substitutions généralisées**

B : la pratique

Quelques règles de spécification en B

- Une opération d'une machine ne peut pas appeler une autre opération de la même machine (violation de la PRE) ;
- On ne peut pas (de l'extérieur) appeler en parallèle, deux opérations d'une même machine (par exemple : `incr || decr`) ;
- Une machine doit comporter des opérations auxiliaires pour tester les préconditions des principales opérations fournies ;
- L'appelant d'une opération doit vérifier ses préconditions avant l'appel ("On ne doit pas diviser par 0") ;
- Lors des raffinements, on affaiblit les PREconditions jusqu'à les faire disparaître (ce n'est pas le cas en B événementiel) ;
- ...

B : les fondements

- Logique du premier ordre
- Théorie des ensembles (+ types)
- Théorie des substitutions généralisées
- Théorie du raffinement
- et une bonne dose de : abstraction et composition !

B : Outils de Génie logiciel

- **Modularité :**
Machine abstraite, Raffinement, Implantation
- **Architecture des applications complexes :**
clauses **SEES, USES, INCLUDES, IMPORTS, ...**
- **Atelier de génie logiciel :**
Editeurs, analyseurs, prouveurs, ...

Introduction à B

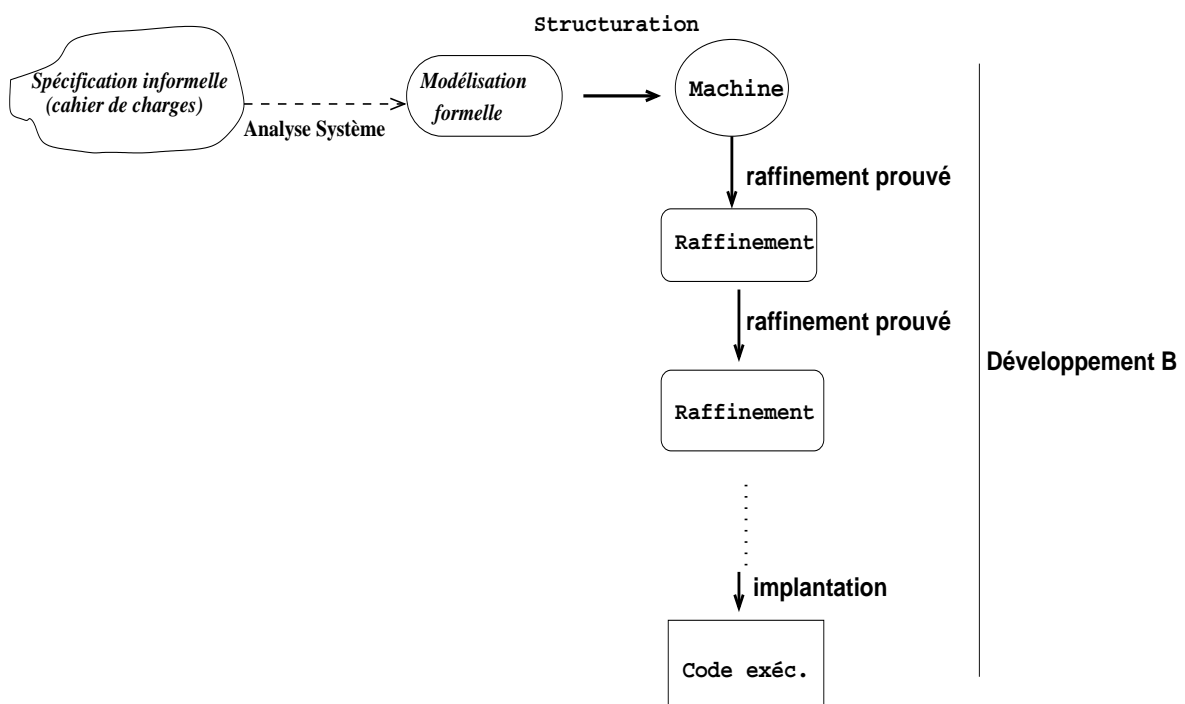
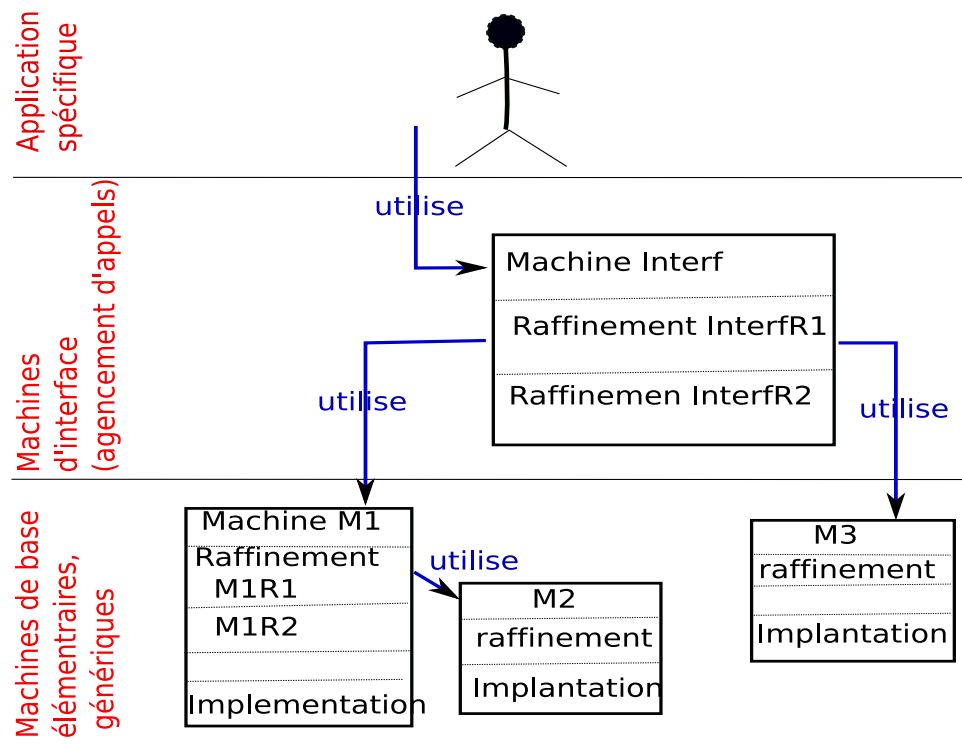


