Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Dr. Tahar Moulay de Saida
Faculté de Technologie

Département d'Informatique

Analyse et Conception Formelle

Présenté par :

Dr. Mebarka YAHLALI

Maître de conférences « A » en Informatique

Mai 2023

AVANT -PROPOS

*Ce cours est adressé aux étudiants de la première année Master SIC (**Sécurité Informatique et Cryptographie**) de l'Université de Saida « Dr Tahar Moulay ». L'objectif de l'ACF (**Analyse et Conception Formelle**) est d'étudier les méthodes et les langages formels de spécification et de développement des applications logicielles.*

*Ce polycopié est organisé en quatre chapitres : **Concepts de base**, **Correction d'un programme impératif**, **la méthode B** et **Tests et qualité de logiciels**. Nous commençons le cours par la présentation des principaux concepts de base dans le contexte de la matière. Dans la deuxième partie du cours, nous étudions une méthode classique de correction des programmes : **la logique de Floyd-Hoare**. Cette méthode permet de prouver la correction des programmes impératifs vise à vis une spécification donnée en entrée. Le chapitre trois présentes l'essentiel de la méthode B qui est une méthode de spécification formelle des systèmes et applications logicielles. Dans le dernier chapitre, nous étudions le développement de logiciels piloté par les tests et nous introduisons la Qualité de logiciel. Et enfin la solution de quelques exercices proposés dans ce cours est donnée dans une annexe.*

