

Spécifications Formelles, M1 2014-2015

Serenella Cerrito et Francesco Belardinelli

Laboratoire Ibisc, Université d'Evry Val d'Essonne, France

2014-2015

Introduction, 1

- ▶ Spécification formelle d'un logiciel = description précise mais abstraite de ce que le logiciel **doit faire**.
- ▶ Aide au développement.
- ▶ **But** : Vérifier **formellement** (en partie automatiquement) la correction de la conception
- ▶ **But** : La corriger, par étapes, avant d'investir dans l'implémentation.
- ▶ Résultat final : implémentation correcte par construction.
- ▶ **Un** exemple concret d'application : **ligne 14 du métro parisien**

Introduction, 2

- ▶ Plusieurs méthodes de spécification formelle, fondées sur :
automates, réseaux de Petri, logiques (équationnelles, des
prédicats, temporelles etc.)
- ▶ Ce cours : **la méthode B** (utilisée pour la ligne 14 du metro).
 - ▶ Fondée sur la notion de **machine abstraite**
 - ▶ Notation **AMN** (*Abstract Machine Notation*)
 - ▶ **Abstraction, modularité.**
 - ▶ AMN contient des notations **ensemblistes** et **logiques** (**logique
(classique) des prédicats**).

Abstract Machine

Une **machine abstraite (AM)** est une **spécification** d'une partie du système (logiciel) à réaliser.

Boîte noire qui devra se composer avec d'autres boîtes (modules).

Décrit les opérations que cette partie du logiciel doit effectuer, leur entrées, leur effets, etc.

Format :

MACHINE ...

VARIABLES...

INVARIANT...

INITIALISATION..

OPERATIONS...

END

Abstract Machine : MACHINE

Ici, le **nom** de la machine.

Ex. : une machine **Ticket** qui distribue des billets (dans un supermarché, ou à la poste) pour ordonner la queue des clients.

Vague spécification : A l'entrée, chaque client prend un ticket avec un nombre. Un écran montre le nombre du prochain client à servir.

MACHINE Tickets

VARIABLES...

INVARIANT...

INITIALISATION...

OPERATIONS...

END

Abstract Machine : VARIABLES

Déclaration des variables qui décrivent l'état courant de la machine.

Pas ici : ni les types des variables, ni d'autres informations.

Dans notre exemple, deux variables

serve : le numero du client à servir (montré sur l'écran)

next : le numéro du prochain ticket à donner.

MACHINE Tickets

VARIABLES *serve*, *next*

INVARIANT...

INITIALISATION...

OPERATIONS...

END

Abstract Machine : INVARIANT 1

INVARIANT : un énoncé qui donne le **type des variables** de la machine et une **propriété** des valeurs des variables qui doit toujours rester vraie, pendant l'exécution.

Abstract Machine : INVARIANT 2

Formes possible d'une déclaration du type d'une variable x dans l'invariant :

- ▶ $x \in TYPE$ (*la valeur de x appartient à l'ensemble $TYPE$*)
- ▶ $x \subseteq TYPE$ (*la valeur de x est un sous-ensemble de l'ensemble $TYPE$*)
- ▶ $x = \text{expression}$

Exemple machine Tickets : $serve \in \mathbb{N}$ et $next \in \mathbb{N}$.

Abstract Machine : INVARIANT 3

Expression de la condition qu'une propriété P doit rester toujours vraie : par une formule logique.

Exemple machine Tickets : $\text{serve} \leq \text{next}$.

Abstract Machine : INVARIANT 4

L'invariant est donc une formule logique complexe.

MACHINE Tickets

VARIABLES *serve*, *next*

INVARIANT $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$

INITIALISATION...

OPERATIONS...

END

Abstract Machine : OPERATIONS, 1

Liste de déclarations de la forme :

outputs $\leftarrow$ *name* (*inputs*)

name = nom de l'opération

inputs = liste des variables d'entrée

outputs = liste des variables de sortie.

Inputs et/ou outputs : optionnels.

MACHINE Tickets

VARIABLES *serve*, *next*

INVARIANT *serve* $\in \mathbb{N}$ $\wedge$ *next* $\in \mathbb{N}$ $\wedge$ *serve* $\leq$ *next*

INITIALISATION...

OPERATIONS

ss $\leftarrow$ *serve* *_next*

tt $\leftarrow$ *take* *_next* **END**

NB : Pas d'inputs, ici !

Abstract Machine : OPERATIONS, 2

Spécifier une opération = indiquer :

sa **precondition**

son **body** = effet sur les variables outputs et, éventuellement, mise à jour de l'état de la machine.

Pour `serve_next` :

MACHINE *Tickets*

VARIABLES *serve, next*

INVARIANT $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$

INITIALISATION...

OPERATIONS

$ss \leftarrow serve_next \hat{=}$

PRE $serve < next$ (*precondition*)

THEN $ss, serve := serve + 1, serve + 1$ (*body*)

END

$tt \leftarrow take_next$

END

Abstract Machine : OPERATIONS 3

Modification de Tickets et violation de l'invariant :

MACHINE Tickets

VARIABLES *serve, next*

INVARIANT $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$

INITIALISATION...

OPERATIONS

$ss \leftarrow serve_next \hat{=}$

PRE *True* (*precondition trop faible !*)

THEN $ss, serve := serve + 1, serve + 1$ (*body*)

END

$tt \leftarrow take_next$

END

Abstract Machine : OPERATIONS 4

Complétons avec `take_next` la machine correcte :

MACHINE *Tickets*

VARIABLES *serve, next*

INVARIANT $serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next$

INITIALISATION...

OPERATIONS

ss $\leftarrow$ *serve_next* $\hat{=}$

PRE $serve < next$ (*precondition*)

THEN $ss, serve := serve + 1, serve + 1$ (*body*)

END

tt $\leftarrow$ *take_next* $\hat{=}$

PRE *True* (*precondition*)

THEN $tt, next := next, next + 1$ (*body*)

END

NB : **PRE** *true* peut même être omise

Abstract Machine : INITIALISATION

Valeurs initiales pour les variables de VARIABLES $\rightsquigarrow$ l'état initial de la machine.

MACHINE Tickets

VARIABLES *serve, next*

INVARIANT *serve* $\in \mathbb{N} \wedge$ *next* $\in \mathbb{N} \wedge$ *serve* $\leq$ *next*

INITIALISATION *serve, next* $:= 0, 0$ **OPERATIONS**

ss $\leftarrow$ *serve_next* $\hat{=}$

PRE *serve* $<$ *next* (*precondition*)

THEN *ss, serve* $:=$ *serve* $+ 1$, *serve* $+ 1$ (*body*)

END

tt $\leftarrow$ *take_next* $\hat{=}$

PRE *True* (*precondition*)

THEN *tt, next* $:=$ *next*, *next* $+ 1$ (*body*)

END

Utilisent des notations ensemblistes et des notations logiques

Notations Ensemblistes

$$\text{PETITS_PAIRS} = \{0, 2, 4, 6\}$$

$$\text{PETITS_PAIRS} = \{x \mid x \in \mathbb{N} \text{ et } x \text{ est pair et } x < 8\}$$

$$\text{DE_0_A_5} = 0..5 = \{0, 1, 2, 3, 4, 5\}.$$

Pour E et E' qui sont des ensembles :

$$E \subset E', E \subseteq E'$$

$$E \cup E', E \cap E', E - E'$$

$$E \times E', \mathcal{P}(E)$$

$$|E| = \text{card}(E) = \text{taille de } E$$

Logique des prédicats (LP), 1

- ▶ Alphabet pour les TERMES de LP :
 - un ensemble infini dénombrable de variables :
 $X = \{x_1, x_2, x_3, \dots\}$,
 - un ensemble C de constantes,
 - un ensemble F de symboles de fonction, chacun muni de son nombre $n > 0$ d'arguments,
 - les parenthèses $()$ et la virgule.
- ▶ Grammaire pour le langage des termes :

$$T := v \mid c \mid f(\underbrace{T, \dots, T}_n)$$

où $v \in X$, $c \in C$, $f \in F$ et a n arguments, et chaque argument T est un terme.

NB : définition récursive.

Notation infixe : si $f \in F$ a 2 arguments, on écrira $x f y$ à la place de $f(x,y)$.

Logique des prédicats (LP), 2

- ▶ Alphabet pour les **FORMULES ATOMIQUES** de LP :
on ajoute à l'alphabet des termes un ensemble R de symboles de relation (ou prédicats), chacun muni de son nombre m d'arguments
- ▶ Grammaire pour le langage des formules atomiques (atomes) :

$$ATOMES := True \mid False \mid r(\underbrace{T, \dots, T}_m)$$

où $r \in R$ et a m arguments, et chaque argument T est un terme.

Notation infixe : si $r \in R$ a 2 arguments, on écrira $x \ r \ y$ à la place de $r(x,y)$

Logique des prédicats (LP), 3

- ▶ Alphabet pour les FORMULES de LP :
on ajoute à l'alphabet des formules atomiques les connecteurs booléens $\neg, \wedge, \vee, \Rightarrow$ et les quantificateurs : existentiel $\exists$ et universel $\forall$.
- ▶ Grammaire pour le langage des formules :

$$F := A \mid (\neg F) \mid (F \wedge F) \mid (F \vee F) \mid (F \Rightarrow F) \mid (Qv F)$$

où $A \in ATOMES$, $v \in X$, $Q \in \forall, \exists$ est un quantificateur et F est une formule.

NB : définition récursive.

Logique des prédicats (LP), 4

Exemples de formules de la logique des prédicats et leur
sémantique :
au tableau.

Logique des prédicats (LP), 5

Etant donnée une formule de la forme $\exists x_i F$ ou $\forall x_i F$, F est la **portée** du quantificateur.

Une occurrence d'une variable est dite **libre** si elle n'est pas dans la portée d'un quantificateur, et elle dite **liée** (par le quantificateur) sinon.

Ex.: pour la formule $(\forall x_1(\exists x_2(x_1 < x_2))) \wedge \text{pair}(x_1)$,
la première occurrence de x_1 est liée par la quantification $\forall x_1$,
la deuxième occurrence de x_1 est libre.

Les occurrences des variables **liées** peuvent tre renommées :

$$(\forall x_1(\exists x_2(x_1 < x_2))) \wedge \text{pair}(x_1) \equiv (\forall x_3(\exists x_2(x_3 < x_2))) \wedge \text{pair}(x_1)$$

Logique des prédicats (LP) 5

Conventions pour épargner des $()$:

1. Droit de ne pas écrire les $()$ le plus extérieurs.
2. Droit d'écrire $A \wedge B \wedge C$ à la place de $A \wedge (B \wedge C)$ ou $(A \wedge B) \wedge C$ (et idem pour le $\vee$). Justification : associativité des fonctions booléennes $\wedge$ et $\vee$
3. Droit d'écrire $Q_1 v_1 Q_2 v_2 F$ à la place de $Q_1 v_1 (Q_2 v_2 F)$, où Q_i est un quantificateur.

Par ex., droit d'écrire $\forall x_1 \exists x_2 x_1 < x_2$ à la place de $\forall x_1 (\exists x_2 x_1 < x_2)$.

Substitution, 1

Si E est une expression (un terme) et F est une formule, la formule obtenue à partir de F en remplaçant toute occurrence libre de la variable v par E est notée : $F[E/v]$. On lit : “ F avec E à la place de la variable v .”

Ex. : pour $F = x_1 < x_2$ (où $<$ est un symbole de relation à 2 arguments), on a :

$$F[2/x_1] = x_1 < x_2[2/x_1] = 2 < x_2$$

La notation “substitutions multiples” est permise :

$$F[2, 3/x_1, x_2] = x_1 < x_2[2, 3/x_1, x_2] = 2 < 3$$

NB : substitutions en //.

Substitution,2

$$\begin{aligned} members &\subseteq PERSON[(members \cup new)/members] = \\ members \cup new &\subseteq PERSON \end{aligned}$$

$$\begin{aligned} \exists p(p \in PERSON \wedge age(p) > limit)[oldlimit + 2/limit] &= \\ \exists p(p \in PERSON \wedge age(p) > oldlimit + 2) \end{aligned}$$

$$\begin{aligned} \text{NB : } \exists n(n \in \mathbb{N} \wedge n > limit)[n + 3/limit] &\neq \\ \exists n(n \in \mathbb{N} \wedge n > n + 3) ! \end{aligned}$$

Pourquoi ?

Capture de variables libres!

Comment réécrire ? $\exists m(m \in \mathbb{N} \wedge m > n + 3)$

Espace d'états

C : collection de variables, avec leur types. L'**espace d'états** associé à C est l'ensemble des combinaisons que ces variables peuvent prendre.

Par ex. si $C = \{x, y\}$ avec types : $x \in \{0, 1, 2\}$ et $y \in \{0, 1, 2\}$, l'espace d'états associé contient 9 couples.

Un énoncé AMN peut décrire une transformation d'états. Ex : l'affectation $y := \max\{0, y - x\}$ associe à chacun de ces 9 états un nouvel état (figure au tableau).

Postcondition

Si P est une formule qui décrit un ensemble d'états qui peuvent être atteints après l'exécution d'un énoncé AMN S (une instruction), alors P est une **postcondition** de S .

Weakest Precondition, 1

Soit P' une postcondition qu'on veut obtenir après l'exécution d'un S (par ex. un invariant).

Il est important d'identifier quel est l'ensemble le plus large d'états qui **assurent** que l'exécution de S conduira à des états vérifiant P .

La notation $[S]P$ indique une formule qui décrit cet ensemble d'états. C'est la *weakest precondition* (la condition la plus faible) afin que S puisse assurer P : elle vaut pour tous les états qui certainement arriveront à P par le biais de S !

