Méthodologie de construction du logiciel (M3301-2)

Méthode B

J. Christian Attiogbé

Université de Nantes

Plan du cours

Méthode de production d'applications correctes

Logiciel correct ? (application correcte ?)

Un logiciel est défini par ses états corrects (ceux attendus en fonction des fonctionnalités) et **leurs enchaînements**.

Bug ?

Un bug est un état incorrect, indésirable, inattendu d'un logiciel.

Un logiciel fonctionne correctement quand à travers ses fonctionnalités il n'arrive jamais dans un état indésirable (bug)

Méthode de production d'applications correctes

Dans la **Méthode B**, et dans les méthodes formelles en général,
- on spécifie un logiciel par son ensemble d'états corrects et
- on évite que les opérations qui décrivent les fonctionnalités aillent dans un état indésirable.

Pour éviter les états indésirables, on prend des précautions (ce sont les préconditions des opérations).

Pas de risque, on **doit respecter les préconditions**, avant de faire les opérations.

Méthode B

- (..1996) Méthode pour spécifier, concevoir et développer des systèmes informatiques séquentiels.
- (1998..) Event B ... systèmes distribués, concurrents
- (...) méthode en constante évolution, les outils (IDE) aussi
- M. **J-R. ABRIAL**

Exemples d'application dans le ferroviaire

Figure: Synchronisation ouverture portes palières - Metro Paris (13, ...)

Exemples de systèmes critiques

Un système est critique lorsque son mauvais fonctionnement ou sa défaillance ont des conséquences néfastes en termes de sécurité des personnes, des biens, de l'environnement, ...

- Pilotage par logiciel d'un avion, d'une voiture, d'une centrale nucléaire
- Pilotage d'un robot industriel,
- Contrôle d'un cœur artificiel (*pacemaker*),
- Opérations en bourse
- ...

☞ Eviter les fautes de construction du logiciel (les bugs).
Construire formellement le logiciel.

Méthodes en génie logiciel

Méthode formelle =

- Langage de spécification ou **modélisation** formelle
- Système de **raisonnement** formel

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

☞ Vous pouvez élaborer la votre !