Table des matières

Avant-propos

Table des matières

Liste des Figures

Introduction Générale **7**

Chapitre I : Concepts de base

I. Processus de programmation **8**

II. Paradigmes de programmation. **8**

II.1.La programmation impérative **9**

II.1.1. La programmation Procédurale **9**

II.1.2. Programmation structurée **10**

II.2. La programmation déclaratif **10**

II.2.1. La programmation descriptive **10**

II.2.2. La programmation fonctionnelle **11**

II.2.2. La programmation logique **11**

II.3. La programmation orientée objet **11**

II.2.1. La programmation descriptive **11**

III. Problèmes rencontrés en algorithmique **11**

III.1. Problème de Terminaison **11**

III.2. Problème de correction **12**

IV. Sémantique des langages de programmation **11**

IV.1. Définition **12**

IV.2. Types de sémantique formelle **13**

Chapitre II: Correction d'un programme impératif

I. Les règles de déduction **16**

II. Les triplets de Hoare..... **16**

III. La plus forte et la plus faible condition **17**

III.1. la plus forte post-condition (SP : Strongest Post-condition) **17**

III.2. la plus faible pré- condition (WP : Weakest Post-condition) **18**

IV. Système de déduction et arbre de preuve **18**

V. Les règles de la logique de Hoare **19**

V.1. Règle 1 : Axiome d'affectation **19**

V.2. Règle 2 : Instruction vide **20**

V.3. Règle 3 : Composition / Séquence **20**

V.4. Règle 4 : Conséquence **20**

V.5. Règle 5 : Conditionnelle **21**

V.6. Règle 6: Boucle **22**

VI .Programme annoté **22**

VII. Exercices **23**

Chapitre III: Méthode B

I. Définitions **26**

I.1. Le cycle de vie des logiciels **26**

I.2. Modèle de cycle de vie **26**

I.3. Spécification, Conception & vérification **26**

I.4. Types de spécification **27**

I.5. Méthodes de spécification **27**

I.6. Méthodes formelles & langage formelle **27**

II. Langage B **28**

II.1. Développement B **28**

II.2. Machine Abstraite **29**

II.2.1. Syntaxe générale **29**

II.2.2.Clauses d'une Machine Abstraite **30**

II.3. Langage pour la partie statique **31**

II.3.1. Typage des données **31**

II.3.2. Définition de base d'ensembles **32**

II.3.3. Les relations **33**

II.3.4. Les fonctions **35**

II.4.Exercices **35**

II.5. Langage des substitutions généralisées **36**

II.5.1.Définitions des opérations..... **36**

II.5.2.Notations **36**

II.5.3 Substitution généralisée **37**

II.5.4. Langage des substitutions généralisées **37**

II.6.Exercices **39**

Chapitre IV: Test et Qualité de logiciels

I. Tests **41**

I.1. Phases du test **41**

I.2. Processus simplifié de test **42**

I.3. Sélection de tests **44**

II. Qualité **45**

II.1. Définition **45**

II.2. Différent types et niveaux de qualité **45**

II.3. Les aspects standards de la qualité **45**

II.4. Modèle de qualité **46**

II.4.1. Le Standard ISO 9162 ..... **47**

Annexe : Solutions des exercices **50**

Bibliographie **60**

Liste des Figures

Fig1 : Programmation impérative 9

Fig2 : Programmation procédurale 10

Fig3 : Format des règles de déduction 16

Fig4 : Arbre de preuve 18

Fig5 : Règle de la logique de Hoare 19

Fig 6: Exemple d'un programme annoté..... 23

Fig7: Développement B 29

Fig8 : Syntaxe d'une Machine Abstraite 30

Fig9 : Exemple d'une Machine Abstraite 31

Fig10 : Phases de test 42

Fig11 : Processus de test 43

Fig12 : Test Fonctionnel 44

Fig13 : Test Structurel 44

Fig14: Différent type de qualité 45

Fig15 : Les aspects standards de la qualité 47

INTRODUTION GENERALE

En Informatique, L'objectif de l'analyse et de la conception est de formaliser les étapes initiales du développement d'une application logicielle afin de répondre aux besoins des clients. Dans le développent il est conseillé de prendre le temps nécessaire durant cette phase afin d'éviter le retour en arrière qui est très couteux.

Généralement les méthodes de spécification sont classées en trois familles : les méthodes informelles, les méthodes semi formelles et les méthodes formelles. Les méthodes informelles sont des méthodes imprécises et ambigües car elles sont basées sur le langage naturel. Les méthodes semi formelles sont basées sur des notations graphiques dotées d'une syntaxe. Les méthodes formelles sont basées généralement sur des modèles mathématiques avec une sémantique et une syntaxe précise. Ces méthodes peuvent remédier aux problèmes d'ambiguïté et conflits posés par le langage naturel et la représentation graphique. Ainsi pour garantir une spécification et une conception de qualité il faux utilisé les meilleures techniques disponibles.

L'objectif de ce cours est d'initier les étudiants à l'analyse et la conception formelle. Nous utilisons la logique de Floyd-Hoare comme une technique de vérification et de correction d'un programme impératif. A chaque instruction algorithmique (affectation , alternative , boucle …) une règle de vérification est attribuée . La méthode B est utilisée comme une méthode de conception formelle. Nous étudions la machine abstraite, le langage utilisé pour la partie statique qui est basé sur la théorie des ensembles et le langage pour la partie dynamique (langage de substitutions généralisées). Le cours contient aussi la partie test des programmes qui permet de minimiser la probabilité d'apparition d'une erreur lors de l'utilisation d'un logiciel.

A l'issue de ce cours, l'étudiant doit être capable de faire une conception formelle à l'aide des méthodes et des langages formels. Il doit également être capable de formaliser les programmes et les invariants ainsi que de vérifier leur terminaison et correction .

Chapitre I : Concepts de base

D'une façon générale l'objectif de la programmation est l'automatisation d'un certain nombre de tâches. Elle représente l'ensemble de moyen et de techniques utilisé pour résoudre des problèmes à l'aide de processus (résultat d'exécution d'un programme) s'exécutant sur un ordinateur [1] [2].

I. Processus de programmation:

Le processus de programmation peut être décomposé en plusieurs phases [3] :

1. **L'analyse et la spécification du problème** : (**Quoi faire?**) permet de préciser les besoins fonctionnelles et les besoins non fonctionnelles (déterminer les données et leurs propriétés, les résultats attendus, et les relations exactes entre les entrées (données) et Sorties (résultats).

2. **La conception ou modélisation** (**Comment Faire ?**) : élaboration des spécifications de l'architecture logiciel et détermination de la méthode de résolution du problème (les algorithmes et les structures de données).

3. **L'implantation (ou codage)** : Traduction de la conception dans un langage de programmation.

4. **Tests et déploiement** : Différents niveaux de tests (test unitaire, test d'intégration, test du système et test d'acception) et le déploiement consiste à mettre le système en fonctionnement opérationnel.

II. Paradigme de programmation :

- L'origine du mot paradigme est le mot grec **paradeigma** qui signifie « modèle » [4] .
- Les langages de programmation sont groupés par familles , selon des principes de fonctionnement différents : ces familles sont appelées « **paradigmes de programmation ».**
- Les paradigmes sont des modèles ou "**patron**" de pensée
- Un paradigme de programmation, exprime une manière de programmation (ensemble de règles et de concepts) :
    - ✓ Paradigme de programmation est un **style de programmation**.
    - ✓ Un bon programme a besoin souvent de plusieurs paradigmes ➔ Chaque paradigme est utile pour certains type de problèmes [5].

II.1. La programmation impérative :

- ✓ Historiquement, c'est le paradigme le plus ancien.
- ✓ Un programme est composé d'un ensemble d'instructions.
- ✓ Nous pouvons traduire le programme par une machine à états : ensemble d'états successifs de la mémoire.
- ✓ Les langages impératifs sont fondés sur une exécution séquentielle
- ✓ L'exécution d'un programme est une séquence d'affectations modifiant son état. (état à l'instant n = les valeurs des différentes variable à cette instant) [6].

Fig1 : Programmation impérative

La majorité des langages impératifs admettent quatre types d'instructions principales

- l'assignation ;
- le branchement conditionnel ;
- le branchement sans condition ;
- le bouclage.