Weakest Precondition, 2

Dans l'exemple précédent de $C = \{x, y\}$ avec $x, y \in \{0, 1, 2\}$:

$$\underbrace{[y := \max\{0, y - x\}]}_S \underbrace{(y > 0)}_P =$$

$$(x = 0 \wedge y = 1) \vee (x = 0 \wedge y = 2) \vee (x = 1 \wedge y = 2)$$

NB :

$(x = 0 \wedge y = 1) \vee (x = 0 \wedge y = 2)$ serait une precondition assurant P , mais pas la plus faible ! Elle interdirait l'état initial $(x = 1 \wedge y = 2)$.

En revanche, *True* serait une precondition trop faible et elle n'assurerait pas P .

Lois pour $[S]P$

- ▶ $[S](P \wedge Q) = [S]P \wedge [S]Q$
- ▶ $[S]P \vee [S]Q \Rightarrow [S](P \vee Q)$ est valide

NB : L'implication réciproque ne l'est pas !

Penser, intuitivement, à : S = lancer une pièce, P = obtenir pile, Q = obtenir face. Alors $[S](P \vee Q) = \text{True}$, car $P \vee Q = \text{True}$, mais $[S](P) \neq \text{True}$ et $[S](Q) \neq \text{True}$!

- ▶ Si $P \Rightarrow Q$ est vraie, alors $[S]P \Rightarrow [S]Q$ aussi.

NB : Pas vrai que $[S]P = [S]Q$. Il y a juste implication.

Calcul de la weakest precondition

Etant donné S et P on a des règles permettant de **calculer** $[S]P$

R1 = règle pour l'affectation :

$$[x := E]P = P[E/x]$$

Exemple : machine Ticket, P est $serve \leq next$ (partie de l'invariant), S est $serve := serve + 1$.

$[S]P =$

$[serve := serve + 1](serve \leq next) =$ (par R1)

$(serve \leq next)[serve + 1/serve] =$ (substitution)

$serve + 1 \leq next$

qui, par l'arithmétique, est équivalent à : $serve < next$.

Calcul de la weakest precondition

R1G : Généralisation de R1 aux **affectations multiples** (en //)

$$[x_1, \dots, x_n := E_1, \dots, E_n]P = P[E_1, \dots, E_n/x_1, \dots, x_n]$$

où si $i \neq j$ alors $x_i \neq x_j$

Exemple

$$[serve, next := serve + 1, next - 1](serve \leq next) = \text{ (R1G)}$$

$$(serve + 1 \leq next - 1) = \text{c'à.d. :}$$

$$serve + 2 \leq next$$

Signification : l'invariant est préservé à coup sûr par cette affectation seulement si on part d'un état où $serve + 2 \leq next$ est vrai.

Calcul de la weakest precondition

L'expression AMN *skip* = affectation vide (**ne rien faire**).

R2 : Règle pour *skip*

$$[skip]P = P$$

Calcul de la weakest precondition

R3= règle pour le conditionnel :

S a la forme *IF E THEN S1 ELSE S2 END*, où E est une formule qui s'évalue à vrai ou faux, et $S1$, $S2$ sont elles-mêmes des instructions.

$$[IF\ E\ THEN\ S1\ ELSE\ S2\ END] = (E \wedge [S1]P) \vee (\neg E \wedge [S2]P)$$

Calcul de la weakest precondition

Exemple pour R3 combinée avec R1

$$\underbrace{[IF\ x < 5\ THEN\ x := x + 4\ ELSE\ x := x - 3\ END]}_S (x < 7) =_{R3}$$

$$(x < 5 \wedge \underbrace{[x := x + 4]}_{S1} \underbrace{(x < 7)}_P) \vee (\neg(x < 5) \wedge \underbrace{[x := x - 3]}_{S2} \underbrace{(x < 7)}_P) =_{R1}$$

$$(x < 5 \wedge (x + 4) < 7) \vee (x \geq 5 \wedge (x - 3) < 7) = (\text{arithm})$$

$$(x < 5 \wedge x < 3) \vee 5 \leq x < 10 \text{ c.à.d}$$

$$x < 3 \vee 5 \leq x < 10$$

Calcul de la weakest precondition

Cas particulier de R3

$$[IF\ E\ THEN\ S\ END]P \ = \ (E \wedge [S]P) \vee (\neg E \wedge P)$$

Calcul de la weakest precondition

Instruction *CASE OF*. Format

```
CASE E OF  
EITHER  $e_1$  THEN  $T_1$   
OR  $e_2$  THEN  $T_2$   
OR...  
OR  $e_n$  THEN  $T_n$   
ELSE V  
END
```

NB : *ELSE* optionnel : si absent, ne pas changer l'état si aucun des n cas est applicable. **Exemple** :

```
CASE dir OF  
EITHER north THEN  $partner := south$   
OR south THEN  $partner := north$   
OR east THEN  $partner := west$   
OR west THEN  $partner := east$   
END
```

Calcul de la weakest precondition

Instruction *CASE OF* : et si nombre infini de cas ? Couvrir les $n + i$ cas avec le *ELSE*.

Exemple

```
CASE sizeoforder OF  
EITHER 0 THEN discount := 0  
OR 1 THEN discount := 0  
OR 2 THEN discount := 5  
OR 3 THEN discount := 10  
ELSE discount := 15  
END
```

Calcul de la weakest precondition

R4 : Règle pour CASE OF

$$\left[\begin{array}{llll} \text{CASE} & E & \text{OF} & \\ \text{EITHER} & e_1 & \text{THEN} & T_1 \\ \text{OR} & e_2 & \text{THEN} & T_2 \\ \text{OR} & \dots & & \\ \text{EITHER} & e_n & \text{THEN} & T_n \\ \text{ELSE} & V & & \\ \text{END} & & & \end{array} \right] P =$$

$$(E = e_1 \Rightarrow [T_1]P) \wedge$$

$$(E = e_2 \Rightarrow [T_2]P) \wedge$$

$$\dots \wedge$$

$$(E = e_n \Rightarrow [T_n]P) \wedge$$

$$((E \neq e_1 \wedge E \neq e_2 \wedge \dots \wedge E \neq e_n) \Rightarrow [V]P)$$

Calcul de la weakest precondition

$$[BEGIN\ S\ END] = [S]P$$

C'est juste une notation permettant de mettre des parenthèses !

Cohérence d'une AM

La AM **M** est-elle cohérente ?

Si oui, certaines **conditions** doivent être respectées.

Chacune engendre une *obligation de preuve*.

Format de M considere ci-dessous :

MACHINE M

VARIABLES v

INVARIANT I

INITIALISATION T

OPERATIONS

$y \leftarrow op(x) =$

PRE P

THEN S

END;

...

END

Cohérence d'une AM

Il faut que l'invariant I soit satisfiable : des états légitimes de la machine pour lesquels I est vrai **doivent exister**.

C_INV : $\exists v$ I doit être vraie.

(Variables v : celles déclarées dans la partie "VARIABLES" de M)

Obligation de preuve de C_INV : revient à prouver que des valeurs adéquates pour les v existent.

Cohérence d'une AM

Exemple de la AM **Tickets** :

C_INV :

$\exists serve \exists next (serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next)$
doit être un énoncé valide de l'arithmétique (standard).

Obligation de preuve : vérifier que $n, m \in \mathbb{N}$ tels que $n \leq m$ existent.

Ici, banal : par ex., $n = m = 0$.

NB : En général, utilité de méthodes de preuve automatique ou des assistants de preuve !

Cohérence d'une AM

Il faut que l'état T déclaré (par une affectation) dans la partie INITIALISATION de M respecte l'invariant I

$C_INIT : [T]I$ doit être toujours vraie.

Obligation de preuve : prouver la formule $[T]I$.

Exemple pour Tickets

$$\begin{aligned} [T]I &= \\ [serve, next := 0, 0](&serve \in \mathbb{N} \wedge next \in \mathbb{N} \wedge serve \leq next)) = \\ (\text{par R1}) & \\ (0 \in \mathbb{N} \wedge 0 \in \mathbb{N} \wedge 0 \leq 0) & \end{aligned}$$

OK !

Cohérence d'une AM

Deux autres initialisations possibles (?) pour Tickets.

1. $T = \text{serve}, \text{next} := 1, 0$. Alors : $[T]I = (0 \in \mathbb{N} \wedge 1 \in \mathbb{N} \wedge 1 \leq 0)$. **False !**
2. $T = (\text{serve}, \text{next} := \text{serve} + 1, \text{next} + 1)$. Alors : $[T]I = (\text{serve} + 1 \in \mathbb{N} \wedge \text{next} + 1 \in \mathbb{N} \wedge \text{serve} + 1 \leq \text{next} + 1) = ??$

Signification ? Tickets démarre dans un état aléatoire, avant de faire T ! Donc **faux** (cas ci-dessus, par ex.).

Coherence d'une AM

Soit op une opération avec $PRE=P$ et $BODY=S$.

On effectue S seulement si P .

On suppose que l'invariant I est vrai dans l'état où on effectue I ;

Sous ces conditions, effectuer op se réduit à faire S , et cette action doit préserver I .

$C_OP : (I \wedge P) \Rightarrow [S]I$ doit être toujours vraie.

$\rightsquigarrow$ obligation de prouver $(I \wedge P) \Rightarrow [S]I$

Cohérence d'une AM

Exemple de C_OP pour Tickets, avec $op = \text{serve_next}$.

Il faut prouver :

$$\underbrace{((\text{serve} \in \mathbb{N} \wedge \text{next} \in \mathbb{N} \wedge \text{serve} \leq \text{next}))}_I \wedge \underbrace{\text{serve} < \text{next}}_P$$

$\Rightarrow$

$$\underbrace{[ss, \text{serve} := \text{serve} + 1, \text{next} + 1](\text{serve} \in \mathbb{N} \wedge \text{next} \in \mathbb{N} \wedge \text{serve} \leq \text{next})}_{[S]I}$$

c.à.d., en utilisant R1 pour calculer $[S]I$:

$$\underbrace{((\text{serve} \in \mathbb{N} \wedge \text{next} \in \mathbb{N} \wedge \text{serve} \leq \text{next}))}_I \wedge \underbrace{\text{serve} < \text{next}}_P$$

$\Rightarrow$

$$(\text{serve} + 1 \in \mathbb{N} \wedge \text{next} \in \mathbb{N} \wedge \text{serve} + 1 \leq \text{next}).$$

Un théorème arithmétique !

Cohérence d'une AM

Encore un exemple de AM, qui doit être cohérente.

MACHINE PaperRounds

VARIABLES *papers, magazines*

INVARIANT $papers \subseteq 1..163 \wedge magazines \subseteq papers \wedge card(papers) \leq 60$

INITIALISATION *papers, magazines* := $\emptyset, \emptyset$

OPERATIONS

addpaper (*hh*) $\hat{=}$ % pas de output
PRE $hh \in 1..163 \wedge card(papers) < 60$
THEN *papers* := *papers* $\cup$ {*hh*}
END;

addmagazine (*hh*) $\hat{=}$ % pas de output
PRE $hh \in papers$
THEN *magazines* := *magazines* $\cup$ {*hh*}
END;

remove (*hh*) $\hat{=}$ % pas de output
PRE $hh \in 1..163$
THEN *papers, magazines* := *papers* - {*hh*}, *magazines* - {*hh*}
END;

END

Cohérence d'une AM

Obligation de preuve pour C_INV . Prouver :

$$\exists papers \exists magazines (\\ papers \subseteq 1..163 \wedge \\ magazines \subseteq papers \wedge \\ card(papers) \leq 60)$$

Vrai, car : $\emptyset \subseteq 1..163 \wedge \emptyset \subseteq papers \wedge card(\emptyset) \leq 60$

ce qui prouve aussi C_INIT .

Cohérence d'une AM

Obligation de preuve pour C_OP , avec $op = addpapers$. Prouver :

$$\underbrace{((papers \subseteq 1..163 \wedge magazines \subseteq papers \wedge card(papers) \leq 60)) \wedge}_{I} \underbrace{hh \in 1..163 \wedge card(papers) < 60)}_{P}$$

$\Rightarrow$

$$papers \cup \{hh\} \subseteq 1..163 \wedge magazines \subseteq papers \cup \{hh\} \wedge card(papers \cup \{hh\}) \leq 60)$$

où R1 a été utilisée pour simplifier le conséquent de l'implication, c.à.d. $[S]I$.

Théorème arithmétique (et ensembliste) !

Cohérence d'une AM

De façon analogue, il faut analyser addmagazine et remove.

Cohérence d'une AM

Raisons possible de l'incohérence d'une AM :

1. La PRE d'une opération est trop faible.
2. Le BODY est incorrect (ne preserve pas l'invariant désiré).
3. C'est I qui est trop restrictive (exclut des états OK).
4. Au contraire, I est trop permissive : permet d'atteindre des "mauvais" états.

Modification de I à cause de 3 ou 4 : il faut refaire toutes les obligations de preuve.

Machines plus expressives

On va étudier l'usage de :

- ▶ **paramètres** $\leadsto$ machines **génériques**
- ▶ **constantes** (analogues aux constantes d'un programme)
- ▶ **ensembles** $\leadsto$ types abstraits

dont l'implémentation va être différée.

On va utiliser l'exemple d'une machine Club.