Figure: Analyse et développement B



Bibliothèques de machines prédéfinies

Figure: Structure d'un développement en B

Positionnement - autres méthodes

- ☞ B : Approche développement correct → *preuves*
- ☞ B : Cadre unique pour :
 - Analyse
 - Spécification
 - Conception
 - Développement
- ☞ B: Raffinements successifs du modèle abstrait au code.
- ☞ (Autres) Approches : développement, test à postériori → *tests*

Développement du PGCD : machine abstraite

MACHINE

```
pgcd1 /* pour le PGCD de deux entiers, JCA, U. Nantes */
      /* pgcd(x,y) is  $d \mid x \bmod d = 0 \wedge y \bmod d = 0$ 
       $\wedge \forall \text{ other divisors } dx \ d > dx$ 
       $\wedge \forall \text{ other divisors } dy \ d > dy$  */
```

OPERATIONS

```
rr <-- pgcd(x,y) = /* SORTIE : rr ; ENTREE x, y */
```

```
...
```

END

Développement du PGCD : machine abstraite

OPERATIONS

```
rr <-- pgcd(x,y) = /* spécification du pgcd */
```

PRE

```
x : INT & x >= 1 & x < MAXINT
```

```
& y : INT & y >= 1 & y < MAXINT
```

THEN

```
ANY d WHERE
```

```
d : INT
```

```
& (x - (x/d)*d) = 0 /* d is a divisor of x */
```

```
& (y - (y/d)*d) = 0 /* d is a divisor of y */
```

```
/* and the other common divisors are < d */
```

```
& !dx.((dx : INT & dx < MAXINT
```

```
& (x - (x/dx)*dx) = 0 & (y - (y/dx)*dx) = 0) => dx < d)
```

```
THEN rr := d
```

```
END
```

END

Développement du PGCD : raffinement

```

REFINEMENT /* raffinement de ...*/
  pgcd1_R1
REFINES pgcd1 /* la machine précédente */
OPERATIONS
rr <-- pgcd (x, y) = /* l'interface ne change pas */
  BEGIN
    ... Corps de l'opération raffinée
  END
END