Quelques langages impératifs : Fortran (1954), Algol (1958), COBOL (1960), BASIC (1963), interprété, objectif éducatif Pascal (1970), C (1970), Smalltalk (1972), Ada (1983), C (1985), Perl, Tcl, Python, Php, Java, Javascript (1987-1995),

II.1.1.Programmation procédurale:

La programmation procédurale est vue comme une extension logique de la programmation impérative. Elle consiste à factoriser une séquence d'instructions souvent répétées en procédure (aussi appelée routine, sous-routine ou fonction) que l'on peut appeler autant de fois que l'on veut. Elle permet La création d'un code plus modulaire, structuré et réutilisable [6].

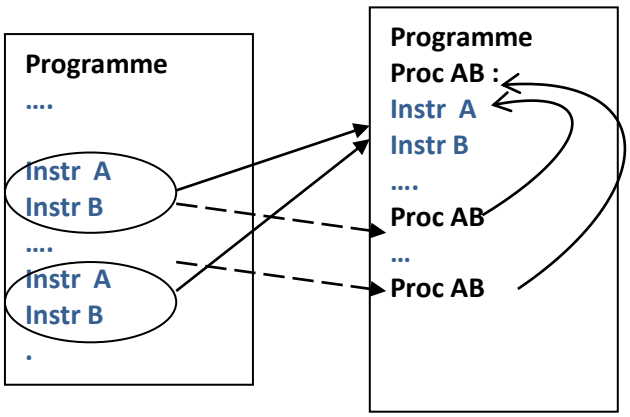


Fig2 : Programmation procédurale

II.1.2.Programmation structurée :

La programmation structurée est proposée en 1968. Elle dérive de travaux :

1. d'Edsger Dijkstra qui dénonce les méfaits de l'instruction goto (problème de complexité).
2. et de Böhm-Jacobi qui montre que toute fonction calculable peut être exprimée en termes de séquences selections (if ... then ... else ...) itérations (while ...) [6]

Exemple :

```
L5:
  if (a >= b)
    goto L4;
  tmp = a;
  a = b;
  b = tmp;
L4:
  a -= b;
L3:
  if (b != 0)
    goto L5;
```

Peut être structuré en

```
while (b != 0) {
  if (a < b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  a -= b;
}
```

II.2. La programmation déclarative:

Ce paradigme exprime la logique du calcul sans décrire son flôt de contrôle. Les langages déclaratifs décrivent ce qui doit être calculé, pas comment. Il existe nombreux types de programmation déclarative :

II.2.1. La programmation descriptive :

Permet de décrire des structures de données : HTML, XML ou LaTeX

II.2.2. La programmation fonctionnelle:

Paradigme déclaratif dans lequel un calcul est vu comme l'évaluation de fonctions. Une application est un ensemble de fonctions (en sens mathématiques) : LISP, Caml, Haskell, Oz [6].

II.2.3. La programmation logique:

Un programme est un ensemble de faits et de règles exprimées dans une logique donnée : Prolog. Le principe de résolution est basé sur des questions (définir un but puis l'évaluer par des axiomes et des règles).

II.3. La programmation orientée objet:

La programmation orientée objet n'est pas un nouveau paradigme mais vient compléter la programmation impérative (SmallTalk, C++, Objective C, Java, Python, ...) et la programmation fonctionnelle (Ocaml, Scala, ...).
L'idée est de regrouper les types de données manipulés par le programme avec les traitements qui peuvent s'y appliquer (La Classe).

III. Problèmes rencontrés en algorithmique :

III.1. Problème de terminaison :

Est-ce que l'algorithme donne un résultat?
Est-ce qu'il ne s'arrête jamais?

Cas :

- ✓ Algorithme itératif contenant une boucle infinie.
- ✓ Algorithme récursif ne contenant pas le point terminal.

Définition : « Un algorithme se termine, si son exécution sur une machine s'arrête toujours quelque soit la nature des données en entrée » [7].

Outil pour la terminaison :

L'algorithme se termine :

- ✓ Lorsque le nombre d'instructions à effectuer est connu à l'avance.
- ✓ **D'une façon générale** : si la condition d'arrêt de la boucle se réalise.

➔ **L'existence d'un convergent pour une boucle garantit que l'algorithme finit par en sortir.**

Définition : le **convergent** est une quantité qui prend ses valeurs dans un ensemble bien conçu et qui diminue strictement à chaque itération de la dans une boucle[7].

III.2. Problème de correction :

Généralement, un programme est dit correct s'il effectue sans erreur les tâches (besoins fonctionnels) demandées dans tous les cas possibles

Est-ce que l'algorithme donne le résultat attendu?
Est-ce qu'il calcule n'importe quoi?

Cas :

✓ Algorithme se termine mais ne donne pas toujours la solution correcte.