Machine Club

MACHINE Club (*capacity*)

CONSTRAINTS $capacity \in \mathbb{N}_1 \wedge capacity \leq 4096$

SETS $REPORT = \{yes, no\}$; NAME

CONSTANTS *total*

PROPERTIES $card(NAME) > capacity \wedge total \in \mathbb{N}_1 \wedge total > 4096$

VARIABLES *member, waiting*

INVARIANT

$member \subseteq NAME \wedge waiting \subseteq NAME \wedge member \cap waiting = \emptyset$

$\wedge card(member) \leq 4096 \wedge card(waiting) \leq total$

INITIALISATION $member, waiting := \emptyset, \emptyset$

OPERATIONS

$join(nn) \hat{=}$

PRE $nn \in waiting \wedge card(member) < capacity$

THEN $member, waiting := member \cup \{nn\}, member \setminus \{nn\}$

END ;

$join_queue(nn) \hat{=}$

PRE $nn \in NAME \wedge nn \notin member \wedge nn \notin waiting \wedge card(waiting) < total$

THEN $waiting := waiting \cup \{nn\}$

END ;

$remove(nn) \hat{=}$

PRE $nn \in member$

THEN $member := member \setminus \{nn\}$

END ;

$semi_reset \hat{=} member, waiting := \emptyset, member$;

$ans \leftarrow query_membership \hat{=}$

PRE $nn \in \overline{NAME}$

THEN

IF $nn \in member$

THEN $ans := yes$

ELSE $ans := no$

END

END

END

Paramètres, 1

MACHINE Club (*capacity*)

capacity est un paramètre

Les paramètres sont donnés après le nom de la machine.

Deux sortes de paramètres : à valeur scalaire, comme pour Club, ou ensembliste (alors en majuscules) :

```
MACHINE Store (ITEM)
VARIABLES elements
INVARIANT elements  $\subseteq$  ITEM
INITIALISATION elements :=  $\emptyset$ 
OPERATIONS
    input (ii)  $\hat{=}$ 
        PRE ii  $\in$  ITEM
        THEN elements := elements  $\cup$  {ii}
        END ;
....
END
```

On peut utiliser un paramètre ensembliste comme type (*généricité*) et il faudra l'instancier avec un ensemble non-vide.

Le type d'un paramètre est donné dans la clause CONSTRAINTS.

Paramètres, 2

MACHINE Club (*capacity*)

CONSTRAINTS *capacity* $\in \mathbb{N}_1 \wedge capacity \leq 4096$

Dans CONSTRAINTS, les types des paramètres scalaires et autres **contraintes sur les paramètres**. Même relation avec les paramètres que INVARIANT par rapport aux variables de la machine. Mais on ne peut pas contraindre le type des paramètres ensemblistes !!

Erreurs :

MACHINE Store (*ITEM*)

CONSTRAINTS *ITEM* $\subseteq \mathbb{N}$

MACHINE Bijouterie (*PIERRE*, *METAL*)

CONSTRAINTS *PIERRE* $\cap$ *METAL* = $\emptyset$

Ensembles

MACHINE Club (*capacity*)

...

SETS *REPORT* = {*yes*, *no*} ; *NAME*

Un type ensembliste générique peut aussi être introduit dans SETS (en majuscules). Alors :

1. On peut le nommer sans donner d'autre information;
2. On peut aussi l'expliciter comme type *énuméré*, comme *REPORT* dans la machine Club;
3. On peut l'introduire comme abbreviation pour un sous-type (par ex. PAIR par rapport à $\mathbb{N}$).

Constantes, 1

MACHINE Club (*capacity*)

....

CONSTANTS *total*

PROPERTIES

$\text{card}(\text{NAME}) > \text{capacity} \wedge \text{total} \in \mathbb{N}_1 \wedge \text{total} > 4096$

- ▶ Analogie : constantes globales dans un programme.
- ▶ Type : tout type connu par la machine, et on l'indique dans la clause PROPERTIES.
- ▶ Dans PROPERTIES on peut aussi indiquer des relations entre des SETS et des paramètres (par ex. entre *NAME* et *capacity*).

Constantes, 2

Différence entre le paramètre *capacity* et la constante *total* ?

On doit pouvoir instancier la machine à *n'importe lequel* nombre naturel valeur de *capacity* (généricité) et *au moins une* valeur appropriée de *total*.

Pour valider les PROPERTIES il faut prouver une formule de la forme :

$\forall capacity \exists total \dots$

Voir après les obligations de preuve associées à la clause PROPERTIES.

Opérations de Requête

MACHINE Club (*capacity*)

...

OPERATIONS

...

$ans \leftarrow query_membership(nn) \hat{=}$

PRE $nn \in NAME$

THEN

IF $nn \in member$

THEN $ans := yes$

ELSE $ans := no$

END

END

....

query_membership : opération de requête : produit une information sur l'état de la machine mais ne change pas l'état de la machine.

Nouvelles Obligations de Preuve, 1

Pour une clause CONSTRAINTS $C(p)$ avec paramètres p , on doit démontrer : $\exists p \ C(p)$

Pour Club :

$\exists capacity (capacity \in \mathbb{N}_1 \wedge capacity \leq 4096)$

Puis : toujours instancier un paramètre ensembliste à un ensemble $\neq \emptyset$!

Nouvelles Obligations de Preuve, 2

Pour une clause PROPERTIES $Prop(p, s, c)$ avec paramètres p , constantes c et ensembles (sets) s on doit démontrer que : $\forall$ valeur de p qui satisfait les contraintes $C(p)$, il existe des valeurs pour s et c tels que $Prop(p, s, c)$ vaut vrai

Obligation : prouver le théorème : $C(p) \Rightarrow (\exists c \exists s Prop(p, s, c))$

N.B. Point de vue logique : paramètre p comme une variable libre x , et prouver $\forall x(A(x) \Rightarrow \exists y(B(x, y)))$ revient à prouver $A(x) \Rightarrow \exists y(B(x, y))$.

Pour Club, il faut prouver :

$$(capacity \in \mathbb{N}_1 \wedge capacity \leq 4096) \Rightarrow \exists NAME \exists total$$

$$(card(NAME) > capacity \wedge total \in \mathbb{N}_1 \wedge total > 4096)$$

Nouvelles Obligations de Preuve, 3

Il faut prouver :

l'INVARIANT $I(v)$ contenant les variables v est satisfiable par des valeurs de v , et cela pour toutes les valeurs des paramètres p , des constantes c et des ensembles (sets) s telles que les contraintes $C(p)$ et les PROPERTIES $Prop(p, s, c)$ sont vraies.

Obligation de preuve de :

$$(Prop(p, s, c) \wedge C(p)) \Rightarrow \exists v I$$

Pour Club, prouver :

$(card(NAME) > capacity \wedge total \in \mathbb{N}_1 \wedge total > 4096 \wedge capacity \in \mathbb{N}_1 \wedge capacity \leq 4096) \Rightarrow$

$$\exists member \exists waiting$$

$$(member \subseteq NAME \wedge waiting \subseteq NAME \wedge$$

$$member \cap waiting = \emptyset$$

$$\wedge card(member) \leq 4096 \wedge card(waiting) \leq total)$$

Nouvelles Obligations de Preuve, 4

Il faut prouver : l'invariant $I(v)$ contenant les variables v est préservé par l'exécution T de l'INITIALISATION, et cela pour toutes les valeurs des paramètres p , des constantes c , des ensembles (sets) s et des variables v) telles que les contraintes $C(p)$ et les *properties* $Prop(p, s, c)$ sont vraies.

Obligation de preuve de :

$$(Prop(p, s, c) \wedge C(p)) \Rightarrow [T]I$$

Pour Club ceci signifie prouver (par R1) :

$$(card(NAME) > capacity \wedge total \in \mathbb{N}_1 \wedge total > 4096 \wedge capacity \in \mathbb{N}_1 \wedge capacity \leq 4096) \Rightarrow$$

$$(\emptyset \subseteq NAME \wedge \emptyset \subseteq NAME \wedge \emptyset \cap \emptyset = \emptyset \wedge card(\emptyset) \leq 4096 \wedge card(\emptyset) \leq total)$$

Nouvelles Obligations de Preuve, 5

Pour toute opération $op \hat{=} \text{PRE } prec \text{ THEN } S \text{ END}$ il faut prouver :

l'invariant $I(v)$ est préservé par l'exécution de S , et cela pour toutes les valeurs des paramètres, des constantes, des ensembles (sets) et des variables telles que les contraintes $C(p)$, les *properties* $Prop(p, s, c)$ et la *prec* de op sont satisfaites.

Pour toute opération $op \hat{=} \text{PRE } prec \text{ THEN } S \text{ END}$ on a l'**obligation de preuve** de :

$$(Prop(p, s, c) \wedge C(p) \wedge I \wedge prec) \Rightarrow [S]I$$

Obligations de preuve pour les opérations de Club : au tableau.

Nouvelles Obligations de Preuve, 6

Cas particulier des opérations de requête : elles ne modifient pas les variables de VARIABLES –les seules variables de l'invariant –.

Soit op une opération de requête qui exécute S . Puisque $[S]I = I$, cohérence banale avec l'invariant I :

$(Prop(p, s, c) \wedge C(p) \wedge I \wedge prec) \Rightarrow [S]I$ est une tautologie.

Pas d'obligations de preuve !

Visibilité entre Clauses d'une AM

- ▶ CONSTRAINTS voit les paramètres;
- ▶ PROPERTIES voit les paramètres, les ensembles (sets) et les constantes;
- ▶ INVARIANT voit les paramètres, les ensembles (sets), les constantes et les variables de VARIABLES;
- ▶ OPERATIONS voit les paramètres, les ensembles (sets), les constantes et les variables de VARIABLES;

Relations

Une AM peut utiliser des variables (déclarées dans VARIABLES) et/ou des constantes (déclarées dans CONSTANTS) pour des **relations**.

Une relation (binaire) r entre l'ensemble S et l'ensemble T est un sous-ensemble de $S \times T$, donc un ensemble de couples.

Le type d'une variable x pour une relation est donné dans l'INVARIANT, et celui d'une constante c dans PROPERTIES.

Notation : $x \in S \leftrightarrow T$, $c \in S \leftrightarrow T$.

N.B. : le type de x et c est un ensemble (de couples). La méthode B sait raisonner en logique et en théorie des ensembles !

Si r est une relation $\in S \leftrightarrow T$ et $U \subseteq S$, la notation $r[U]$ indique ces éléments de T qui sont reliés à des éléments de U (*image relationnelle* de U par r).

Ex: si r est $>$ sur $\mathbb{N}$, alors $r[\{4\}] = \{0, 1, 2, 3\}$.

Fonctions, 1

Une **fonction partielle** f de S à T est une relation $\subseteq S \times T$ telle que $\forall s \in S \exists$ au maximum un $t \in T$ tel que $(s, t) \in f$.

Domaine de f = le plus grand sous-ensemble S' de S tel que, $\forall s \in S' \exists ! t \in T$ tel que $(s, t) \in f$. Notation : $dom(f)$

Ensemble d'arrivé de f = le plus grand sous-ensemble de T contenant des éléments t tels que $(s, t) \in f$ (pour quelques $s \in dom(f)$). Notation : $ran(f)$ (*range* de f).

Une **fonction totale** f de S à T est une fonction partielle de S à T telle que le domaine de f est S , c'est-à-dire que $\forall s \in S \exists ! t \in T$ tel que $(s, t) \in f$.

Fonctions, 2

Notation	Signification
$f \in S \rightharpoonup T$	f est une fonction (partielle) de S à T
$f \in S \rightarrow T$	f est une fonction <u>totale</u> de S à T
$f \in S \not\rightarrow T$	f est une fonction (partielle) et <u>injective</u> de S à T
$f \in S \rightarrowtail T$	f est une fonction totale et <u>injective</u> de S à T
$f \in S \Leftrightarrow T$	f est une fonction totale et <u>bijjective</u> de S à T
$f; f'$	la fonction composée de f par f' ($f' \circ f$)
$\text{dom}(f)$	le domaine de la fonction f
$\text{ran}(f)$	l'ensemble d'arrivé de la fonction f
$f[U]$	l'image de $U \subseteq S$ par f .

Exemple

Machine utilisant déclarations de relations et fonctions : [Reading](#).

Partie Statique

MACHINE Reading

SETS *READER* ; *BOOK* ; *COPY* ; *RESPONSE* = {yes, no}

CONSTANTS *copyof*

PROPERTIES $\text{copyof} \in \text{COPY} \rightarrow \text{BOOK}$

VARIABLES *hasread*, *reading*

INVARIANT

$\text{hasread} \in \text{READER} \leftrightarrow \text{BOOK} \quad \% \text{ hasread relation}$

$\wedge \text{reading} \in \text{READER} \not\rightarrow \text{COPY}$

$\wedge (\text{reading}; \text{copyof}) \cap \text{hasread} = \emptyset$

INITIALISATION $\text{hasread}, \text{reading} := \emptyset, \emptyset$

Exemple, suite

Partie Dynamique de l'AM Reading : les Opérations, 1

L'opération *start* ajoute un nouveau couple (*lecteur*, *copie*) à la fonction *reading*, qui dit qui est en train de lire quoi.

$start(rr, cc) \hat{=}$

PRE

$rr \in \text{READER} \wedge cc \in \text{COPY} \wedge \text{copyof}(cc) \notin \text{hasread}[\{rr\}] \wedge$
 $rr \notin \text{dom}(\text{reading}) \wedge cc \notin \text{ran}(\text{reading})$

THEN $\text{reading} := \text{reading} \cup \{rr \mapsto cc\}$

où :

- $a \mapsto b$ est une façon de noter le couple $(a, f(a))$ d'une relation (fonctions incluses)
- Rappel : $r[U]$ est l'image relationnelle de U , c'est-à-dire $\{t \mid t \in T \text{ et } \exists u \in S (s, t) \in r\}$.

Exemple, suite

Partie Dynamique de l'AM Reading : les Opérations, suite

L'opération *finished* fait si qu'un lecteur *rr* soit considéré comme ayant terminé de lire une copie *cc* d'un livre donné.