```

Développement du PGCD : raffinement

```

rr <-- pgcd (x, y) = /* opération raffinée */
  BEGIN
    VAR cd, rx, ry, cr IN
      cd := 1
      ; WHILE ( cd < x & cd < y) DO
        ; rx := x - (x/cd)*cd ; ry := y - (y/cd)*cd
        IF (rx = 0 & ry = 0)
          THEN /* cd divise x et y, possible GCD */
            cr := cd /* possible rr */
          END
        ; cd := cd + 1 ; /* on tente plus grand */
      INVARIANT
        x : INT & y : INT & rx : INT & rx < MAXINT
        & ry : INT & ry < MAXINT & cd < MAXINT
        & x = cr*(x/cr) + rx & y = cr*(y/cr) + ry
      VARIANT
        x - cd
    END
  END

```

Forme générale simplifiée d'une Machine Abstraite

MACHINE

M (prm) /* Nom de la machine et ses paramètres */

CONSTRAINTS

C /* Prédicat sur X et x */

/* clauses USES, SEES, INCLUDES, EXTENDS, */

SETS

ENS /* liste d'identificateurs d'ensembles de base */

CONSTANTS

K /* liste d'identificateurs */

PROPERTIES

B /* prédicat(s) sur K */

VARIABLES

V /* liste d'identificateurs de variables */

DEFINITIONS

D /* liste de définitions (abréviations/macros) */



Forme générale simplifiée (suite ...)

...

INVARIANT

|

INITIALISATION U

OPERATIONS

$u \leftarrow O(pp) =$

PRE

P

THEN

$Subst$

END;

...

end



Sémantique : cohérence de la machine

$$\exists prm.C$$

Il est possible d'avoir des valeurs de paramètres satisfaisant les contraintes

$$C \Rightarrow \exists (ENS, K).B$$

Il y a des ensembles et des constantes qui satisfont les propriétés de la machine

$$B \wedge C \Rightarrow \exists V.I$$

Il y a un état satisfaisant l'invariant

$$B \wedge C \Rightarrow [U]I$$

L'initialisation établit l'invariant

Pour chaque opération de la machine

$$B \wedge C \wedge I \wedge P \Rightarrow [Subst]I$$



toute opération appelée sous sa précondition préserve l'invariant

Obligations de preuve

Les prédicats à prouver pour s'assurer de la cohérence (et de la correction) du modèle mathématique défini par la machine abstraite. Le développeur de la machine abstraite a deux types d'obligations de preuve :

- prouver que l'**INITIALISATION** établit l'invariant ;
- prouver que **chaque OPERATION**, lorsqu'elle est appelée sous sa **précondition**, préserve l'invariant.

$$I \wedge P \Rightarrow [Subst]I$$

Dans la pratique, on s'équipe d'outils ou d'ateliers qui aident à décharger ces preuves.

Sémantique d'une machine - Cohérence : démarche

Pour établir formellement les conditions de bon fonctionnement d'une machine, on parle d'**obligations de preuve**.

Pour garantir la correction d'une machine, on doit s'affranchir des 2 principales obligations de preuve :

- L'initialisation établit l'invariant
- Chaque opération de la machine, lorsqu'elle est appelée sous sa précondition, préserve l'invariant.

Ce sont là des expressions logiques, des prédicats, qui sont prouvés.

Nouvel Exemple

...TRI...

Exemple de spécification en B

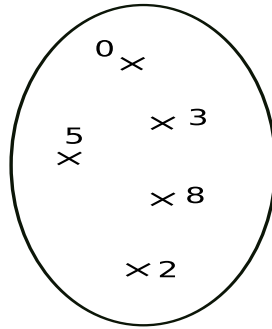
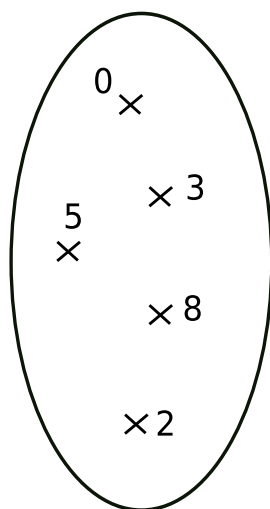


Figure: Modélisation du tri : donnée de départ (ens. d'entiers)

Exemple de spécification en B



ensemble d'entiers :
FIN(NAT)