Définition

Un algorithme est correct ou valide , s'il se termine et si le résultat qu'il fournit est correct

➔ l'algorithme fait bien ce qu'il est supposé faire dans sa spécification. [7]

Outil pour la Correction :

L'algorithme est correct si :

1. l'initialisation est correcte
2. l'itération est correcte
3. le nombre d'itérations est correct.

➔ La mise d'un invariant de boucle adapté permet de vérifier et prouver la correction d'un algorithme.

Définition : l' **invariant de boucle** est une propriété qui doit vérifier la condition suivante : si elle est vraie à l'état initiale de la boucle (avant l'entrée dans la boucle) , elle restera vraie après chaque itération de la boucle, et donc elle est vraie aussi à la sortie de cette boucle[7].

IV. Sémantique des langages de programmation :

IV.1. Définition :

- « La sémantique étudie le sens ou le potentiel de sens de divers types d'expressions : mots et phrases» [9]
- *« Etude du sens, de la signification dans le langage ».* (**Dictionnaire Robert**)
- « *Etude du sens des unités linguistiques et de leurs combinaisons. Aspect de la logique qui traite de l'interprétation et de la signification des systèmes formels, par opposition à la syntaxe, entendue comme l'étude des relations formelles entre formules de tels systèmes*. » (**Dictionnaire Larousse**)

D'une façon générale La sémantique définit la signification d'un programme (donne un description formelle) : Que signifie ce programme ? Est-ce que ce programme satisfait aux spécifications ? [8]

Objectif : Garantir certaines propriétés vérifiées par les programmes (terminaison , correction) ➔...augmenter la confiance que l'on peut avoir dans les programmes.

IV.2. Types de sémantique formelle :

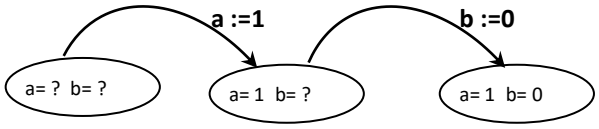
Il existe principalement trois styles de sémantiques formelles :

1. La sémantique opérationnelle :

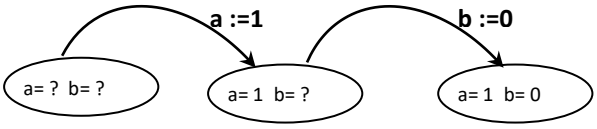
Le programme est vu comme un système de transitions entre états➔ Le sens d'un programme est donné par la séquence d'états successifs d'une machine qui l'exécute (description de l'exécution d'un programme).

Sémantique opérationnelle d'une instruction : C'est l'effet de cette instruction sur un état mémoire.

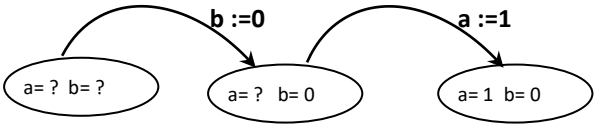
P1:
a:=1;
b:=0;



P2:
a:=1 ;
/* un commentaire */
b:=0;



P3:
b:=0;
a:=1;



Point de vue de la sémantique opérationnelle :

- P1 et P2 sont sémantiquement équivalents.
- P1 n'est pas équivalent à P3.

2. La sémantique dénotationnelle:

Le programme est interprété par une dénotation, c'est-`a-dire une fonction mathématique qui représente l'effet du programme sur la dénotation de son entrée.

Exemple 1 : **expressions arithmétiques**

La dénotation est une fonction :
eval : Exp →N
eval(n)=n
eval(E1 + E2) ::= eval(E1) + eval(E2)
eval(E1 x E2) ::= eval(E1)·eval(E2)
Exemple :
eval((5+1) +(8 + 9))) = eval(5+1) + eval(8+9)
= eval(5)+eval(1) + eval(8+ 9)
= 5+eval(1) + eval(8+ 9)= 5+1+ eval(8+ 9)= 5+1+ eval(8)+ eval(9)
=5+1+8+ eval(9)= 5+1+8+9= 23

Exemple 2 :
P1: a:=1; b:=0; ➔ eval(a)= 1 ; eval(b)=0
P2: b:=0; a:=1; ➔ eval(b)=0 ; eval(a)= 1
P3: b:=1; a:=1; ➔ eval(b)=1 ; eval(a)= 1

P1, P2 et P3 sont dénotationnellement équivalents, mais ils ne sont pas équivalents à P3.

3. La sémantique axiomatique:

Généralement, les propriétés de sémantique axiomatique se précisent, sous la forme d'expressions de la logique de Hoare. Un Programme est définit comme un "transformateur" de propriétés logiques.

Remarque : Ce point est développé dans la chapitre II.

Chapitre II : Correction d'un programme impératif

La logique de Hoare-Floyd a été introduite à la fin des années 60 pour formalisé la relation entre le langage de programmation (impéretatif) et le langage de spécification.

But : formaliser la preuve de la correction des programmes.