$finished(rr, cc) \hat{=}$

PRE

$rr \in \text{READER} \wedge cc \in \text{COPY} \wedge cc = \text{reading}(rr)$

THEN $hasread := hasread \cup \{rr \mapsto copyof(cc)\} \parallel$

$reading := \{rr\} \text{ remove_de } reading$

où $\parallel$ est une façon de noter deux actions faites en parallèle et, si E est un sous-ensemble du domaine d'une fonction f , alors $E \text{ remove_de } f$ note la fonction f privée des couples $(x, f(x))$ où $x \in E$.

Exemple, suite

Partie Dynamique de l'AM Reading : les Opérations, suite

L'opération de requête *precurrentquery*(*rr*) teste si *rr* est en train de lire un livre ou pas;

L'opération de requête *currentquery*(*bb*) produit *bb*, le livre que *rr* est en train de lire;

L'opération de requête *hasreadquery*(*rr*, *bb*) teste si *bb* est un livre déjà lu par *rr*.

```
resp  $\leftarrow$  precurrentquery(rr)  $\hat{=}$   
  PRE rr  $\in$  READER  
  THEN  
    IF rr  $\in$  dom(reading) THEN resp := yes ELSE resp := no  
  END ;  
bb  $\leftarrow$  currentquery(rr)  $\hat{=}$   
  PRE rr  $\in$  READER  $\wedge$  rr  $\in$  dom(reading)  
  THEN bb := copyof(reading(rr))  
  END ;  
resp  $\leftarrow$  hasreadquery(rr, bb)  $\hat{=}$   
  PRE rr  $\in$  READER  $\wedge$  bb  $\in$  BOOK  
  THEN  
    IF bb  $\in$  hasread[{rr}] THEN resp := yes ELSE resp := no  
  END ;
```

Tableaux

Dans la méthode B : un tableau avec indices dans I et valeurs dans V est vu comme une fonction partielle de I à V . **N.B** : la méthode B ne sait raisonner logiquement que sur les ensembles !

Exemples

Le tableau de taille 4 :

3	7	5	0
---	---	---	---

peut tre vu comme une fonction partielle de $\mathbb{N}$ à $\mathbb{N}$, ou comme une fonction totale de $1..4$ à $\mathbb{N}$ ou comme une fonction totale de $1..4$ à $\{0, 1, \dots, 17\}$ etc.

Le tableau vide existe, et c'est la fonction vide (pas de couples).

↪ si on met à jour la valeur de la case i du tableau t , on écrit $t(i) := \text{nouvelle_valeur}$ mais **ce que l'on modifie c'est la fonction** t , qui change de valeur pour l'argument i et reste identique pour tout autre argument.

Les affectations multiples dans le même tableau ($t(i), t(j) := 3, 4$) sont **interdites**, car problème si $i = j$!

Tableaux : Weakest Precondition

Soit t un tableau, i un index, P une postcondition souhaitée après la mise à jour de $t(i)$. On obtient, comme cas particulier de la weakest précondition pour les affectations :

$$[t(i) := E]P = P[(t <+ \{i \mapsto E\}) / t]$$

où la fonction $t <+ \{i \mapsto E\}$ qui remplace t est définie par :
 $(t <+ \{i \mapsto E\})(j)$ est égal à E si $i = j$ et à $t(j)$ sinon.

Exemples

$[t(3) := 6](t(3) = 6) \rightsquigarrow_{R1} (t <+ \{3 \mapsto 6\})(3) = 6 \rightsquigarrow 6 = 6$ par déf. de $<+$.

$[t(4) := 6](t(3) = 6) \rightsquigarrow_{R1} (t <+ \{4 \mapsto 6\})(3) = 6 \rightsquigarrow t(3) = 6$ par déf. de $<+$. Cet énoncé sera vrai si $t(3) = 6$ était déjà vrai avant l'affectation.

Ici, noté par $\rightsquigarrow$ la réécriture due à $R1$ ou aux définitions.

Swap et Σ

Les affectations multiples modifiant $t(i)$ et $t(j)$ sont interdites, en général, mais le swap, qui échange les valeurs de $t(i)$ et $t(j)$, est OK.

swap : $t := t < + \{i \mapsto a(j), j \mapsto a(i)\}$

où : $(t < + \{i \mapsto t(j), j \mapsto a(i)\})(n)$ est $t(j)$ si $n = i$, est $t(i)$ si $n = j$ et c'est $t(n)$ sinon.

Pourquoi le cas $i = j$ ne pose pas de problèmes, ici ?

Notation $\Sigma x.(P(x) \mid E(x) =)$: somme de toutes les valeurs de $E(x)$ pour lesquelles $P(x)$ est vraie.

Si t est un tableau ayant les indices dans $1..\mathbb{N}$, la notation

$$\Sigma i.(i \in 1..\mathbb{N} \mid t(i))$$

signifie la somme de tous les valeurs du tableau t .

Machine Hotelguests

```
MACHINE Hotelguests (sze)
CONSTRAINTS  $sze \in \mathbb{N}_1$ 
SETS ROOM ; NAME ; REPORT = {present, nopresent}
CONSTANTS empty % nom d'un client inexistant
PROPERTIES  $card(ROOM) = sze \wedge empty \in NAME$ 
VARIABLES guests
INVARIANT  $guests \in ROOM \rightarrow NAME$  % guests est un tableau
INITIALISATION  $guests := ROOM \times \{EMPTY\}$ 
OPERATIONS
  guestcheckin( $rr, nn$ )  $\hat{=}$ 
    PRE  $rr \in ROOM \wedge nn \in NAME \wedge nn \neq empty$ 
    THEN  $guest(rr) = nn$ 
    END ;
  guestcheckout( $rr$ )  $\hat{=}$ 
    PRE  $rr \in ROOM$ 
    THEN  $guests(rr) = empty$ 
    END ;
   $nn \leftarrow$  guestquery( $rr$ )  $\hat{=}$ 
    PRE  $rr \in ROOM$ 
    THEN  $nn = guests(rr)$ 
    END
   $rr \leftarrow$  presentquery( $nn$ )  $\hat{=}$ 
    PRE  $nn \in NAME \wedge nn \neq empty$ 
    THEN
      IF  $nn \in ran(guests)$ 
      THEN  $rr := present$ 
      ELSE  $rr := nopresent$ 
      END
    END
  guestswap( $rr, ss$ )  $\hat{=}$ 
    PRE  $rr \in ROOM \wedge ss \in ROOM$ 
    THEN  $guests := guests <+ \{rr \mapsto guests(ss), ss \mapsto guests(rr)\}$ 
    END
END
```

Indéterminisme

Toutes les constructions AMN vues jusqu'à ici étaient **déterministes** : une seule valeur des sorties possible pour une entrée donnée, un seul état final pour un état initial donné.

Mais utiliser de l'**indéterminisme** dans les spécifications est **utile** : liberté pour le programmeur, possibilité de retarder certain choix.

Indéterminisme = sous-spécification

La syntaxe AMN offre plusieurs opérateurs non-déterministes : **ANY, CHOICE, SELECT, PRE** (on verra après en quel sense **PRE** est indéterministe).

ANY 1

ANY x WHERE Q THEN T END

- ▶ x est une variable nouvelle et locale à l'énoncé ANY;
- ▶ Q est une formule qui donne le type de x et d'autres infos, et peut faire référence à d'autres variables;
- ▶ T le *body* de l'énoncé, est une n'importe quelle instruction AMN dont le résultat dépend de la valeur de x .

Sémantique : Une x arbitraire pour laquelle P vaut vrai est choisie, et T est exécutée pour cette x .

Il faudra s'assurer que au moins une telle x existe.

Exemple 1 : "Énoncé de diminution"

**ANY t WHERE $t \in \mathbb{N} \wedge t \leq total \wedge 2 \times t \geq total$ THEN
 $total := t$ END**

Que se passe-t-il pour des états initiaux où $total$ est un nombre dans 1...6 ? Au tableau.

ANY 2

Exemple 2 : “Achats raisonnables”

ANY a **WHERE** $a \subseteq \text{articles_choisis} \wedge \text{prix}(a) \leq \text{limite}$ **THEN**
 $\text{achats} := a$ **END**

On choisit un ensemble d'articles tels que le prix globale est au dessous d'une certaine limite.

Exemple 3 : Comment écrire une opération qui, étant donné un $x \in \mathbb{N}$, affecte la variable div à n'importe quel diviseur d de x ?

ANY 3

Weakest Precondition pour **ANY**

Soit la weakest precondition :

$$WP = [\mathbf{ANY} \ x \ \mathbf{WHERE} \ Q \ \mathbf{THEN} \ T \ \mathbf{END}] \ P$$

Cette precondition doit dire que peu importe la valeur de x telle que Q , l'exécution de T assure P .

Calcul de WP :

$$[\mathbf{R5}] : WP = \forall x (Q \Rightarrow [T]P)$$

Exercice au tableau : calculer la weakest precondition pour l'exemple 1 (énoncé de diminution) quand P est $total > 1$:

[ANY t
WHERE $t \in \mathbb{N} \wedge t \leq total \wedge 2 \times t \geq total$
THEN $total := t$ **END]**
 $(total > 1)$

ANY 4

La syntaxe AMN permet des énoncés de la forme :

LET x **BE** $x = E$ **IN** S **END**

disant que les occurrences de x dans S doivent être évaluées par E .

Il s'agit d'une macro pour un cas particulier d'énoncé **ANY** :
lequel ?

ANY 5

En général, un énoncé **ANY** peut utiliser une liste de variables :

ANY $x_1, \dots, x_n$ **WHERE** Q **THEN** T **END**

et alors Q est une formule qui donne les types de toutes les x_i et d'autres infos, peut faire référence à d'autres variables et peut aussi spécifier des contraintes sur les combinaisons de valeurs permises (voir après l'exemple loto, version 2).

Donc la forme générale de la weakest precondition pour **ANY** est :

[R5G] :

[ANY $x_1, \dots, x_n$ **WHERE** Q **THEN** T **END]** $P =$

$\forall x_1 \dots \forall x_n (Q \Rightarrow [T]P)$

ANY 6

Exemple : opération loto, version 1, avec 1 variable, TT , de type ensembliste

$SS \leftarrow loto \hat{=}$

PRE $True$

THEN ANY TT

WHERE $TT \subseteq 1..49 \wedge card(TT) = 6$

THEN $SS := TT$

END

END

ANY 7

Exemple : opération loto, version 2, avec 6 variables, TT , de type $\mathbb{N}$

$a, b, c, d, e, f \leftarrow loto \hat{=}$

PRE $True$

THEN ANY $a0, b0, c0, d0, e0, f0$

WHERE $a0 \in 1..49 \wedge b0 \in 1..49 \wedge c0 \in 1..49 \wedge$
 $d0 \in 1..49 \wedge e0 \in 1..49 \wedge f0 \in 1..49 \wedge$
 $card(\{a0, b0, c0, d0, e0, f0\}) = 6$

THEN $a, b, c, d, e, f := a0, b0, c0, d0, e0, f0$

END

END

Les 6 variables du **ANY** sont contraintes à prendre des valeurs différentes entr elles.

CHOICE 1

On peut faire un choix parmi un nombre fixé d'alternatives :

CHOICE S **OR** T ... **OR** U **END**

Exemple 1 : test pour le permis de conduire

CHOICE $resultat := success$ || $permis_donnes :=$
 $permis_donnes \cup \{candidat\}$
OR $resultat := echec$

Exemple 2 : variation de l'exemple "achats raisonnables"

CHOICE
ANY a **WHERE** $a \subseteq articles_choisis \wedge prix(a) \leq limite$ **THEN**
 $achats := a$ **END**
OR $limite := prix(articles_choisis) + 100$ || $achats :=$
 $articles_choisis$

NB : indéterminisme dans les alternatives, aussi.

CHOICE 2

Weakest Precondition pour CHOICE

[R6] :

[CHOICE S_1 OR... OR S_n] P = $[S_1] P \wedge \dots \wedge [S_n] P$

NB : c'est bien un AND !

SELECT 1

Chaque alternative est contrôlée par une condition permettant de la déclencher.

```
SELECT  $Q_1$  THEN  $T_1$   
WHEN  $Q_2$  THEN  $T_2$   
:  
WHEN  $Q_n$  THEN  $T_n$   
ELSE  $V$   
END
```

- ▶ Plusieurs Q_i peuvent être vraies, à un état. Alors une S_i quelconque parmi celles pouvant tre déclanchées, est executée (indéterminisme !)
- ▶ Si aucune des S_i est vraie, c'est V qui est executée.
- ▶ Le **ELSE** est optionnel mais, si absent, alors les $Q_1, \dots Q_n$ doivent couvrir tous les cas possibles !

SELECT 2

Exemple 1 : choix d'un assistant

$a \leftarrow \text{assistant} \hat{=}$

PRE *True*

THEN

SELECT *anne* $\in$ *presents*

THEN $a := \text{anne}$

WHEN *bernard* $\in$ *presents*

THEN $a := \text{bernard}$

WHEN *tarek* $\in$ *presents*

THEN $a := \text{tarek}$

ELSE $a := \text{damien}$

END

END

Si plusieurs personnes présentes, choix indéterministe de l'assistant.

SELECT 3

Exemple 2 : la position d'une pièce sur un échiquier 4×4 est représentée par ses coordonnées (x, y) où $1 \leq x \leq 4$ et $1 \leq y \leq 4$. On bouge la pièce d'un carré à la fois, de façon horizontale ou verticale, mais on ne peut pas la sortir de l'échiquier.

```
SELECT  $x > 1$  THEN  $x := x - 1$   
WHEN  $x < 4$  THEN  $x := x + 1$   
WHEN  $y > 1$  THEN  $y := y - 1$   
WHEN  $y < 4$  THEN  $y := y + 1$   
END
```

Que se passe-t-il si $x = 2$ et $y = 4$?