0 2 3 5 8

ordonné

Figure: Modélisation du tri : on ordonne un ens. d'entiers

Exemple de spécification en B

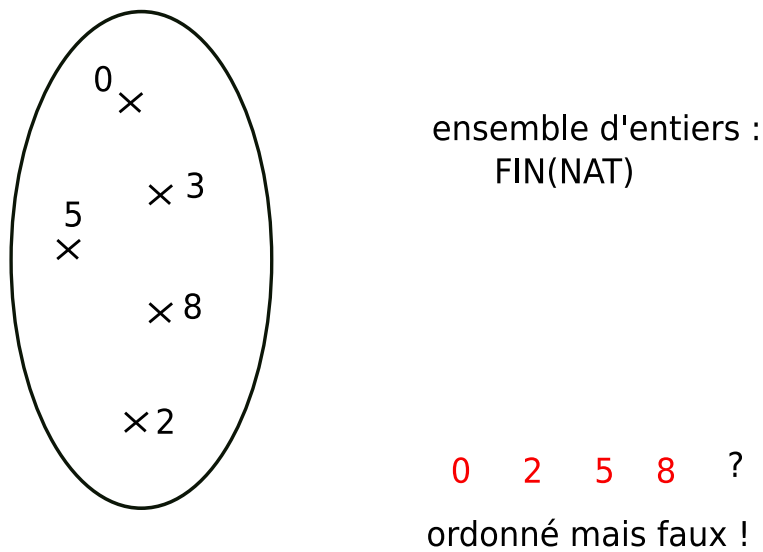


Figure: Modélisation du tri : mais attention !

Exemple de spécification en B

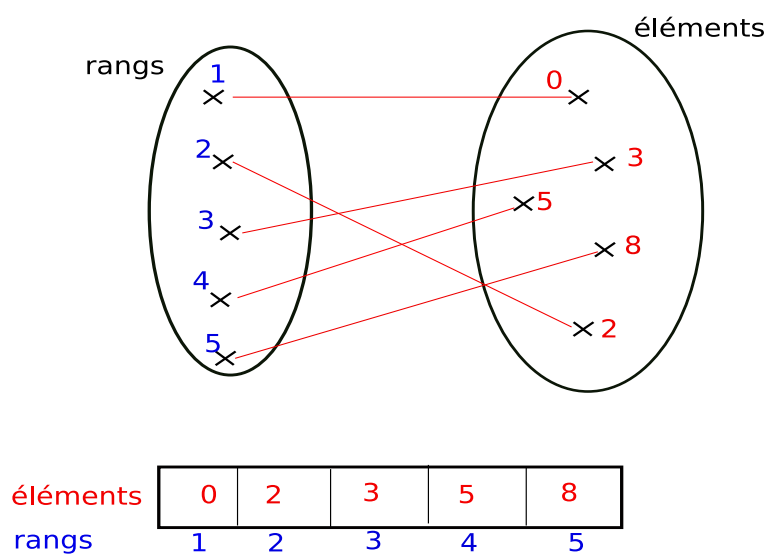


Figure: Modélisation du tri

Exemple de spécification en B

```

MACHINE /* Spécifie le tri d'un ens. d'entiers -> seq. d'entiers */
  Tri
CONSTANTS
  tride /* définition d'une fonction */
PROPERTIES
  tride : FIN(NAT) -> seq(NAT) &
    %ss.(ss : FIN(NAT) =>
      (ran(tride(ss)) = ss &
        %(ii,jj).(ii : dom(tride(ss)) &      jj : dom(tride(ss)) &
          ii < jj =>
            (tride(ss))(ii) < (tride(ss))(jj) ) ) )
END

```



Exemple de spécification en B

```

MACHINE
  SpecTri
  /* spécifie une appli qui enregistre des entiers puis les trie */
SEES
  Tri /* utilisation de la machine précédente */
SETS
  ModeTri = {insertion, extraction}
VARIABLES
  nontrie, trie, mode
INVARIANT
  nontrie : FIN(NAT)
  &      trie : seq(NAT)
  &      mode : ModeTri
  &      ((mode = extraction) => (trie = tride(nontrie)))

```



Exemple de spécification en B

```

/* MACHINE SpecTri
    (suite ...) */

INITIALISATION
    nontrie := {} || trie := [] || mode := extraction

OPERATIONS
    passerEnInsertion =
        PRE
            mode = extraction
        THEN
            mode := insertion ||
            nontrie := {} ||
            trie :: seq(NAT)
        END
    ;
    ...