Cette logique considère un programme comme un transformateur d'états. Généralement l'état d'un programme représente les valeurs de ses différentes variables. L'exécution avec sucée d'un programme (elle est correct et elle se termine) a pour effet, de transformer l'état initial en un état final.

I. Les règles de déduction :

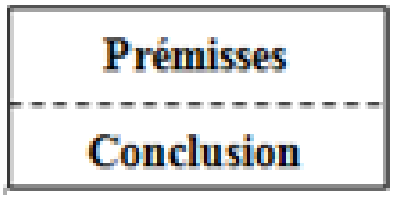


Fig3 : Format des règles de déduction [8]

« Si toutes les prémisses de cette règle sont vraies alors la conclusion est vraie »

- La conclusion et l'ensemble des prémisses sont des formules de la logique de Hoare.
- La conclusion est une assertion.
- Prémisse 1, Prémisse 2,… , Prémisse n : les prémisses (ou hypothèses) sont des assertions.
- En générale n≥0. Si n=0 la règle est un axiome[1].

II. Les triplets de Hoare:

{ P } Prog { Q } :

Prog : ensemble d'instructions
P : formule logique appelée pré-condition
Q : une formule logique appelé post-condition.

La logique de Hoare permet de montrer qu'a partir d'un état initial vérifiant certaines propriétés P , et après l'exécution d'une série d'instruction prog , on obtient un état final vérifiant d'autres propriétés Q [10][13].

[1] Axiome : jugement sans prémisses : ils sont toujours vrais.

Quelques triplets :

{x ≥ 0} x :=3 {x=3} vrais
{x ≥ 0} x :=x + 1 { x > 0 } vrais
{x ≥ 0}x := x + 1 { x < 0} faux

Remarques :

- ✓ Un tel triplet peut être valide ou non
- ✓ La logique de Hoare est correcte vis-à-vis de la sémantique opérationnelle autrement dit pour tout état mémoire remplit P, l'exécution de I produit un état mémoire dans lequel Q est vérifié➔ il faut définir la sémantique opérationnelle d'une instruction.

III. La plus forte post-condition et la plus faible

Notation : A ⇒ B

Signification :

- ✓ la condition A est dite plus forte que B = la condition B est plus faible que A .
- ✓ Pour A il faut B = B est une condition nécessaire pour A.
- ✓ Pour B il suffit A= A est une condition Suffisante pour Q

Objectif :

Prouver des triplets avec pré-condition faibles et post -conditions fortes.

III.1. la plus forte post-condition (SP : Strongest Post-condition)

Définition :

Soit {P} S {Q }
Pour tout Q' tel que {P} S {Q'} / Si Q⇒ Q' alors Q est la plus forte post condition.

Exemples :

- ✓ {x=5} x :=x*2{true}
- ✓ {x=5} x :=x*2{x>10}
- ✓ {x=5} x :=x*2{x=10 || x=5}
- ✓ {x=5} x :=x*2{x=10}

Toutes ces post conditions sont vraies mais on cherche la plus précise ou la plus exacte.

- ✓ {x=10} ⇒{true}
- ✓ {x=10} ⇒{x≥10}
- ✓ {x=10} ⇒{x=10 || x=5}

⇨ {x=10} est la plus forte post-condition

III.2. la plus faible pré- condition (WP : Weakest Post-condition)

Définition :

Soit {P} S {Q }

Pour tout P' tel que {P'} S {Q} / Si P⇒ P' alors P' est la plus faible post condition.

- ✓ La plus faible post condition permet d'invoquer le programme dans la plus part des cas .

Exemples :

- ✓ {x=5&& y=10 } z :=x/y {z<1}
- ✓ {x<y && y>0 } z :=x/y {z<1}
- ✓ {y≠0 &&x/ y<10 } z :=x/y {z<1}

Toutes ces pré- conditions sont vraies mais on cherche la plus faible.

{x=5&& y=10 } ⇒ {y≠0 && x/ y<1 }
{x=5&& y=10 } ⇒ {x<y && y>0 }
{x<y && y>0 }⇒ {y≠0 && x/ y<1 }

⇨ { y≠0 && x/ y<1 } est la plus faible pré-condition

IV. Système de déduction et arbre de preuve

- ✓ Un système de déduction est un ensemble fini de règles.
- ✓ Un arbre de preuve est formé de plusieurs règles, les conclusions des unes formant les prémisses des autres [10] :

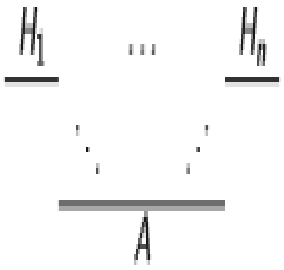


Fig4 : Arbre de preuve

V. Les règles de la logique de Hoare

Une règle est constituée [10] :

- ✓ d'hypothèses ou prémisses H1,...,Hn
- ✓ d'une conclusion
- ✓ Une étiquette : nom de la règle

$$\frac{H_1 \dots H_n}{C}\ R$$

Fig5 : Règle de la logique de Hoare

Remarque : L'ensemble de règle présenté dans ce qui suit est basé sur les références [10-12],

V.1. Règle 1 : Axiome d'affectation

Sémantique : «P dans laquelle x a été remplacé par E»

{P(E) } x :=E {P(x)}

{P(E/x) } x :=E {P}

$$\frac{}{\{P(E/x)\}\ x := E\ \{P\}}\ \text{aff}$$

Si une propriété est vraie pour E alors après l'exécution x :=E (substitution x par E) , la propriété est raie pour x.