CHOICE 4

Weakest Precondition pour SELECT

Le cas où **ELSE** est présent se réduit au cas où c'est absent, en ajoutant une autre clause de la forme **WHEN** $\neg Q_1 \wedge \dots \neg Q_n$ **THEN** V .

Donc la formulation suivante est générale :

[R7] :

$$[\text{SELECT } Q_1 \text{ THEN } T_1 \dots \text{WHEN } Q_m \text{ THEN } T_m \text{ END}] P = Q_1 \Rightarrow [T_1]P \wedge \dots \wedge Q_m \Rightarrow [T_m]P$$

Exercice au tableau :

Calculer la weakest precondition pour l'exemple de l'échiquier et $P : y > 2$.

Exercice au tableau :

L'expression **IF** E **THEN** T est équivalente à un énoncé **SELECT**.
Lequel ?

PRE

L'opérateur **PRE**, déjà vu pour indiquer la precondition d'une opération est indéterministe au sens que

PRE Q THEN S END

est tel que si Q est fausse alors toute exécution est permise, mme une qui ne termine pas ! Si pas de terminaison, aucune postcondition est assurée, même pas *True* !

Donc :

R8 : $[\text{PRE } Q \text{ THEN } S \text{ END}]P = Q \wedge [S]P$

(une autre règle de calcul de la weakest precondition)

Structuration des machines et modularité

- ▶ Structurer une “grosse” machine en modules est un principe de base du GL;
- ▶ Reutiliser une machine dont la cohérence a été déjà prouvée comme composante d’une nouvelle machine est utile;
- ▶ Des mécanismes de structuration par rapport aux machines composantes de la “grande” machine permettent une forme d’abstraction dite “*semi-hiding*”;
- ▶ $\leadsto$ Etude de quelques opérateurs de structuration.

INCLUDES, 1

On peut réutiliser des machines. Une machine M1 peut être incluse dans M2 avec la clause de M2 : **INCLUDES** M1

- 1 L'état de M1 devient une partie de l'état de M2, mais c'est juste M1 que peut affecter ces variables, car la cohérence de M1 (déjà prouvée) doit être préservée.
- 2 Toutefois, l'INVARIANT de M2 peut contraindre les variables d'état de M1, et leur relation avec les variables d'état natives de M2.
- 3 Tout aspect de M1 est visible par M2, mais M1 est une machine **indépendante** : pas de référence à M2, dans M1;
- 4 Les SETS et les CONSTANTS de M1 sont visibles par M2, exactement comme les SETS et CONSTANTS "natifs" de M2. Les PROPERTIES de M2 peuvent les utiliser.
- 5 Les opérations de M2 ont un accès de lecture à l'état de M1, dans leur PRE et dans leur *body*.

INCLUDES, 2

- 6 L'INVARIANT de M2 presuppose l'INVARIANT de M1;
- 7 L'INITIALISATION de M2 commence par initialiser M1
- 8 Une opération *op* de M1 peut tre rendue disponible à M2 avec la clause de M2 : **PROMOTES** *op*. Une *op* promue **doit présever à priori l'INVARIANT de M2**.
On peut aussi juste appeler une opération de M1, dans une opération de M2, sans la promouvoir (voir la suite pour la $\neq$).
- 9 Cas particulier : M2 **PROMOTES** toute opération de M1. On peut déclarer, dans M2 : **EXTENDS** M1

Inclusion Multiple

Une machine M peut inclure, au même niveau, plusieurs machines $M'_1, \dots, M'_n$. Chaque M'_i peut inclure elle-même des machines, etc.

Quand 2 machines M'_1 et M'_2 sont incluses dans M au même niveau, on peut appeler une opération $op1$ de M'_1 et une opération $op2$ de M'_2 de façon parallèle, avec l'opérateur $op1 \parallel op2$.

PRE P_1 THEN S_1 END $\parallel$ PRE P_2 THEN S_2 END

se réécrit en :

PRE $P_1 \wedge P_2$ THEN $S_1 \parallel S_2$.

Ensuite, on peut encore réduire $S_1 \parallel S_2$, selon plusieurs cas de figure, jusqu'à faire disparaître $\parallel$. Cette simplification est nécessaire pour calculer les obligations de preuve générées par $S_1 \parallel S_2$ car il n'y a pas de règle permettant de calculer $[S_1 \parallel S_2]P$, directement à partir de $[S_1]P$ et $[S_2]P$.

Réductions d'instructions ||, 1

- ▶ $S \parallel \text{skip} = S$
- ▶ $S \parallel T = T \parallel S$
- ▶ $(x_1, \dots, x_n := E_1, \dots, E_n) \parallel (y_1, \dots, y_m := F_1, \dots, F_m) = x_1, \dots, x_n, y_1, \dots, y_m := E_1, \dots, E_n, F_1, \dots, F_m$
- ▶ $(\text{IF } E \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}) \parallel T = \text{IF } E \text{ THEN } S_1 \parallel T \text{ ELSE } S_2 \parallel T$
- ▶ $(\text{CHOICE } S_1 \text{ OR } S_2 \text{ END}) \parallel T = \text{CHOICE } S_1 \parallel T \text{ OR } S_2 \parallel T \text{ END}$
- ▶ $(\text{ANY } x \text{ WHERE } E \text{ THEN } S \text{ END}) \parallel T = \text{ANY } x \text{ WHERE } E \text{ THEN } S \parallel T \text{ END}$
- ▶ $(\text{PRE } P \text{ THEN } S \text{ END}) \parallel T = \text{PRE } P \text{ THEN } S \parallel T \text{ END}$

Pourquoi, ici, la precondition P de S devient une precondition de $S \parallel T$? Car la sémantique de $\parallel$ est telle que $S \parallel T$ termine ssi les 2, S et T , terminent. Or, si P est fausse, et on exécute quand mme $S \parallel T$, S pourrait ne pas terminer....

Exemples de réduction (exercice)

Exercices

Réduire :

1. **ANY** x **WHERE** $x \in \mathbb{N}$ **THEN** $x := x^2$ **END** ||
 $y := y - 3$
2. **IF** $x = 1$ **THEN** $y := y + 1$ **ELSE** $y := y - 1$ **END** || $z := z + 3$
3. **IF** $x = 1$ **THEN** $y := y + 1$ **ELSE** $y := y - 1$ **END** ||
ANY w **WHERE** $w \in 1..10$ **THEN** $z := z + w$ **END**
4. **IF** $x > y$ **THEN** $y, z := y + z, 0$ **END** ||
CHOICE $x := x + y$ **OR** $x := x + z$ **END**
5. **ANY** x **WHERE** $x \in 4..y$ **THEN** $z := x^2$ **END** ||
PRE $y > 3$ **THEN** $y := y - 3$ **END**

INCLUDES à travers d'un exemple

On va étudier un exemple de machine SAFE qui INCLUDES directement une machine Keys et une machine Locks, laquelle INCLUDES Doors.

On va commencer par le machines de plus bas niveau.

Exemple INCLUDES 1, Machine atomique Keys

MACHINE Keys

SETS *KEY*

VARIABLES *keys*

INVARIANT $keys \subseteq KEY$

INITIALISATION $keys := \emptyset$

OPERATIONS

$insertkey(k) \hat{=}$ % une var input, pas de var output

PRE $k \in KEY$

THEN $keys := keys \cup \{k\}$

END ;

$removekey(k) \hat{=}$ % une var input, pas de var output

PRE $k \in KEY$

THEN $keys := keys \setminus \{k\}$

END

END

Exemple INCLUDES 1, Machine atomique Keys, legenda

Legenda.

Afin que tout le reste ait un sense, il faut voir *Keys* comme les clés qui sont de facto insérées dans une quelque porte, et c'est tout.

Donc : *keys* (variable d'état de *Keys*)= ensemble des clés INSEREES.

Cette machine prend sons sense juste dans le contexte de la machine de top niveau Safes.

INCLUDES 2, Machine atomique Doors

MACHINE Doors

SETS *DOOR* ; *POSITION* = {*open*, *closed*}

VARIABLES *position* % var fonctionnelle

INVARIANT $position \in DOOR \rightarrow POSITION$ % fonc totale

INITIALISATION $position := DOORS \times \{closed\}$

OPERATIONS

$opening(d) \hat{=}$

PRE $d \in DOOR$

THEN $position(d) := open$

END ;

$closedoor(d) \hat{=}$

PRE $d \in DOOR$

THEN $position(d) := closed$

END

END

INCLUDES 3, Machine atomique Doors, Legenda

La machine Doors ne s'occupe pas des clés.

Elle se limite à déclarer, via **sa fonction d'état position**, ayant les portes comme arguments, si une porte a la position *open* ou bien *closed*.

INITIALISATION : Au départ, toutes les portes sont *closed*.

Elles changent de *position* via *opening(d)* ou *closedoor(d)* qui presupposent juste que *d* soit une porte.

L'INVARIANT dit juste que toute porte est ou bien ouverte ou bien fermée.

INCLUDES 4, Machine Locks, including Doors

La machine Locks s'occupe des portes (des coffreforts d'une banque), et de leur verrous, sans s'occuper des clés. Mais elle includes Doors.

Partie Statique

MACHINE Locks

INCLUDES Doors

PROMOTES closedoor % promotion

SETS $STATUS = \{locked, unlocked\}$

VARIABLES *status*

INVARIANT

$status \in DOOR \rightarrow STATUS$ % fonc totale

$\wedge position^{-1}[\{open\}] \subseteq status^{-1}[\{unlocked\}]$

INITIALISATION $status := DOORS \times \{locked\}$

INCLUDES 4, Machine Locks, including Doors. Legenda

Legenda de la partie statique de Locks

PROMOTES closedoor dit que closedoor est comme une opération native de Locks.

La variable d'état *status* associe à chaque porte soit la propriété *locked* soit la propriété *unlocked*.

Dans l'INVARIANT de Locks on dit que :

- ▶ Chaque porte est soit *locked* soit *unlocked*;
- ▶ les portes *open* (= ouvertes, delon Doors) ne sont pas *locked* (verrouillées). A nouveau; les clés ne jouent pas encore de rôle.

On établit donc une relation entre Locks et Doors.

INCLUDES 4, Machine Locks, including Doors. Partie dynamique

Machine M2 : la machine Locks, Partie dynamique

OPERATIONS

$\text{opendoor}(d) \hat{=} \quad \% \neq \text{opening}, \text{ op de Doors}$

PRE $d \in \text{DOOR} \wedge \text{status}(d) = \text{unlocked}$

THEN $\text{opening}(d) \quad \% \text{ op de Doors utilisée, mais pas promue}$

END ;

$\text{unlockdoor}(d) \hat{=}$

PRE $d \in \text{DOOR}$

THEN $\text{status}(d) := \text{unlocked}$

END

$\text{lockdoor}(d) \hat{=}$

PRE $d \in \text{DOOR} \wedge$

$\text{position}(d) = \text{closed} \quad \% \text{ NB : } \text{position} = \text{var d'état de Doors}$

THEN $\text{status}(d) := \text{locked}$

END

INCLUDES 4, Machine Locks, including Doors, partie dynamique. Legenda

- ▶ L'opération *opendoor* de M2 est native, mais appelle *opening*, opération de M1.
- ▶ L'opération *closedoor* est une opération de M1 promue.
Différence entre promotion et appel ?
- ▶ *closedoor* n'a pas besoin de preconditions autres que celles déjà déclarées dans Locks. Son utilisation ne peut pas violer l'INVARIANT de Locks, qui dit juste que les portes *open* ne sont pas *locked*
- ▶ *opening*, en revanche (opération de DOORS qui déclare une porte *open*) peut être utilisée seulement si le status de la porte est "unlocked" (voir la PRE de *opendoor*), sinon l'INVARIANT de Locks, (qui dit que si *open* alors *unlocked*) serait violée. Donc il faut que *Locks* contrôle *opening* avec ses propres preconditions !
- ▶ Pourquoi si on déclarait *Locks EXTENDS Doors* on aurait une machine incohérente ?

INCLUDES 5, Machine Safe, la machine de top niveau; son rôle ?

Jusqu'à ici, avec Locks on a mis en relation les verrous avec la condition ouverte/fermée des portes (DOORS), mais les clés n'ont pas joué de rôle.

On assemble le tout avec la machine Safes (coffres), qui includes au premier niveau Keys et Locks (la quelle include Doors) .

INCLUDES 6, Machine Safe, la machine de top niveau

Machine Safes (coffreforts), partie statique

MACHINE Safes

INCLUDES Locks, Keys % au même niveau

PROMOTES *opendoor, closedoor, lockdoor*

CONSTANTS *unlocks* % association bijective des clés à leur portes

PROPERTIES *unlocks* $\in KEY \Leftrightarrow DOOR$ % bijection

INVARIANT $status^{-1}[\{unlocked\}] \subseteq unlocks[keys]$

INCLUDES 7, Machine Safe, Partie Statique, Legenda

Legenda

SAFE n'a pas de variable d'état propre, mais a le rôle d'assurer la cohérence du tout.

N.B : l'opération *closedoor* est native de Doors (donc 2 niveaux + bas) mais elle a été déjà importée dans Locks, donc c'est son opération aussi. *opendoor* et *lookdoor* sont natives de Locks. Les trois opérations sont promues dans Safes.

La constante fonctionnelle *unlocks* associe de façon **bijective** des clés aux portes qu'elles peuvent déverrouiller. **Donc une clé peut déverrouiller une et une seule porte, et chaque porte a sa propre clé.** On a autant de portes que de clés.