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

Méthode B : Approche globale

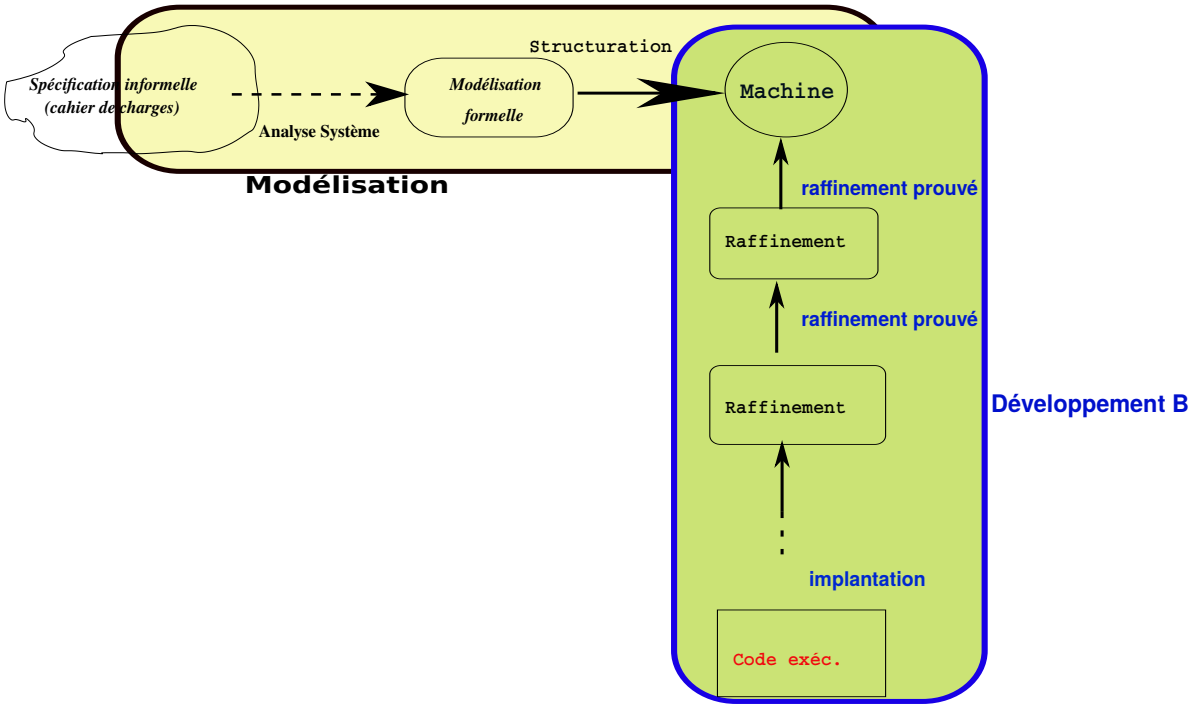

Figure: Analyse et développement B

Machine abstraite : offre des opérations

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

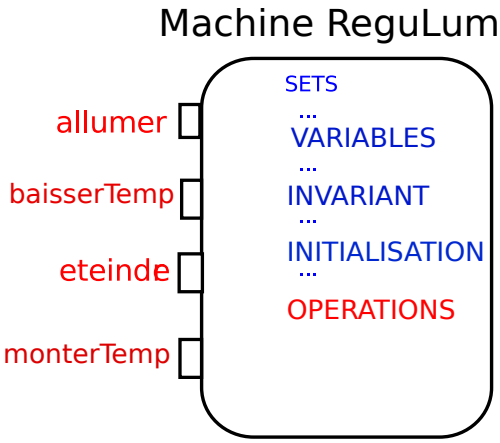

Figure: Les opérations sont appelables de l'exterieur

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

Exemple de machine abstraite : régulation lumière

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

VARIABLES
mode
, lumiere
, temp
INVARIANT
mode : MODEJ
& lumiere : ETATLUM
& temp : ENTIER
& ... /* d'autres propriétés */
INITIALISATION
mode := jour || temp := 20
|| lumiere := eteint
```

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

Machine abstraite : exemple de la jauge

Exemple

Soit à **contrôler la valeur d'une variable (jauge) entre 2 et 45** alors qu'on y effectue des opérations d'incrémentation et de décrémentation.

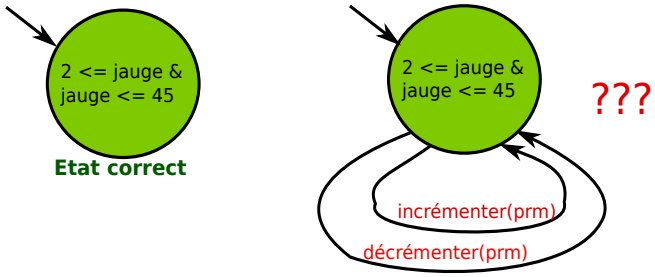

Figure: Rester dans un état correct

Exprimer les états corrects par un prédicat (invariant)

Machine abstraite : exemple de la jauge

```
MACHINE MaJauge
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 de la jauge

```
MACHINE MaJauge
VARIABLES
    jauge
INVARIANT
    jauge : NAT & ....
INITIALISATION
    jauge := 1 // quoi ???
OPERATIONS
   decrementer1 = ...
;  decrementer(pas) = ...
;  incrementer ...
;  ...
END
```

Avec la méthode B,
on transforme cette machine abstraite en programme (C) exécutable ; la transformation se fait à l'aide de raffinements.

Exemple : construction du programme de PGCD

De la machine abstraite vers son raffinement en code exécutable.

Du modèle mathématique (abstrait) → modèle informatique (concret)

Développement du PGCD : machine abstraite

```
MACHINE
    pgcd1 /* pour le PGCD de deux entiers, JCA, U. Nantes */
          /* pgcd(x,y) is d | x mod d = 0 ∧ y mod d = 0
          ∧ ∀ other divisors dx, d > dx
          ∧ ∀ other divisors dy, d > dy */
OPERATIONS
        rr <-- pgcd(xx,yy) = /* SORTIE : rr ; ENTREE xx, yy */
            ...
END
```


Développement du PGCD : machine abstraite

```
OPERATIONS
rr <-- pgcd(xx,yy) = /* spécification du pgcd */
PRE
    xx : INT & xx >= 1 & xx < MAXINT
&   yy : INT & yy >= 1 & yy < MAXINT
THEN
    ANY dd WHERE
    dd : INT
    & (xx - (xx/dd)*dd) = 0 /* d is a divisor of x */
    & (yy - (yy/dd)*dd) = 0 /* d is a divisor of y */
        /* and the other common divisors are < d */
    & !dx.((dx : INT & dx < MAXINT
        & (xx- (xx/dx)*dx) = 0 & (yy-(yy/dx)*dx)=0) => dx < dd)
    THEN rr := dd
    END
END
```