Exemples :

1. **{x+2> 2} x := x + 2 {x > 2}**

Raisonnement en arrière :

{ ?? } x := x + 2 {x > 2}

wp(x := x + 2, x > 2)= [x + 2/x](x>2)

=x+2>2

⇨ Le jugement est vrais .

$$\frac{}{\{x+2>2\}\ x := x + 2\ \{x > 2\}}\ \text{aff}$$

2. **{x+1≤ n} y := x + 1 {y≤ n}**

Raisonnement en arrière :

{ ?? } y := x + 1 {y≤ n}
wp(y := x + 1, y≤ n)= [x + 1/y](y≤ n)
= x+1≤ n
⇨ Le jugement est vrais .

$$\frac{}{\{x+1\leq n\}\ y := x + 1\ \{y\leq n\}}\ \textbf{aff}$$

V.2. Règle 2 : Instruction vide

Skip : Commande vide.

$$\frac{}{\{P\}\ skip\ \{P\}}\ \textbf{skip}$$

L'instruction vide ne change pas l'état, Si P est vraie avant, elle est toujours vraie après

V.3. Règle 3 : Composition / Séquence

$$\frac{\{P\}\ prog1\ \{Q'\} \quad \{Q'\}\ prog1\ \{Q\}}{\{P\}\ prog1, prog2\ \{Q\}}\ \textbf{seq}$$

Une règle naturelle pour l'enchaînement séquentiel de deux portions.

Exemple :

{x>2} x :=x+1 ; x :=x+2 {x>5}

Preuve :

{ ?} x :=x+1 ; x :=x+2 {x>5}

wp(**x :=x+1 ; x :=x+2, x>5**) = wp(**x :=x+1 , wp(x :=x+2, x>5)**)
= wp(**x :=x+1 , [x+2**/x](**x>5**))
= wp(**x :=x+1 , x+2>5**)
= wp(**x :=x+1 , x>3**)
= [**x+1/x**](**x>3**)
= x+1>3
= x>2
⇨ Le jugement est vrais .

$$\frac{\{x>2\}\ x := x+1\ \{x>3\} \quad \{x>3\}\ \ x := x+2\ \{x>5\}}{\{x>2\}\ x := x+1\ ;\ x := x+2\ \{x>5\}}\ \text{seq}$$

V.4. Règle 4 : Conséquence

Les propriétés que nous pouvons prouver immédiatement à partir des règles vues précédemment sont d'une forme très contrainte, cependant, il faut pouvoir retenir de ces propriétés celles qui nous intéressent. Ceci se fait par les règles de conséquence qui sont des déductions logique simple .

1. **Conséquence gauche** : Cette règle permet de renforcer les prè-conditions

$$\frac{P \Rightarrow P' \quad \{P'\}\ prog1\ \{Q\}}{\{P\}\ prog1\ \{Q\}}\ \text{consG}$$

2. **Conséquence droite** : Cette règle permet d'affaiblir les post-conditions

$$\frac{Q \Rightarrow Q' \quad \{P\}\ prog1\ \{Q'\}}{\{P\}\ prog1\ \{Q\}}\ \text{consD}$$

V.5. Règle 5 : Conditionnelle

$$\frac{\{P \wedge b\}\ prog1\ \{Q\} \quad \{P \wedge \neg b\}\ prog1\ \{Q\}}{\{P\}\ \text{if b then prog1 else prog2}\ \{Q\}}\ \text{if}$$

Quand le conditionnelle est exécutée, soit prog1soit prog2 est exécuté. Pour établir que Q est une post-condition, il faut donc que ce soit une post-condition des deux branches. De la même façon, P est une pré-condition des deux branches.

- Dans la branche prog1, on sait que la condition **b** est vraie, on peut donc l'ajouter en pré-condition **(P∧b).**
- Similairement, on ajoute ٦**b** comme pré-condition de prog2.

Exemple:

{P} if x>0 then y:=x else y:=-x {y>5}

wp(if x>0 then y:=x else y:=-x , y>5)
⇒ x>0 && wp(x|y,y>5) || x≤0 && wp(-x|y,y>5)
⇒ (x>0 && x>5) || (x≤0 && x<-5)
⇒**P=** x>5 || x<-5

V.6. Règle 6 : Boucle

$$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ \text{tant que b faire S}\ \{I \wedge \neg b\}}\ \textbf{loop}$$

I : invariant de boucle

- ✓ L'invariant doit être vérifié à chaque début d'itération.
- ✓ En sortie de boucle, l'invariant est toujours vrais, mais la condition de boucle est devenue fausse.
- ✓ Avant d'exécuter l'intérieur de la boucle la condition est varie.