L'invariant dit : si une porte *d* est *unlocked* (*status* est une variable fonctionnelle de Locks) alors *d* est une porte donc la clé associée (selon la bijection *unlocks*) est une clé INSEREE (penser à *Keys* de KEYS!).

INCLUDES 8, Machine Safe, partie dynamique

OPERATIONS

$insert(k, d) \hat{=}$

PRE $k \in KEY \wedge d \in DOOR \wedge unlocks(k) = d$ % k est la clé de d

THEN $insertkey(k)$ % opération de Keys

END ;

$extract(k, d) \hat{=}$

PRE $k \in KEY \wedge d \in DOOR \wedge$

$unlocks(k) = d \wedge status(d) = locked$

THEN $removekey(k)$ % opération de Keys

END ;

$unlock(d) \hat{=}$

PRE $d \in DOOR \wedge unlocks^{-1}(d) \in keys$

THEN $unlockdoor(d)$ % opération de Locks

END ;

$quicklock(d) \hat{=}$

PRE $d \in DOOR \wedge position(d) = closed$

THEN $lockdoor(d) \parallel removekey(unlocks^{-1}(d))$

END ;

INCLUDES 8, Machine Safe; que fait elle donc ?

- ▶ Avec $insert(k, d)$, si *unlocks* a déclaré que d est LA porte correspondant à k (selon la bijection *unlocks* établie par safe), alors on **ajoute k aux clés insérées**;
- ▶ Avec $extract(k, d)$, si *unlocks* a déclaré que d est LA porte correspondant à k (selon la bijection *unlocks* établie par safe) mais d est déjà verrouillée, alors on **supprime k de l'ensemble des clés insérées** (donc : on enlève la clé k de la porte d !);
- ▶ Avec $unlock(d)$, si la clé associée à d (selon la bijection *unlocks* établie par Safe) est une clé insérée, alors on deverrouille d ;
- ▶ L'opération **quicklock** met à jour, au même temps, les deux machines (directement) incluses : si une porte est fermée, alors on déclare son status *locked* (par le biais d'une operation de Locks) et on déclare la clé correspondante comme non-insérée (opération *removekeys* de Keys) : on l'enlève de sa porte;

Obligations de Preuve spécifiques à l'inclusion

Soit $M1$ incluse dans $M2$ et soient : C_1 les CONSTRAINTS de $M1$, C_2 celles de $M2$, ST_1 les SETS de $M1$, ST_2 ceux de $M2$, k_1 les CONSTANTS de $M1$, k_2 celles de $M2$, B_1 les PROPERTIES de $M1$, B_2 celles de $M2$, v_1 les VARIABLES de $M1$, v_2 celles de $M2$, I_1 l'INVARIANT de $M1$, I_2 celui de $M2$, T_1 l'instruction de l'INITIALISATION de $M1$, T_2 celle de $M2$.

Il faut prouver :

1. Si p_2 sont les paramètres de $M2$: $\exists p_2(C_2)$
2. $(C_1 \wedge C_2) \rightarrow \exists ST_1, ST_2 k_1 k_2 (B_1 \wedge B_2)$
3. $(C_1 \wedge C_2 \wedge B_1 \wedge B_2) \rightarrow \exists v_1 v_2(I_1 \wedge I_2)$
4. $(C_1 \wedge C_2 \wedge B_1 \wedge B_2) \rightarrow [T_1; T_2]I_2$
où “;” est la composition séquentielle.
5. $(C_1 \wedge C_2 \wedge B_1 \wedge B_2 \wedge I_1 \wedge I_2 \wedge P) \rightarrow [S]I_2$
pour toute opération de $M2$ (possibly PROMOTED) ayant la forme **PRE P THEN S END**.

Raffinement

L'atelier B permet de **RAFFINER** pas par pas une spécification, **jusqu'au code** (C ou Ada).

Niveau spécification pure : le **QUOI**. Raffinement : le **COMMENT**, de plus en plus.

Plusieurs opérateurs utilisables dans les étapes de raffinement.

Composition Séquentielle 1

$S ; T$: exécuter S , **puis** T .

Si S ne termine pas, $S ; T$ non plus !

Exemples

$x := x + 2 ; x := x + 4$ est équivalent à $x := x + 6$

On peut composer plusieurs opérations, par ex. $t := x ; x := y ; y := t$
(swap, $t = \text{var temporaire}$)

Règle pour calculer la Weakest Précondition :

[R8] :

$$[S; T]P = [S]([T]P)$$

S doit assurer d'**arriver** à un état tel que $[T]P$ est vraie.

Composition séquentielle 2

$$[x := x + 2; x := x + 4](x > 9) =$$

$$[x := x + 2]([x := x + 4](x > 9)) = \textcolor{red}{R1}$$

$$[x := x + 2](x + 4 > 9) = \textcolor{red}{R1}$$

$$x + 2 + 4 > 9 = x + 6 > 9 = x > 3$$

Exercice : Calculer $[x := y; y := x^2](y > x)$.

Variables Locales

Dans le contexte de ; c'est parfois utile de déclarer des variables **locales** à une instruction (comme t pour le swap entre x et y).

Instruction générale pour le `faInRe` :

VAR $x_1 \cdots x_n$ **IN** S

La **Weakest Precondition** pour la déclaration de variables locales est semblable à celle du **ANY** :

$$[R9] : [\mathbf{VAR} \ x_1 \cdots x_n \ \mathbf{IN} \ S]P = \forall x_1 \cdots \forall x_n ([S]P)$$

Exemple (swap) :

$$[\mathbf{VAR} \ t \ \mathbf{IN} \ t := x ; x := y ; y := t \ \mathbf{END}](x = A \wedge y = B) \\ = (R9, \text{weakest precondition pour les variables locales})$$

$$\forall t ([t := x ; x := y ; y := t] (x = A \wedge y = B)) \\ = (R8, \text{weakest Precondition pour ;})$$

$$\forall t ([t := x]([x := y](y := t) (x = A \wedge y = B))) = (R1)$$

$$\forall t ([t := x] ([x := y] (x = A \wedge t = B)) = (R1)$$

$$\forall t ([t := x] ((y = A \wedge t = B)) = (R1)$$

$$\forall t ((y = A \wedge x = B) = (\text{logique})$$

$$y = A \wedge x = B$$

Machines de Raffinement

Une machine MR qui raffine une machine M est déclarée par :

REFINEMENT MR

REFINES M

Il faut établir un **INVARIANT de liaison** qui connecte les états de MR aux états (abstraits) de M.

Il faut aussi décrire ce que deviennent l'initialisation et les opérations de M.

Illustration par l'exemple du raffinement d'une machine abstraite
Equipe qui stocke dans une variable *équipe* l'ensemble des joueurs
sur le terrain (11) d'une équipe de football de 22 joueurs qui
comptent aussi les joueurs assis : on peut remplacer un joueur qui
est sur le terrain par un autre qui est assis.

Machine ABSTRAITE Equipe

```
MACHINE Equipe
SETS REPONSES = {in, out}
VARIABLES equipe
INVARIANT equipe  $\subseteq$  1..22  $\wedge$  card(equipe) = 11
INITIALISATION equipe = 1..11 % un ensemble abstrait, les joueurs sur le terrain
OPERATIONS
  remplacer(old, new)  $\hat{=}$  % on remplace un joueur
    PRE old  $\in$  equipe  $\wedge$  new  $\in$  1..22  $\wedge$  new  $\notin$  equipe
    THEN equipe := (equipe  $\cup$  {new})  $\setminus$  {old}
    END ;
  rep  $\leftarrow$  query(j)  $\hat{=}$  % le joueur j est sur le terrain ?
    PRE j  $\in$  1..22
    THEN
      IF j  $\in$  equipe
      THEN rep := in
      ELSE rep := out
      END
    END
END
```

N.B : Pour exécuter *query*, il faut stocker la valeur de *equipe* avec quelque structure de données. Laquelle ?

On commence par un tableau, indexé avec 1....11, où chaque case contient le numéro identifiant un des 22 joueurs;

$\leadsto$ Machine EquipeR

Raffinement d'Equipe, Version 1

```
REFINEMENT EquipeR
REFINES Equipe
VARIABLES equipeR % nouvelle variable d'état de la machine raffinée
INVARIANT  $equipeR \subseteq 1..11 \mapsto 1..22$  % Fonct. inj. et totale, tableau
           $\wedge \text{ran}(equipeR) = \text{equipe}$  % Invariant de liaison, volnR après
INITIALISATION  $equipeR = \lambda n.(n \in 1..11 \mid n)$ 
OPERATIONS % PRE des opérations implicites, volnR après
  remplacer(old, new) $\hat{=}$ 
     $equipeR(equipeR^{-1}(old)) := new$  % mise à jour du tableau equipeR
  END ;
   $rep \leftarrow query(j) \hat{=}$  % le joueur j est sur le terrain ?
    IF  $j \in \text{ran}(equipeR)$ 
    THEN  $rep := in$ 
    ELSE  $rep := out$ 
    END
  END
END
```

Notation λ pour les fonctions.

La fonction qui fait le carré d'un nombre naturel :

$carre = \lambda x.(x \in \mathbb{N} \mid x^2).$

A l'initialisation, la fonction *equipeR* est l'identité sur $\mathbb{N}$.

Relations entre Equipe et EquipeR

- ▶ L'**invariant de liaison** dit que la valeur d'*equipe*, variable d'EQUIPE, est l'ensemble d'arrivée du tableau *equipeR*.
- ▶ On suppose que les **PRE** des opérations de EQUIPE (**abstraite**) sont valables :
 - *remplacer* déclenchée seulement si *equipe* contient *old* mais ne contient pas *new* et $new \in 1..22$;
 - *query* déclenchée seulement si $j \in 1..22$.Ceci assure, par ex, que *old* soit dans $ran(equipeR^{-1})$ (étant donné l'invariant de liaison), donc $equipeR^{-1}(old)$ est bien définie (*equipeR* étant injective) $\leadsto$
 $equipeR(equipeR^{-1}(old)) := new$ est bien définie.
- ▶ Plusieurs états d'EquipeR (= plusieurs tableaux) représentent le même état d'Equipe (ensemble abstrait).
Par ex., $[3 \mid 5 \mid 11 \mid 1 \mid 2 \mid 9 \mid 4 \mid 10 \mid 7 \mid 8 \mid 6]$ et $[1 \mid 2 \mid 11 \mid 5 \mid 3 \mid 9 \mid 4 \mid 10 \mid 7 \mid 8 \mid 6]$ correspondent à $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.

Raffinement d'Equipe, Version 2

On peut choisir de représenter autrement la variable d'état d'Equipe, par exemple par un tableau indexé par les joueurs, disant si un joueur donné est sur le terrain ou pas par le biais des valeurs *in* et *out*.

Rappel : Equipe déclare : **SETS** *REPONSES* = {*in*, *out*}



Raffinement d'Equipe, Version 2

REFINEMENT EquipeRbis

REFINES Equipe

VARIABLES *equipeRep* % nouvelle variable d'état de EquipeRbis

INVARIANT $equipeRep \subseteq (1..22 \rightarrow REPONSES)$

$\wedge equipeRep^{-1}[\{in\}] = equipe$ % **nouvel Invariant de liaison**

INITIALISATION $equipeRep = (1..11) \times \{in\} \cup (12..22) \times \{out\}$

OPERATIONS % PRE des opérations implicites

$remplacer(old, new) \hat{=}$

BEGIN $equipeRep(old) := out$; $equipeRep(new) := in$ **END**

$rep \leftarrow query(j) \hat{=} rep := equipeRep(j)$

END

$equipeRep \subseteq (1..22 \rightarrow REPONSES)$ est une fonction totale. C'est un tableaux indexé par les joueurs, disant si un joueur donné est sur le terrain ou pas. Le **nouveau invariant de liaison** dit que l'ensemble abstrait *equipe*, variable d'état de la machine abstraite Equipe, correspond aux joueurs j t.q. $equipeRep(j) = in$.

Du côté, $query(j)$, qui veut savoir si le joueur j est sur le terrain, teste la valeur (*in* ou *out*) de ce nouveau tableau pour le joueur j .

Raffiner en éliminants des informations inutiles

Machine Abstraite NotesExam

```
MACHINE NotesExam
SETS ETUDIANTS
VARIABLES notes
INVARIANT  $notes \in ETUDIANTS \rightarrow 0..20$     % fonct. partielle
INITIALISATION  $notes = \emptyset$ 
OPERATIONS
  inserer( $e, n$ )  $\hat{=}$ 
    PRE  $e \in ETUDIANTS \wedge e \notin dom(notes) \wedge n \in 0..20$ 
    THEN  $notes(e) := n$ 
    END ;
  rep  $\leftarrow$  moyenne  $\hat{=}$ 
    PRE  $notes \neq \emptyset$ 
    THEN  $\sum z.(z \in dom(notes) \mid notes(z)) / card(dom(notes))$ 
    END ;
  n  $\leftarrow$  nombre  $\hat{=}$   $n := card(dom(notes))$ 
END
```

N.B. Cette machine veut juste faInRe des statistiques.

Etant donné les opérations indiquées, l'identité de l'étudiant est InRrelevante. Ce qui compte, ce sont le nombre n de notes données, et leur moyenne (stockée dans rep).

Donc, quand on raffine, on peut éliminer les ETUDIANTS : volnR la machine qui raffine dans ce sense, NotesExamR.