```



Exemple de spécification en B

```

/* MACHINE SpecTri
    (suite ...) */

    inserer(xx) =
        PRE
            xx : NAT & mode = insertion
        THEN
            nontrie := nontrie \ / {xx} ||
            trie :: seq(NAT)
        END
    ;

    passerEnExtraction() =
        PRE
            mode = insertion
        THEN
            mode := extraction ||
            trie := tride(nontrie)
        END

```



Exemple de spécification en B

```
/* MACHINE SpecTri
(suite ...) */

yy <- extraire(ii) =
    PRE
        ii : dom(trie) & mode = extraction
    THEN
        yy := trie(ii)
    END
END
```

Fin exemple

...FIN exemple Tri...

B - Langage des données - ensembles et typage

- Ensembles prédéfinis (statut de **type**)
BOOL, **CHAR**,
INTEGER (\mathbb{Z}), **NAT** (\mathbb{N}), **NAT1** (\mathbb{N}^*) ,
STRING
- Produit cartésien $E \times F$
- Ensemble des sous-ensembles $\mathcal{P}E$
noté **POW(E)**

B - Langage des données

A l'aide du langage des données

- On **modélise l'état d'un système** avec les données qui le caractérisent
- on explicite les **propriétés invariantes** d'un système

Modélisation de l'état :

- **Abstraction, modélisation** (ensembles abstraits, relations, fonctions, ...)
- **Propriétés logiques**, algébriques.

B - Langage des données

- Lorsqu'on modélise un système (par l'ensemble de ces états) puis qu'on explicite ses (bonnes) propriétés, on s'assure après que le système ne parcourent que l'ensemble des états qui respectent les propriétés : c'est la **cohérence du système**.
- Pour montrer qu'il est possible d'avoir des états satisfaisant les propriétés énoncées, on exhibe au moins un état (l'état initial).
- Le système spécifié est correct si après chacune de ses opérations, l'état obtenu est un état respectant les propriétés invariantes.

B - Langage des données

Logique du premier ordre

Désignation	Notation	Ascii
et	$p \wedge q$	p & q
ou	$p \vee q$	p or q
non	$\neg p$	not p
implication	$p \Rightarrow q$	(p) ==> (q)
quantif. univ.	$\forall x.p(x)$!x.(p(x))
quantif. exist.	$\exists x.p(x)$	#x.(p(x))

Il faut typer les x quantifiés :

#x.(x : T ==> p(x)) et **!x.(x : T ==> p(x))**

B - Langage des données

Les opérateurs ensemblistes classiques

E , F et T des ensembles, x un élément de F

Désignation	Notation	Ascii
union	$E \cup F$	<code>E \ / F</code>
intersection	$E \cap F$	<code>E /\ F</code>
appartenance	$x \in F$	<code>x : F</code>
différence	$E \setminus F$	<code>E - F</code>
inclusion	$E \subseteq F$	<code>E <: F</code>
sélection	<code>choice(E)</code>	

+ Union et intersection généralisées

+ Union et intersection quantifiées

B - Langage des données

En notation ascii la négation est réalisée avec `\`.

Désignation	Notation	Ascii
non appartenance	$x \notin F$	<code>x /: F</code>
non inclusion	$E \not\subseteq F$	<code>E /<: F</code>
non égalité	$E \neq F$	<code>E /= F</code>

Union généralisée

$$S \in \mathcal{P}(\mathcal{P}(T))$$

$$\Rightarrow$$

$$\text{union}(S) = \{x \mid x \in T \wedge \exists u.(u \in S \wedge x \in u)\}$$

Exemple

$$\begin{aligned} & \text{union}(\{\{aa, ee, ff\}, \{bb, cc, gg\}, \{dd, ee, uu, cc\}\}) \\ &= \{aa, ee, ff, bb, cc, gg, dd, uu\} \end{aligned}$$

Union quantifiée

C'est un opérateur qui permet de faire l'union généralisée d'expressions ensemblistes bien définies.

$$\forall x.(x \in S \Rightarrow E \subseteq T)$$

$$\Rightarrow$$

$$\bigcup x.(x \in S \mid E) = \{y \mid y \in T \wedge \exists x.(x \in S \wedge y \in E)\}$$