Comment trouver un bon invariant de boucle ?

1. Simuler le fonctionnement de la boucle
2. Sélectionner les variables "importantes" de la boucles (l'exécution de la boucle change l'état de ces variables).
3. Etudier l'´évolution des valeurs de variables.
4. Déduire un invariant

VI .Programme annoté

Un programme annoté est un programme dans lequel des assertions sont introduites. On peut insérer des annotations avant et après chaque instruction. On peut même introduire plusieurs annotations entre deux instructions [8].

Exemple:

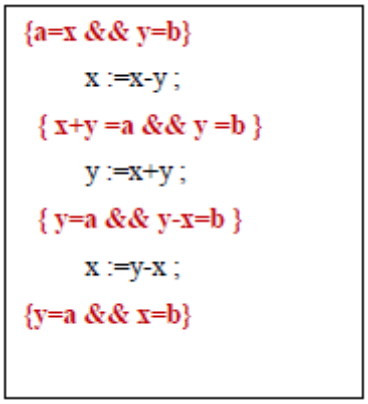


Fig 6: Exemple d'un programme annoté

VII. Exercices

Exercice N°1 :

A. Quelle doit être la plus faible pré- condition pour les triplets suivants :

1. {P} x :=3 {x+y>0}
2. {P} x :=3*y+z {x*y-z>0}
3. {P} x :=x+1 ;y :=x+y {y>5}

B. Les jugements suivants sont-ils corrects ?

1. {x=4} x :=x+1 {x>3}
2. {x>2} x :=x+1 { x>3}

Exercice 2 : Soit le programme Prog suivant :

x :=x-y ;

y :=x+y ;

x :=y-x ;

Montrer que {a=x && y=b} prog {y=a && x=b}

Exercice N°3 :

{x≥-100 && x≤ 100}

If x>0 then x :=x+100 ;

y :=y*2 ;

{y≥0 && x≤ 300}

Exercice N°4 :

Prouver le triplet suivant :

{x≥0 && y≥0 }
q :=0 ;
r :=x ;
while r≥y do
    r :=r-y ;
    q :=q+1 ;
    end ;
{x=q*y+r && 0≤r<y }

Exercice N°5 :

Donner une preuve aux triplets suivants :

1. {n≥1} p :=0 ; i :=1 ; while (i≤n) do p :=p+m ; i :=i+1 ; {p=m*n}
2. {n≥0} f:=1;i:=1; while (i≤n) do f:=f*i; i:= i+1; end; {f:=n!}

Chapitre III :

La Méthode B

Les méthodes de spécification formelle peuvent remédier aux problèmes d'ambiguïté et conflits posés par le langage naturel. Ces méthodes proposent une représentation claire et précise grâce à une notation mathématique et logique .

I. Définitions

I.1.Le cycle de vie des logiciels :

Selon IEEE (1991) : « La période de temps qui débute au moment de la définition et du développement d'un produit logiciel et se termine lorsque le produit logiciel n'est plus disponible pour utilisation. » [14]

I.2. Modèle de cycle de vie :

Représentation idéale et simplifiée des principales activités durant le développement :

- ✓ fournir un vocabulaire de base pour faciliter la communication.
- ✓ rendre explicite (visible) les étapes du développement.
- ✓ permettre la gestion et le contrôle du processus

il existe plusieurs Modèle (Cascade , V , Spiral, Semi-itératif, Itératif ….). Ces Modèles Partagent plusieurs points :

- ✓ analyse et spécification
- ✓ conception générale et détaillée
- ✓ implémentation
- ✓ tests et validation

I.3. Spécification, Conception & vérification :

- ✓ **Spécification :** décrire le système qui satisfera les besoins/exigences et respectera les contraintes définies lors de la phase de l'analyse du problème.
  ➔La spécification : indique ce que le système doit et ne doit pas faire .
  Spécification = QUOI ? que fera le système ?

- ✓ **Conception :** Selon l'IEEE (1991) « Processus qui consiste à définir l'architecture du logiciel, les composants, les modules, les interfaces, les stratégies de tests et les données qui feront en sorte que le système logiciel va satisfaire aux spécifications. » [14]
  ➔Conception =Comment ? comment le système sera-t-il construit ?

- ✓ **Vérification :** assurer qu'un programme obéit à sa spécification.
  - ⇨ **Vérification = Construit-on le produit correctement ?**

I.4. Types de spécification :

- ✓ **Spécification fonctionnelle :** définir les fonctionnalités de l'application (aspects métiers de l'application)

- ✓ **Spécification non fonctionnelle :** Les propriétés non fonctionnelles présentent l'ensemble des propriétés liées à la sécurité, la performance, la portabilité…. (la qualité) [15]

I. 5. Méthodes de spécification : Les méthodes de spécification peuvent être catégorisées selon leur «degré de formalité» :

- ✓ **Méthodes informelles :** sont des notations simples basées sur le langage naturel. C'est une représentation simple est pratique mais elle est imprécise et ambigüe.