Machine NotesExamR

```
REFINEMENT NotesExamR
REFINES NotesExam
VARIABLES total, num    % les variables d'états de la nouvelle machine
INVARIANT num = card(dom(notes))    % Partie de l'invariant de liaison
         $\wedge$  total =  $\sum z. (z \in \text{dom}(\text{notes}) \mid \text{notes}(z))$     % Partie de l'invariant de liaison
INITIALISATION total := 0, num = 0
OPERATIONS
  insérer(e, n)  $\hat{=}$ 
    BEGIN total := total + n || num := num + 1 END
  rep  $\leftarrow$  moyenne  $\hat{=}$  total / num ;
  n  $\leftarrow$  nombre  $\hat{=}$  n := num
END
```

Un état donné de NotesExamR peut correspondre à plusieurs états de la variable de la machine abstraite NotesExam. Par ex.
 $total = 25, num = 2$ à $notes = \{\langle Marc, 7 \rangle, \langle Julie, 18 \rangle\}$ mais aussi à $notes = \{\langle Marie, 14 \rangle, \langle Isabelle, 11 \rangle\}$.

Les opérations peuvent être déclenchées seulement si les **PRE** de la machine abstraite sont trues. En particulier, *insérer*(*e*, *n*) est déclenchée seulement si $e \notin \text{dom}(note)$, ce qui revient à supposer qu'on ne réécrit pas la note d'un étudiant, donc que que *total* et *num* se comportent bien.

Machine NotesExamR

L'invariant de liaison de NotesExamR dit que *num* correspond à la cardinalité de l'ensemble d'arrivée de la fonction *notes* de la machine abstraite NotesExam et *total* à leur somme.

Donc ici on explicite juste la relation entre *total* et *num* (les variables d'état de NotesExamR) et la variable d'état abstraite *notes* de NotesExam (qui était un tableau).

Stocker un reseau de routes

Avec l'exemple suivant, on étudie le raffinement d'une machine abstraite Ville par plusieurs étapes de raffinement.

La machine de départ Villes stocke les *routes* connectant des villes; les routes sont bidInRectionnelles, mais on peut les stocker comme uni-dInRectionnelles.

Machine Abstraite Villes

```
MACHINE Villes (VILLE)    % VILLE = paramètre
SETS REPONSES = {oui, non}
VARIABLES routes
INVARIANT routes ∈ VILLE ↔ VILLE    % relation binaInRe
INITIALISATION routes = ∅
OPERATIONS
  lier(v1, v2) ≡
    PRE v1 ∈ VILLE ∧ v2 ∈ VILLE
    THEN routes := routes ∪ {v1 ↦ v2}    % on ajoute le couple orienté ⟨v1, v2⟩
    END ;
  rep ← query_connection(v1, v2) ≡
    PRE v1 ∈ VILLE ∧ v2 ∈ VILLE
    THEN
      IF (v1 ↦ v2 ∈ (routes ∪ routes-1)* ) ∨ v1 = v2    % VolnR la note (1)
      THEN verdict := oui
      ELSE verdict := non
      END
END
END
```

Note(1) : Si R est une relation binaInRe, R^n est R composée n fois, et R^* est la **cloture transitive** de R , c'est-à-dInRe $R^* = \bigcup_{n \geq 0} R^n$.

Donc le **IF** de l'opération *query_connection* teste si deux villes $v1, v2$ sont connectées en testant si ou bien il y a un chemin qui va de $v1$ à $v2$ ou de $v2$ à $v1$, ou bien ou bien $v1$ et $v2$ sont la même ville. Dit autrement, on teste si $\langle v1, v2 \rangle \in$ **la cloture symétrique, transitive et réflexive** de *routes*.

Comment peut on raffiner (implémenter ?) ?

Machine Abstraite Villes; Raffiner

Premier pas de raffinement.

De facto, Villes maintient une relation binaire d'équivalence entre les villes : la clôture symétrique, transitive et réflexive de *routes*, est réflexive, symétrique et transitive.

Alors, on peut implémenter en stockant une partition des villes en classes d'équivalence de villes reliées (composantes connexes maximales du réseau).



Raffinement : VillesR, 1

Partie Statique

REFINEMENT VillesR % héritage du paramètres VILLE de Villes

REFINES Villes

VARIABLES *partition, classe*

INVARIANT % contient l'invariant de liaison

$$\begin{aligned} & partition \in \mathbb{P}(VILLE) \wedge \\ & \bigcup cl (cl \in partition \mid cl\} = VILLE \wedge \\ & \forall cl1 \forall cl2 ((cl1 \in partition \wedge cl2 \in partition) \Rightarrow \\ & \quad (cl1 = cl2 \vee cl1 \cap cl2 = \emptyset)) \\ & \wedge \forall v (v \in VILLE \Rightarrow \\ & \quad (routes \cup routes^{-1} \cup id(VILLE))^*[\{v\}] \in partition) \\ & \wedge classe \in VILLE \rightarrow \mathbb{P}(VILLE)) \\ & \wedge \forall v (v \in VILLE \Rightarrow v \in classe(v)) \\ & ran(classe) = partition \\ & \forall cl (cl \in partition \Rightarrow classe^{-1}[\{cl\}] = cl \end{aligned}$$

En général, la machine raffinée hérite toute la partie statique (paramètres, sets et constantes) de la machine abstraite.

Raffinement : VillesR, 2

Que dit l'INVARIANT de VillesR ?

1 $partition \in \mathbb{P}(VILLE) \wedge \bigcup cl (cl \in partition \mid cl) = VILLE$:
partition est un ensemble de sous-ensembles cl_i de *VILLE* tel que l'union des cl_i est *VILLE*;

2 $\forall cl1 \forall cl2 ((cl1 \in partition \wedge cl2 \in partition) \Rightarrow (cl1 = cl2 \vee cl1 \cap cl2 = \emptyset))$:

Les éléments de *partition* sont disjoints.

↪ la variable d'état *partition* stocke une véritable partition de l'ensemble *VILLE*.

3 $\forall v (v \in VILLE \Rightarrow$

$(routes \cup routes^{-1} \cup id(VILLE))^*[\{v\}] \in partition)$:

pour chaque ville v , l'ensemble des villes v' tels que $\langle v, v' \rangle$ est dans la clôture symétrique, transitive et réflexive de *routes* est un élément de *partition*. Donc : *partition* est l'ensemble des classes d'équivalence de *VILLE* selon cette relation (c.à.d. des composantes maximales du réseau). C'est l'invariant de *liaison*: connexion de *partition* à *routes*.

Raffinement : VillesR, 2

Que dit l'INVARIANT de VillesR, SUITE ?

$$4 \text{ classe} \in VILLE \rightarrow \mathbb{P}(VILLE) \wedge$$

$$\forall v (v \in VILLE \Rightarrow v \in \text{classe}(v)) \wedge$$

$$\text{ran}(\text{classe}) = \text{partition} \wedge$$

$$\forall cl (cl \in \text{partition} \Rightarrow \text{classe}^{-1}[\{cl\}] = cl :$$

classe est une fonction totale qui associe à chaque $v \in VILLE$ sa classe d'équivalence et *partition* est l'ensemble des classes d'équivalence.

Raffinement : VillesR, 3

Partie Dynamique

INITIALISATION

```
partition := {cl | cl ∈ (P)(VILLE) ∧ card(cl) = 1} % villes isolées  
|| classe := {(v, c) | (v, c) ∈ VILLE × (P)(VILLE) ∧ c = {v}}
```

OPERATIONS

```
lier(v1, v2) ≐
```

```
IF classe(v1) ≠ classe(v2)
```

```
THEN partition := (partition \ {classe(v1), classe(v2)}) ∪ {classe(v1) ∪ classe(v2)} {;} % fusion  
    classe <+ (classe(v1) ∪ classe(v2)) × {classe(v1) ∪ classe(v2)} % m.à.j. analogue de classe
```

```
END ;
```

```
verdict ← query_connection(v1, v2) ≐
```

```
IF classe(v1) = classe(v2)
```

```
THEN verdict := oui
```

```
ELSE verdict := non
```

```
END
```

```
END
```

La fonction *classe* associe à chaque ville sa classe d'équivalence.

Par ex., à partir de l'initialisation, et en prenant $VILLE = \mathbb{N}$, par des appels de *lier*, on peut créer les classes successives :

1. {1, 2}, {3, 4}, {5, 6}
2. {1, 2}, {3, 4, 5, 6} et puis encore l'unique classe
3. {1, 2, 3, 4, 5, 6}

Encore un raffinement

Une classe d'équivalence peut être identifiée par un de ses éléments, qu'on voit comme **l'élément représentatif** de sa classe?

On peut alors implémenter VillesR avec un ultérieur pas de raffinement, où on en mémorise juste l'association de chaque ville à la ville représentative de sa classe.

Ceci conduit à un pas ultérieur de raffinement, $VillesRR \rightsquigarrow VillesRR$

VillesRR

```
REFINEMENT VillesRR
REFINES VillesR
VARIABLES rep  % une seule var stocke le réseau
INVARIANT  % contient l'invariant de liaison avec VillesR
    rep ∈ VILLE → VILLE ∧  % fonct totale, tableau
    ∀v1 ∀v2 (v1 ∈ VILLE ∧ v2 ∈ VILLE) ⇒
        classe(v1) = classe(v2) ⇔ (rep(v1) = rep(v2)))
INITIALISATION rep := id(VILLE)  % villes isolées
OPERATIONS
    lier(v1, v2) ≐
        IF rep(v1) ≠ rep(v2)
        THEN rep := rep <+ ((rep-1[{rep(v1)}]) × {rep(v2)})
        END ;
    verdict ← query_connection(v1, v2) ≐
        IF rep(v1) = rep(v2)
        THEN verdict := oui
        ELSE verdict := non
        END
END
```

Ici on lie $v1$ à $v2$ en déclarant que la ville représentative de (la classe de) $v1$ est la même ville qui représente (la classe de) $v2$. On le fait par le biais de la fonction total rep qui est une fonction totale de $VILLE$ à $VILLE$, un **tableau indexé par $VILLE$** : toute ville v_i d'une même classe indexe une case du tableau qui contient sa ville représentative $rep(v_i)$.

VillesRR, suite

Suite de l'exemple de création de classes de VillesR, vu du point de VillesRR :

1. $\{1, 2\}, \{3, 4\}, \{5, 6\}$ éléments *rep* correspondants : 2,4,6
2. $\{1, 2\}, \{3, 4, 5, 6\}$ éléments *rep* correspondants : 2, 4
3. $\{1, 2, 3, 4, 5, 6\}$ unique élément *rep* correspondant : 2

Les tableaux *rep* correspondants (en supposant 8 villes) sont :

1. $\{1, 2\}, \{3, 4\}, \{5, 6\}$

2	2	4	4	6	6	7	8
1	2	3	4	5	6	7	8

2. $\{1, 2\}, \{3, 4, 5, 6\}$

2	2	4	4	4	4	7	8
1	2	3	4	5	6	7	8

3. $\{1, 2, 3, 4, 5, 6\}$

2	2	2	2	2	2	7	8
1	2	3	4	5	6	7	8

VillesRR, suite

Selon VILLERR, pour fondre les 2 classes de (2) dans l'unique classe de (3) il faudra lInRe tout le tableau de (2) de façon à falnRe les mises à jour nécessalnRes.

On peut encore raffiner.

Chaque classe peut être vue comme un *arbre* dont le l'élément représentative est la racine et les autres éléments sont des fils ou des descendants.

Le forêt d'arbres correspondant aux classes peut être codée par un tableau *parent*, indexé par VILLE et ayant les éléments de VILLES dans les cases. La case d'index i aura j comme contenu si $j \neq i$ est son parent, tandis que j est une racine si $i = j$. Etant donnée n'importe quelle entrée i dans le tableau, la racine peut être retrouvée en suivant de façon itérative la fonction *parent* jusqu'à remonter à une ville qui est son même parent. La fusion de 2 classes se fera simplement en déclarant la racine d'un des 2 arbres comme étant le parent de la racine de l'autre.

Exemple de ce dernière approche

On reprend l'exemple de création successive de classes et des tableaux *parent* correspondant :

1. $\{1, 2\}, \{3, 4\}, \{5, 6\}$ éléments *rep* correspondants : 2,4,6
2. $\{1, 2\}, \{3, 4, 5, 6\}$ éléments *rep* correspondants : 2, 4
3. $\{1, 2, 3, 4, 5, 6\}$ unique élément *rep* correspondant : 2

1.

2	2	4	4	6	6	7	8
1	2	3	4	5	6	7	8

2.

2	2	4	4	6	4	7	8
1	2	3	4	5	6	7	8

3.

2	2	4	2	6	4	7	8
1	2	3	4	5	6	7	8

VillesRRR

```
REFINEMENT VillesRRR
REFINES VillesRR
VARIABLES parent, n
INVARIANT  % contient l'invariant de liaison avec VillesR, rep étant la var d'état de VillesRR
  parent ∈ VILLE → VILLE    % parent est un tableau
  ∧ n ∈ ℕ    % n est un compteur
  ∧ rep = parentn ∧
  ∀ v ( v ∈ VILLE ⇒ parent(rep(v)) = rep(v))    % racine = représ. de la classe
INITIALISATION parent := id(VILLE) || n = 0    % tout arbre initial a 1 seul noeud
OPERATIONS
  lier(v1, v2) ≡
    VAR rep1, rep2    % var locales
    IN rep1 := parentn(v1) ; rep2 := parentn(v2) ;
      IF rep1 ≠ rep2
      THEN parent(rep1) := rep2 ; n := n + 1    % fusion de 2 arbres, increment de n
      END ;
  rep ← query_connection(v1, v2) ≡
    IF parentn(v1) = parentn(v2)
    THEN rep := oui
    ELSE rep := non
    END
END
```

N.B. $parent^n$ est la composition de la fonction *parent* *n* fois; le compteur *n* sert à itérer la fonction *parent* jusqu'à arriver à la racine de l'arbre en question. Ce *n* doit être au max la hauteur maximale de l'arbre moins 1.

rep sert toujours à donner l'élément représentant une classe d'équivalence.