Développement du PGCD : raffinement

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


Développement du PGCD : raffinement

```
rr <-- pgcd (xx, yy) = /* opération raffinée */
    BEGIN    VAR cd, rx, ry, cr IN
             cd := 1
             ; WHILE ( cd < xx & cd < yy) DO
                 ; rx := xx - (xx/cd)*cd ; ry := yy - (yy/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
                 xx : INT & yy : INT & rx : INT & rx < MAXINT
                 & ry : INT & ry < MAXINT & cd < MAXINT
                 & xx = cr*(xx/cr) + rx   &   yy = cr*(yy/cr) + ry
             VARIANT
                 xx - cd
             END
        rr := cr    END     END
```

La méthode B

Dans la suite nous allons étudier :

- les concepts de base utilisés dans la méthode B : la logique, la théorie des ensembles, pour modéliser les données et les traitements, de façon abstraite ;
- les bases de la méthode B : espaces d'états et propriétés invariants, les substitutions généralisées pour modéliser les opérations
- les bases du raffinement

La méthode B

Concepts et principes de base :

- machine abstraite (espace d'états + opérations abstraites),
- raffinements prouvés (passer du modèle abstrait vers du concret)
  Raffinement = passage de l'abstrait au concret.
  - raffinement des données
  - raffinement des opérations

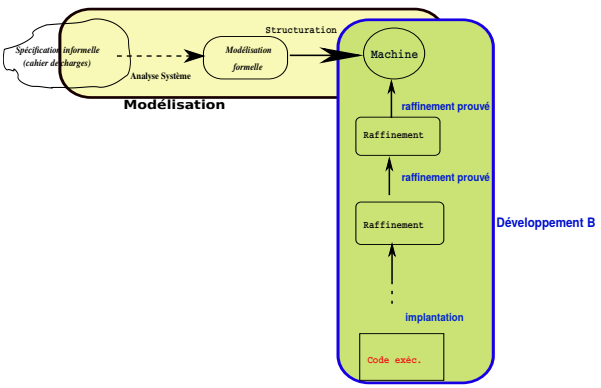

Figure: Analyse et développement B

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 $I = 2$, $I = 6$, $I = 0$, $\cdots$ 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 $\langle I,\ J\rangle$, $\langle V1,\ V2,\ V4,\ ...\rangle$

Approche de développement

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

- Décrire un espace d'états (variables + prédicats)
- Décrire des opérations qui parcourent cet espace (opérations)
- Décrire le système de transition entre états

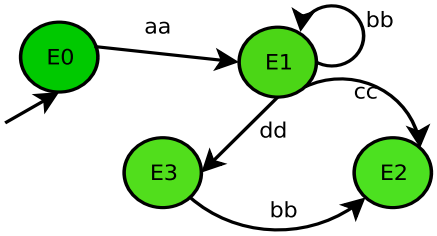

Figure: Evolution d'un système logiciel

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
```

Exemple : système de régulation de lumière

Etude

Besoins : Régulation de la lumière et de la température

- Gérer deux modes dans le système : jour, nuit
- Gérer allumage et extinction de la lumière
- Gérer la baisse et l'augmentation de la température

Exigences :

- plus tard
- ...

Machine abstraite : les clauses

```
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 ; cet état doit vérifier 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.
  Elles modélisent les changements d'état des variables par des substitutions logiques (notées :=).
  Les substitutions sont généralisées pour plus d'expressivité.
  Les opérations peuvent avoir une PREcondition (la POST est implicitement l'invariant)

Exercice : le système de régulation de lumière

Etude

Compléter le système de régulation de température et de lumière par :
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.

Interface des opérations

En B les opérations sont avec ou sans paramètres entrée/sortie.

- Aucun paramètre :
  nomOperation = ...
- Que des paramètres d'entrée (p1, p2, $\cdots$) :
  nomOperation(p1, p2, $\cdots$) = ...
- Paramètres de sortie (r1, r2, $\cdots$)
  r1, r2, $\cdots$ <-- nomOperation = ...
- Entrée et sortie
  r1, r2, $\cdots$ <-- nomOperation(p1, p2, $\cdots$) = ...

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/prédicats).
- Tant que le système est dans un de ces états, il fonctionne bien ; il faut l'y maintenir !
- 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 - Cohérence d'une machine abstraite

- Lorsqu'on modélise un système (par l'ensemble de ses états) puis qu'on explicite ses (bonnes) propriétés dans l'invariant, on doit s'assurer après, que le système ne parcourt que l'ensemble des états qui respectent les propriétés : c'est la cohérence du système : **Obligations à prouver**.
- 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. **A prouver**.