Exemple

$$\begin{aligned} & \text{UNION}(x).(x \in \{1, 2, 3\} \mid \{y \mid y \in \text{NAT} \wedge y = x * x\}) \\ &= \{1\} \cup \{4\} \cup \{9\} = \{1, 4, 9\} \end{aligned}$$

Intersection généralisée

$$S \in \mathcal{P}(\mathcal{P}(T))$$

\Rightarrow

$$\mathit{inter}(S) = \{x \mid x \in T \wedge \forall u.(u \in S \Rightarrow x \in u)\}$$

Exemple

$$\mathit{inter}(\{\{aa, ee, ff, cc\}, \{bb, cc, gg\}, \{dd, ee, uu, cc\}\}) = \{cc\}$$

Intersection quantifiée

C'est un opérateur qui permet de faire l'intersection généralisée d'expressions ensemblistes bien définies.

$$\forall x.(x \in S \Rightarrow E \subseteq T)$$

\Rightarrow

$$\bigcap x.(x \in S \mid E)$$

$$= \{y \mid y \in T \wedge \forall x.(x \in S \Rightarrow y \in E)\}$$

Exemple

$$\mathit{INTER}(x).(x \in \{1, 2, 3, 4\} \mid \{y \mid y \in \{1, 2, 3, 4, 5\} \\ \wedge y > x\})$$

$$= \mathit{inter}(\{\{1, 2, 3, 4, 5\}, \{2, 3, 4, 5\}, \{3, 4, 5\}, \{4, 5\}\})$$

Les relations

Désignation	Notation	Ascii
relation	$r : S \leftrightarrow T$	$r : S \leftrightarrow T$
domaine	$dom(r) \subseteq S$	$dom(r) <: S$
image	$ran(r) \subseteq T$	$ran(r) <: T$
composition	$r;s$	$r;s$
composition $r(s)$	$r \circ s$	$r(s)$
identité	$id(S)$	$id(S)$

Les relations (suite)

Désignation	Notation	Ascii
restriction domaine	$S \triangleleft r$	$S < r$
restriction codomaine	$r \triangleright T$	$r > T$
antirestriction domaine	$S \triangleleft r$	$S << r$
antirestriction codomaine	$r \triangleright T$	$r >> T$
inverse	r^{\sim}	$r \sim$
image relationnelle	$r[S]$	$r[S]$
écrasement	$r1 \oplus r2$	$r1 <+ r2$
produit direct de rel.	$r1 \otimes r2$	$r1 >< r2$
fermeture	$closure(r)$	$closure(r)$
fermeture reflexive trans.	$closure1(r)$	$closure1(r)$

Les fonctions

Désignation	Notation	Ascii
fonction partielle	$S \mapsto T$	S +-> T
fonction totale	$S \rightarrow T$	S --> T
injection partielle	$S \mapsto\!\!\!\rightarrow T$	S >+-> T
injection totale	$S \succrightarrow T$	S >->
surjection partielle	$S \twoheadrightarrow T$	S +->> T
surjection totale	$S \twoheadrightarrow T$	S -->> T
bijection totale	$S \xrightarrow{\sim} T$	S >->>
lambda abstraction	$\%x.(P \mid E)$	

Les séquences

Désignation	Notation
suite d'éléments de T	$seq(T)$ = $union(n).(n \in N$ $\mid 1..n \rightarrow T)$
suite vide	[]
suite inj. d'élém. de T	$iseq(T)$
suite bij. d'élém. de T	$perm(T)$
taille d'1 séq. s	$size(s) = card(dom(s))$

Les séquences (suite)

Désignation	Notation
premier élém. d'une séq. s	$first(s) = s(1)$
dernier élém. d'une séq. s	$last(s) = s(size(s))$
restric. de s à ses n prem. éléments	$s \uparrow n$
élimination de n premiers éléments de s	$s \downarrow n$

Concepts de base pour la partie dynamique

...

Concepts de base pour la partie dynamique

Les plus faibles préconditions

Contexte : *Logique de Hoare/Floyd/Dijkstra* triplet de Hoare
(Etat, espace d'état, commandes, exécution, **triplet de Hoare**)

$$\{P\} S \{R\}$$

S une **commande** et R un **prédicat décrivant le résultat de S** .