Raffinement du non-déterminisme

Le raffinement permet de faire des choix laissés ouverts par la sous-spécification induite par les constructions non-déterministes.

```
MACHINE Allouer    % alloue des num de tél aux usagers
SETS ANS = {true, false}
VARIABLES alloues
INVARIANT alloues  $\subseteq \mathbb{N}_1$ 
INITIALISATION alloues :=  $\emptyset$ 
OPERATIONS
  choix(n)  $\hat{=}$ 
    PRE  $n \in \mathbb{N}_1 \wedge n \notin alloues$ 
    THEN  $alloues := (alloues \cup \{n\})$ 
    END ;
  rep  $\leftarrow query(n) \hat{=}$ 
    PRE  $n \in \mathbb{N}_1$ 
    THEN
      IF  $n \in alloues$  THEN  $rep := true$  ELSE  $rep := false$ 
    END
  n  $\leftarrow allouer \hat{=}$     % opération non-déterministe
  ANY m
  WHERE  $m \in \mathbb{N}_1 \setminus alloues$ 
  THEN  $n := m \parallel alloues := alloues \cup \{m\}$ 
  END
END
```

On raffine en donnant un critère pour choisir le nouveau numéro à allouer : on choisit le min des nombres pas encore alloués $\rightsquigarrow$

Machine AllouerR

REFINEMENT AllouerR **REFINES** Allouer

VARIABLES *alloues*

INITIALISATION *alloues* := $\emptyset$

OPERATIONS

choix(*n*) $\hat{=}$ *alloues* := (*alloues* $\cup$ {*n*} ;

rep $\leftarrow$ *query*(*n*) $\hat{=}$

IF *n* $\in$ *alloues* **THEN** *rep* := *true* **ELSE** *rep* := *faux* **END** ;

n $\leftarrow$ *allouer* $\hat{=}$

BEGIN *n* := *min*($\mathbb{N}_1 \setminus \textit{alloues}$) ; *alloues* := *alloues* $\cup$ {*n*} **END**

END

N.B La VAR de Allouer et AllouerR étant la même, *alloues*,
l'invariant de liaison implicite est que les valeurs sont les mêmes.

Obligations de Preuve spécifiques au Raffinement, 1

Illustrées par un raffinement d'une machine abstraite Couleurs

MACHINE Couleurs

SETS *COULEUR* = {rouge, vert, blue}

VARIABLES *cols*

INVARIANT *cols* $\subseteq$ *COULEUR*

INITIALISATION *cols* : $\in \mathbb{P}(\text{COULEUR} \setminus \{\text{blue}\})$ % affectation non-deterministe, plusieurs *cols* ok

OPERATIONS

ajout(*c*) $\hat{=}$ PRE *c* $\in$ *COULEUR* THEN *cols* := *cols* $\cup$ {*c*} END ;

c $\leftarrow$ *query* $\hat{=}$

PRE *cols* $\neq \emptyset$ THEN *c* : $\in$ *cols* % renvoi d'une couleur arbitraire

change $\hat{=}$ *cols* : $\in (\mathbb{P} \setminus \{\text{cols}\})$ % modif arbitrale de *cols*

END

REFINEMENT CouleursR

REFINES Couleurs % set *COULEUR* hérité

VARIABLES *couleur*

INVARIANT *couleur* $\in$ *cols* % invariant de liaison

INITIALISATION *couleur* : $\in (\text{COULEUR} \setminus \{\text{blue}\})$ % indéterminisme nouveau !

OPERATIONS

ajout(*c*) $\hat{=}$ *couleur* : $\in \{\text{couleur}, c\}$ % indéterminisme

c $\leftarrow$ *query* $\hat{=}$ *c* := *couleur* % renvoie la couleur courante, détermination de *query*

change $\hat{=}$ *couleur* : $\in (\text{COULEUR} \setminus \{\text{couleur}\})$ % modif non déterministe de *couleur*

END

Couleurs gere un ensemble de couleurs, CouleursR un seul élément de cet ensemble.

Une expression de la forme $x := S$, où *S* dénote un ensemble, est un raccourcis pour ANY *e* WHERE $e \in S$ THEN $x := e$.

Obligations de Preuve spécifiques au Raffinement, 2

En général, une machine M et son raffinement MR peuvent avoir plusieurs exécutions (indéterminisme).

Il faut assurer que tout état que MR peut atteindre correspond, par l'invariant de liaison J , à quelque état que M peut atteindre.

NB : J est une formule qui porte sur les variables d'états de M et celles de MR .

Intuition : Toute exécution de MR doit correspondre (par J) à quelque exécution de la machine abstraite M , car MR a fait des (pas de) choix d'implémentation d'une spécification abstraite.

Obligations pour l'INITIALISATION, 1

Dans le cas de l'INITIALISATION, il faut établir que, pour n'importe quel état er atteignable par MR à partir de son initialisation InR , il existe au moins un état e atteignable par M à partir de son initialisation In , et tel que J est vraie quand e et er sont les valeurs des variables d'états respectives.

Ceci revient à dire que, quelque soit l'état er de MR atteint par une exécution quelconque de InR , c'est faux que toute exécution de In porte à un état e de M tel que l'invariant J de liaison est faux pour (er, e) .

Il faut donc établir :

$$[InR] \neg [In] \neg J$$

Lire : **Toute** exécution de InR **assure** que c'est faux que toute exécution de In porte à un état où J est fausse, c.à.d. :

toute exécution de InR **assure qu'il existe au moins une exécution** de In permettant la vérité de J . $\Rightarrow$ Exemple sur Couleurs et CouleursR

Obligations pour l'INITIALISATION, Exemple

Figure au tableau

Commentaire à l'Exemple

On peut lier l'état r de CouleursR à un état atteint par In , et de même pour v : donc $\neg[In]\neg J$ est vraie pour r et pour v . Mais $\neg[In]\neg J$ est fausse à l'état b de CouleursR, et il faut donc empêcher à InR de partir de cet état de CouleursR.

C'est bien ce que l'initialisation InR de CouleursR fait.

Obligations pour l'INITIALISATION, 2

Preuve formelle de : $[InR] \neg [In] \neg J$ dans notre cas de Couleurs et CouleursR

- ▶ Lemme : $\neg [In] \neg J = \text{couleur} \neq \text{blue}$

$$\neg [In] \neg J = \neg ([\text{cols} : \in \mathbb{P}(\text{COULEUR} \setminus \{\text{blue}\})] \neg \text{couleur} \in \text{cols})$$
$$=$$

(par la règle de la weakest precondition pour ce cas particulier de ANY)

$$\neg (\forall \text{cols} (\text{cols} \in \mathbb{P}(\text{COULEUR} \setminus \{\text{blue}\}) \Rightarrow \neg \text{couleur} \in \text{cols})$$
$$\equiv \exists \text{cols} (\text{cols} \in \mathbb{P}(\text{COULEUR} \setminus \{\text{blue}\}) \wedge \text{couleur} \in \text{cols})$$
$$\equiv \text{couleur} \neq \text{blue}.$$

- ▶ $[InR] \neg [In] \neg J \equiv$ (par le Lemme)
 $[\text{couleur} : \in (\text{COULEUR} \setminus \{\text{blue}\})] (\text{couleur} \neq \text{blue}) \equiv$
 $\forall \text{couleur} ((\text{couleur} : \in (\text{COULEUR} \setminus \{\text{blue}\}) \Rightarrow \text{couleur} \neq$
 $\text{blue}) \equiv$
 True

Obligations pour les Opérations

Les obligations de preuve pour les opérations suivent la même philosophie :

Il faut s'assurer que toute exécution d'une opération de MR correspond, par l'invariant de liaison J , à quelque exécution de l'opération correspondant de M.

On analyse de façon différente deux types d'opérations :

- ▶ celles sans outputs (comme *ajout*(c) de notre exemple)
- ▶ celles avec outputs

Obligations pour les Opérations : cas sans outputs

En général :

- $op(x)$ de M a la forme PRE P THEN S END
- son raffinement, en MR , a la forme PRE P1 THEN SR END

Mais, en pratique, on ignore souvent P1 car P contient déjà toutes les informations suffisantes.

On veut, essentiellement : la vérité de $[SR] \neg [S] \neg J$, (J étant l'invariant de liaison), mais, cela, *pour tous les états de M et MR atteignables de façon conjointe par MR et M .*

Lire : A partir d'états atteignables de façon conjointe par MR et M , **toute** exécution de SR **assure** que c'est faux que toute exécution de S porte à un état où J est fausse, c.à.d. :

toute exécution de SR **assure qu'il existe au moins une exécution** de S permettant la vérité de J .

Obligations pour les Opérations : cas sans outputs

Par exemple, pour Couleurs, CouleursR et l'opération ajout(*c*) : on veut établir :

$[couleur \in \{couleur, c\}] \neg [cols := cols \cup \{c\}] \neg couleur \in cols.$

Cela a un sens seulement à partir d'états des deux machines qui soient "liés" par *J*.

Par ex : *cols* = {rouge, vert} et *couleur* est *blue* ne sont pas liés par *J*, car *J* = *couleur* ∈ *cols*. A partir de ces valeurs de *cols* et *couleur* :

- ▶ ajout(rouge), pour la machine Couleurs, porte à l'état {rouge, vert}
- ▶ ajout(rouge), pour la machine CouleursR, porte ou bien à l'état *rouge* ou bien à l'état *bleu*, et *bleu* n'est pas lié par *J* à {rouge, vert}. Mais peu importe !

La situation est différente si *cols* = {blue, vert} et *couleur* est *blue*: il y a liaison *J*, et toute exécution de ajout(rouge) de CouleursR DOIT correspondre à quelque exécution de ajout(rouge) de Couleurs.

Obligations pour les Opérations : cas sans outputs

De façon générale, l'obligation de preuve pour $op(x)$ est :

$$I \wedge J \wedge P \Rightarrow [SR] \neg [S] \neg J$$

où I est l'invariant de M , J l'invariant (de liaison) de MR , P est la précondition de $op(x)$ de M , S est son corps, et SR est le corps du raffinement de $op(x)$ dans MR .

Dans notre exemple, pour $op(x) = ajout(c)$, il faut prouver :

$$(cols \subseteq COLOR \wedge couleur \in cols \wedge c \in COLOR) \Rightarrow$$

$$[couleur : \in \{couleur, c\}] \neg [cols := cols \cup \{c\}] \neg (couleur \in cols)$$



Obligations pour les Opérations : exemple de preuve

Supposons :

1. $cols \subseteq COULEUR$ (I)
2. $couleur \in cols$ (J)
3. $c \in COULEUR$ (P)

Alors :

$$[couleur \in \{couleur, c\}] \neg [cols := cols \cup \{c\}] \neg (couleur \in cols) \\ = (R1)$$

$$[couleur \in \{couleur, c\}] \neg (\neg (couleur \in cols \cup \{c\})) \\ = (\text{logique})$$

$$[couleur \in \{couleur, c\}] (couleur \in cols \cup \{c\}) \\ = (\text{cas particulier de la règle de WP pour ANY})$$

$$\forall x (x \in \{couleur, c\} \Rightarrow x \in cols \cup \{c\}) \\ = (\forall \text{ sur deux valeurs de } x)$$

$$(couleur \in cols \cup \{c\}) \wedge (c \in cols \cup \{c\}) \\ = (\text{par 2, et par une banalité ensembliste})$$

true

Suite Exemple

Couleurs et ColuleursR marchent bien ensemble, par rapport à l'opération *ajout(c)* : figure au tableau.

Obligations pour les Opérations : Remarque

Si l'opération raffinée a une precondition explicite $P1$, il faudra aussi prouver :

$$I \wedge J \wedge P \Rightarrow P1$$

Obligations pour les Opérations : cas avec outputs

C'est le cas, par exemple, de l'opération *query* des machines Couleurs et CouleursR.

Ici, il faut aussi assurer que tout output (sortie) possible pour MR est permis par M. Par exemple, si Couleurs et CouleursR sont interrogées par *query* à partir d'états liés par *J*, et *r* (rouge) est un output possible pour CouleursR, il doit être aussi un output possible pour Couleur.

Plus généralement, si *out'* est un output pour une opération de MR (ayant corps SR), ce *out'* doit être égal à quelque valeur *out* qui est la sortie d'une exécution de l'opération correspondante de M (ayant corps S), et doit aussi lui être lié par *J*.

Obligation de preuve :

$$I \wedge J \wedge P \Rightarrow [SR[out'/out]] \neg [S] \neg (J \wedge out' = out)$$

Obligations pour les Opérations : Exemple pour *query*

Supposons :

1. $cols \subseteq COULEUR$ (I)
2. $couleur \in cols$ (J)
3. $c \in COULEUR$ (P)

Alors :

$$\begin{aligned} & [SR[c'/c]] \neg [S] \neg (J \wedge c' = c) = \\ & [c' := couleur] \neg [c \in cols] \neg (J \wedge c' = c) \\ & = (\text{r\`egle pour la WP pour ANY}) \\ & [c' := couleur] \neg (\forall c (c \in cols \rightarrow \neg (J \wedge c' = c))) \\ & = (\text{logique}) \\ & [c' := couleur] \exists c (c \in cols \wedge (J \wedge c' = c)) \\ & = (R1) \\ & \exists c (c \in cols \wedge (J \wedge couleur = c)) \\ & = (\text{logique, car J ne contient pas la variable c}) \\ & J \wedge \exists c (c \in cols \wedge couleur = c) = (\text{logique}) \\ & J \wedge couleur \in cols = true \text{ (par (1) et (2))} \end{aligned}$$