B - Cohérence d'une machine abstr. (= sémantique)

La **preuve de cohérence** d'une machine abstraite est une **obligation**. Pratiquement, la méthode B génère les obligations de preuve pour s'assurer qu'une machine est cohérente :

- l'initialisation ($Init$) établit l'invariant (Inv) :

$$[Init]Inv$$

- chaque opération ($Op = P \mid Subst$) , appelée sous sa précondition (P) préserve l'invariant:

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

B : Approche logique pour les opérations

Originalité de B : tout est en Logique (données et opérations)

- L'espace d'état : Invariant : Prédicat : $P(x, y, z)$
  Un état : une valuation des variables
  $x := v_x \quad y := v_y \quad z := v_z$ dans $P(x, y, z)$
  $\Rightarrow$ 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 $\Rightarrow$ **substitutions généralisées**

  [Dijkstra]

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 auxilliaires 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 disparaitre (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 : les 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, ...
  (disponibles librement : AtelierB, Rodin, ProB)

Méthode B : apprentissage du langage

Le langage B:

- Le langage des données (logique + théorie des ensembles)
- Le langage des opérations : logique, logique de Hoare, substitutions logiques généralisées,

Et surtout, des exemples, des exercices...la pratique

Exemple : construction du programme de PGCD

De la machine abstraite vers son raffinement en code exécutable.

Du modèle mathématique (abstrait) → modèle informatique (concret)

Développement du PGCD : machine abstraite

```
MACHINE
    pgcd1 /* pour le PGCD de deux entiers, JCA, U. Nantes */
            /* pgcd(x,y) is d | x mod d = 0 ∧ y mod d = 0
            ∧ ∀ other divisors dx, d > dx
            ∧ ∀ other divisors dy, d > dy */
OPERATIONS
        rr <-- pgcd(xx,yy) = /* SORTIE : rr ; ENTREE xx, yy */
            ...
END
```


Développement du PGCD : machine abstraite

```
OPERATIONS
rr <-- pgcd(xx,yy) = /* spécification du pgcd */
PRE
    xx : INT & xx >= 1 & xx < MAXINT
&   yy : INT & yy >= 1 & yy < MAXINT
THEN
    ANY dd WHERE
    dd : INT
    & (xx - (xx/dd)*dd) = 0 /* d is a divisor of x */
    & (yy - (yy/dd)*dd) = 0 /* d is a divisor of y */
        /* and the other common divisors are < d */
    & !dx.((dx : INT & dx < MAXINT
        & (xx- (xx/dx)*dx) = 0 & (yy-(yy/dx)*dx)=0) => dx < dd)
    THEN rr := dd
    END
END
```

Développement du PGCD : raffinement

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


Développement du PGCD : raffinement

```
rr <-- pgcd (xx, yy) = /* opération raffinée */
    BEGIN    VAR cd, rx, ry, cr IN
             cd := 1
             ; WHILE ( cd < xx & cd < yy) DO
                 ; rx := xx - (xx/cd)*cd ; ry := yy - (yy/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
                 xx : INT & yy : INT & rx : INT & rx < MAXINT
                 & ry : INT & ry < MAXINT & cd < MAXINT
                 & xx = cr*(xx/cr) + rx   &   yy = cr*(yy/cr) + ry
             VARIANT
                 xx - cd
             END
        rr := cr    END     END
```

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)
- Ensembles abstraits : PERSONNE, NUMERO, MOT, PROCESSUS, SERVICE, PRODUIT, NOM
- Ensembles énumérés : IDGROUPE = {g1, g3, g4, g2, g5}

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

Exemples de prédicats :

$2 \leq jauge \leq 45$ $2 \leq jauge \ \wedge \ jauge \leq 45$
$valReg < 255$
$\forall x.(x \in S_1 \Rightarrow temp(x) < 37)$
$\forall p.(p \in processus \Rightarrow status(p) \in \{ready, wait, active\})$
$\exists y.(y \in N \wedge y > 1024)$
$\ldots$

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 : 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	$E \cup F$	`E \/ F`
intersection	$E \cap F$	`E /\ F`
appartenance	$x \in F$	`x:F`
différence	$E \setminus F$	`E - F`
inclusion	$E \subseteq F$	`E <: F`
sélection	`choice(E)`	
cardinal	card(E)	card(E)

+ Union et intersection généralisées
+ Union et intersection quantifiées

Exemple d'utilisation des ensembles

```
MACHINE
    Resrc
SETS
    RESC // un ensemble abstrait
CONSTANT
    maxRes
PROPERTIES
     maxRes : NAT & maxRes > 1
VARIABLES
     rsc
INVARIANT
     rsc <: RESC // sous
ens-ensemble
    &  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
```

B - Langage des données

En notation ascii la négation est réalisée avec /.