$wp(S, R)$, prédicat qui représente :

l'ensemble de tous les états | exécution de S commençant par un d'entre eux **se termine** en un temps fini dans un état satisfaisant R ,
 $wp(S, R)$ est la **plus faible précondition** de S par rapport à R .

Quelques exemples

Soient S une affectation et
 R le prédicat $i \leq 1$

$$wp(i := i + 1, i \leq 1) = (i \leq 0)$$

Soient S la conditionnelle suivante :

if $x \geq y$ then $z := x$ else $z := y$

et R le prédicat $z = \max(x, y)$

$$wp(S, R) = \text{Vrai}$$

Plus-faibles précondition - exemples

Le sens de $wp(S, R)$ peut être précisé par deux propriétés :

- $wp(S, R)$ est une **précondition garantissant R après l'exécution de S** , c'est à dire que :

$$\{wp(S, R)\} S \{R\}$$

- $wp(S, R)$ est **la plus faible de telles préconditions**, c'est à dire que :
si $\{P\} S \{R\}$ alors $P \Rightarrow wp(S, R)$

Plus-faibles précondition - exemples

En pratique un programme S établit une postcondition R .

Intérêt pour les préconditions qui permettent d'établir R .

wp est une fonction à deux arguments :

une instruction (ou programme) S et

un prédicat R .

Pour un S fixé, on peut voir $wp(S, R)$ comme une fonction à un seul argument $wp_S(R)$.

La fonction wp_S est appelé **transformateur de prédicats**.

C'est la fonction qui associe à tout prédicat R la plus faible précondition telle que $\{P\} S \{R\}$.

B : Substitutions généralisées - Axiomes

Généralisation de la substitution simple de la logique classique (pour modéliser des comportements d'opérations).

Soit R un prédicat à établir, la sémantique des substitutions généralisées est définie par **le transformateur de prédicat**.

- **Substitution simple** S
sémantique $[S]R$ se lit : **S établit R**
- **Substitution multiple** $x, y := E, F$
Sémantique $[x, y := E, F]R$

B : Substitutions généralisées - Jeu de base

Le langage de syntaxe abstraite pour spécifier les opérations:

Soit R l'invariant, S, T des substitutions

Nom	Synt. abs.	définition	équivalent logique
neutre (id.)	$skip$	$[skip]R$	R
Pré-condition	$P \mid S$	$[P \mid S]R$	$P \wedge [S]R$
Choix borné	$S \parallel T$	$[S \parallel T]R$	$[S]R \wedge [T]R$
Garde	$P \Longrightarrow T$	$[P \Longrightarrow T]R$	$P \Rightarrow [T]R$
non borné	$@x.S$	$[@x.S]R$	$\forall x.[S]R$ x non libre dans R

suffit comme langage de spécification de B mais ...

Non déterminisme - Substitutions

- **Abstraction** \Rightarrow non déterminisme possible. OK pour spécifier
- **concrétisation** \Rightarrow raffinement vers code
- Extension du jeu de base à d'autres substitutions proches de la programmation

CASE OF
SELECT
IF THEN ELSE

B - Langage des substitutions généralisées

Extension syntaxique des substitutions : jeu de base

Substitution simple

notée S

Extension syntaxique

BEGIN
 S
END

Substitutions simultanées

Soient S et T deux substitutions.

S étant $x := E$ et

T étant $y := F$

on note $S \parallel T$

B - Langage des substitutions généralisées

Substitution neutre

Extension syntaxique

skip

skip

Subst. avec précondition

Extension syntaxique

 $P \mid S$

PRE

P

THEN

S

END

B - Langage des substitutions généralisées

choix borné

Extension syntaxique

 $S \parallel T$

CHOICE

S

OR

T

END

Substitution avec garde

Extension syntaxique

 $(P \implies T) \parallel (\neg P \implies S)$

IF P

THEN T

ELSE S

END

B - Langage des substitutions généralisées

Substitution de choix non borné Extension syntaxique

$@x.S$

```
VAR x IN
S
END
```

Extension du jeu de base : non-déterminisme

Nondéterminisme @

$@x.(P \implies S)$

Extension syntaxique

```
ANY x
WHERE P
THEN S
END
```


Extension du jeu de base : non-déterminisme

Nondéterminisme $x : \in U$

(devient appartient)