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

Union généralisée

On veut écrire $A \cup C \cup B \cup E$, pour regrouper les ensembles

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

Exemple

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

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

$$UNION(x).(x \in \{1, 2, 3\} \mid \{y \mid y \in NAT \wedge y = x * x\})$$

$$= \{1\} \cup \{4\} \cup \{9\} = \{1, 4, 9\}$$

Intersection généralisée

On veut écrire $A \cap C \cap B \cap E$, pour calculer l'intersection des ensembles.

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

$\Rightarrow$

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

Exemple

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

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

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

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

Les relations - définition, vocabulaire

définition : relation

Une relation r entre D et A est un sous-ensemble du produit $D \times A$

On note $r : D \leftrightarrow A$ ou bien $r \subseteq D \times A$

r est un ensemble de couples (d, a) encore noté $d \mapsto a$

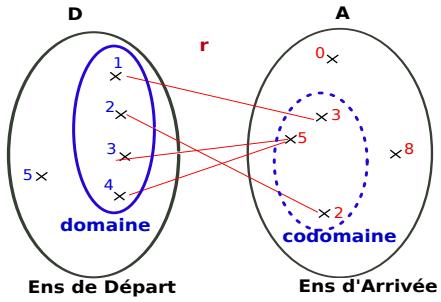

Figure: Diagramme sagittal de r

$r = \{(1, 3), (2, 2), (3, 5), (4, 5)\}$ ou
$r = \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 5, 4 \mapsto 5\}$

$dom(r) = \{1, 2, 3, 4\}$
$ran(r) = \{3, 5, 2\}$

Domaine : domaine
Codomaine : range

Les relations

Désignation	Notation	Ascii
relation	$r : S \leftrightarrow T$	r : S <-> 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)

l'image d'un élément : $r(element)$
les images d'un ens. d'éléments : $r[Ensemble]$

Notez que r étant un ensemble (de couples), on peut lui appliquer tous les opérateurs sur les couples : $\cup, \cap, \setminus, \subset, \in$, card, ...

Les relations (suite)

Désignation	Notation	Ascii
restrictition domaine	$S \lhd r$	S <| r
restriction codomaine	$r \rhd T$	r |> T
antirestriction domaine	$S \lhd r$	S <<| r
antirestriction codomaine	$r \rhd T$	r |>> T
inverse	$r^{\sim}$	r ~
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 relations (suite)

$annuaire$: $personnes$<->$numeros$

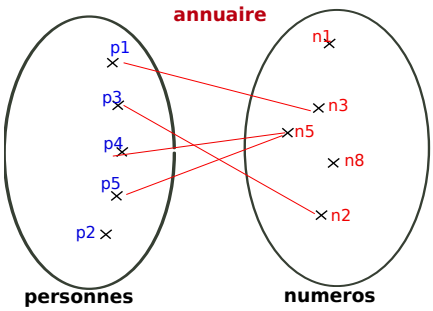

Figure: Relation *annuaire*

$annuaire =$
$\{(p1, n3), (p3, n2), (p4, n5), (p5, n5)\}$ ou
$annuaire =$
$\{p1 \mapsto n3, p3 \mapsto n2, p4 \mapsto n5, p5 \mapsto n5\}$

$\{p3, p5\} <|$ *annuaire* = {p3 ↦ n2, p5 ↦ n5}
annuaire |> { n5 } = { p4 ↦ n5, p5 ↦ n5 }

Les relations (suite) - exemple en B

```
MACHINE
     ExpRelation   // exemple de machine avec une relation
DEFINITIONS
     NUMERO == 10..100 // un ens. défini l'interv sur 10..100
SETS
     ETUDIANT // un ens. abstrait
VARIABLES
     personnes // un ens. d'étudiants
,    annuaire // annuaire des étudiants
INVARIANT
     personnes <: ETUDIANT // un ens d'étudiants
&    annuaire : personnes <-> NUMERO // un étu a 0/plus numéros
INITIALISATION
     personnes, annuaire := {}, {} // tout est vide au début
END
```

Les fonctions

définition : fonction

Une fonction f est une relation avec des propriétés particulières : chaque élément de l'ensemble de départ n'a au plus qu'une image.

Désignation	Notation	Ascii
fonction partielle	$f : S \nrightarrow T$	f: S +-> T
fonction totale	$f : S \rightarrow T$	f: S --> T
injection partielle	$f : S \rightarrowtail T$	f: S >+> T
injection totale	$f : S \rightarrowtail T$	f: S >-> T
surjection partielle	$f : S \twoheadrightarrow T$	f: S +->> T
surjection totale	$f : S \twoheadrightarrow T$	f: S -->> T
bijection totale	$f : S \rightarrowtail T$	f: S >->> T
lambda abstraction	$\%x.(P \mid E)$	

Fonctions : exemple

```
MACHINE
     ExpFonction   // exemples de fonctions
DEFINITIONS
     PLACE == 1..200 // un ens. défini l'interv sur 1..200
SETS
     ETUDIANT
VARIABLES
     placeEtu1 // une fonction partielle
,    placeEtu2 // une injection (pas de partage d'images)
,    placeEtu3, placeEtu4 // fonction totale
INVARIANT
     placeEtu1 : PLACE +-> ETUDIANT // à une place, un étudiant
&    placeEtu2 : PLACE >+> ETUDIANT // c'est bien mieux
&    placeEtu3 : PLACE --> ETUDIANT // à toute place, un étudiant
&    placeEtu4 : PLACE >-> ETUDIANT // à tte place,un étu différent
...
END
```

Les séquences

Les séquences sont des fonctions (le domaine est un intervalle continu d'entiers) : `maSeq1 : 1..N --> NAT`

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	$s \uparrow n$
élémination de n premiers éléments de s	$s \downarrow n$

Le langage des opérations

Après le langage des données,
le concept de base des substitutions
et le langage pour les opérations

Le langage des opérations

Concepts de base pour la partie dynamique :

- Logique de Hoare
- Plus faibles préconditions (Dijkstra)
- Substitutions (Logique) et Substitutions généralisées (extension)

Les plus faibles préconditions

Contexte : *Logique de Hoare/Floyd/Dijkstra* triplet de Hoare
(Etat, espace d'état, commandes, triplet de Hoare, ACM, 1969)

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

P un prédicat : précondition,
S une **commande** (instruction ou programme) et
R un **prédicat décrivant le résultat de** S : post-condition.
$wp(S,R)$, prédicat qui représente :
l'ensemble de tous les états tels que l'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 $z = max(x,y)$

$$wp(S,R) \ = \ Vrai$$

Plus-faibles préconditions - sémantique

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 préconditions - sémantique

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 etablir, 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	$S [] T$	$[S [] T]R$	$[S]R \wedge [T]R$
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 specification de B mais ...

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 || T`

B - Langage des substitutions généralisées

Substitution neutre

skip

Extension syntaxique

skip

Subst. avec précondition

$P \mid S$

Extension syntaxique

```
PRE
P
THEN
S
END
```

B - Langage des substitutions généralisées

choix borné

$S \; [] \; T$

Extension syntaxique

```
CHOICE
S
OR
T
END
```

Substitution avec garde

$(P \Longrightarrow T) \; [] \; (\neg \, P \Longrightarrow S)$

Extension syntaxique

```
IF P
THEN T
ELSE S
END
```


B - Langage des substitutions généralisées

Substitution de choix non borné

$@x.S_x$

Extension syntaxique

```
VAR x IN
Sx
END
```

Extension du jeu de base : non-déterminisme

Nondéterminisme @

$@x.(P_x \implies S_x)$

Extension syntaxique

```
ANY x
WHERE Px
THEN Sx
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
```


Non déterminisme - Substitutions

- **Abstraction** ⇒ non déterminisme possible.
  c'est OK pour spécifier, mais ensuite il faut déterminiser
- **Concrétisation** ⇒ raffinement vers code
  avec des substitutions de programmation (séquences, boucles, ...)
- Extension du jeu de base à d'autres substitutions proches de la programmation.
  Substitutions de programmation (dans les raffinements)
  - ... ; ... séquence
  - IF ... THEN ...ELSE...
  - CASE OF ...
  - SELECT ...

Références

- J-R. Abrial, The B-Book, Cambridge University Press, 1996
- J. Wordsworth, Software Engineering with B, Addison-Wesley, 1996
- J-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press
- H. Habrias, Spécification formelle avec B, Hermès - Lavoisier, 2001