$x :: U$

$@y.(y \in U \implies x := y)$

Extension syntaxique

ANY y

WHERE y : U

THEN x := y

END

B - Langage des substitutions généralisées

Extensions... non-déterminisme

Nondéterminisme $x : P(x)$

(x tel que P)

$x : P(x)$

BEGIN

x : (x < 30)

END

Démo division euclidienne

Euclid Pgm demo

```
+-----+
+  Menu de l'application      +
+-----+
+      Nouvelle division      : 1
+-----+
+      Quitter                : 0
+-----+
choix ? 1
```

Division euclidienne

Donnez le dividende (entre 3 et 78)

56

Donnez le diviseur (entre 1 et 78)

78

Resultat de la division : 0

Reste de la division : 56



suite démo

```
+-----+
+  Menu de l'application      +
+-----+
+      Nouvelle division      : 1
+-----+
+      Quitter                : 0
+-----+
```

choix ? 1

Division euclidienne

Donnez le dividende (entre 3 et 78)

67

Donnez le diviseur (entre 1 et 78)

6

Resultat de la division : 11

Reste de la division : 1



Spécification de Euclide

MACHINE

euclide

OPERATIONS

reste, quot \leftarrow calculReste (divis, divid) =

PRE

divis \in NAT \wedge divid \in NAT \wedge divis $>$ 0
 \wedge divis \leq divid /* sinon B le trouve */

THEN

ANY vq, vr **WHERE**

vq \in NAT

\wedge vr \in NAT

\wedge divid = vq*divis + vr

THEN

quot := vq

|| reste := vr

END

END

END



Exemple de développement en B

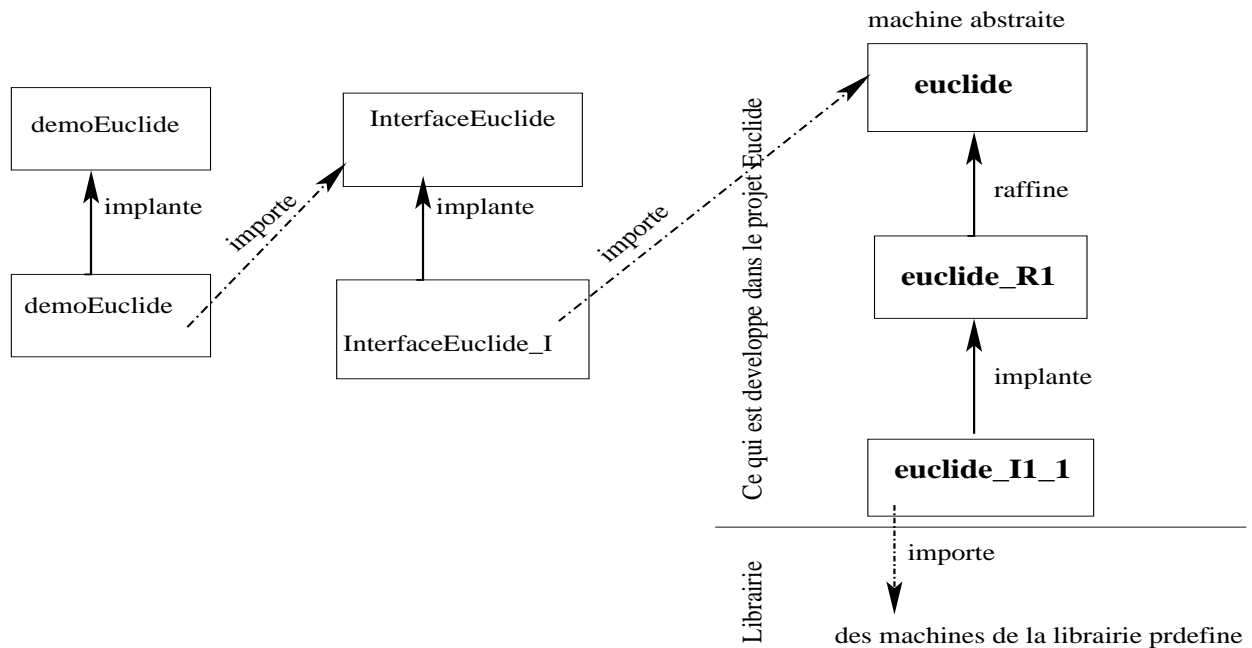


Figure: Structure des applications en B