- ✓ **Méthodes semi-formelles :** sont des notations graphiques normalisées dotées d'une syntaxe précise (diagrammes complexes). Par exemple UML[2], SADT[3].

- ✓ **Méthodes formelles :** syntaxe est sémantique précises basée sur les mathématiques. Les méthodes formelles sont plus précises et non ambigüe et elles permettent le respect des propriétés grâce à des preuves mathématiques. [15}

I.6. Méthodes formelles & langage formelle :

- ✓ On appelle **langage formelle** de spécification si sa syntaxe et ses notations sont rigoureusement définies par une sémantique bien précise basée sur des modèles mathématiques. [15]

- ✓ On appelle **méthode formelle** de spécification si elle est basée sur un ou plusieurs langages formels. [16]

On peut distinguer deux familles des méthodes spécification :

[2] UML : Unified Modeling Language
[3] *SADT* : Structured Analysis and Design Technics

- ✓ **les approches basées sur les états** : représentent le système à travers deux modèles complémentaires:
  1. **la partie statique** : permet de décrire les entités constituant le système et leurs états,
  2. **la partie dynamique** : modélise les changements d'´etats que le système peut effectuer par l'intermédiaire d'opérations ou d'actions.

  Exemple : méthode Z, méthode B

- ✓ **les approches basées sur les événements** : représentent le système à travers des processus ou des agents, qui sont des entités indépendantes qui communiquent entre elles ou avec l'extérieur du système.

  Exemple : Réseaux de Petri (RDP) [16]

II. Langage B :

La méthode B est une méthode de développement incrémental et modulaire de système logiciel. Elle fournit un langage uniforme nommé AMN : Abstract Machine Notation. Cette notation est utilisée pour **spécifier**, **concevoir** et **vérifier** les programmes (un système est découpé en plusieurs machines abstraites). [17]

II.1. Développement B

- ✓ Analyse système
- ✓ Structuration : Machines abstraites
- ✓ Modélisation données (partie statique)
- ✓ Modélisation opérations (partie dynamique)
- ✓ Raffinements
- ✓ Implantation

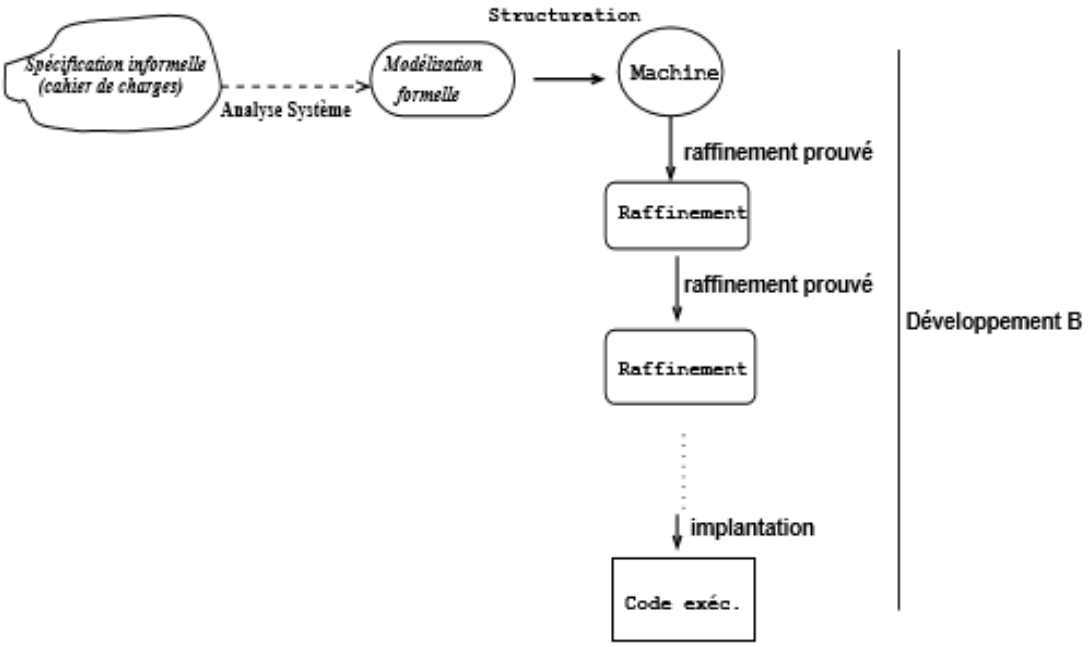


Fig 8: Développement B [17]

II.2. Machine Abstraite

L'AMN (Abstract Machine Notation) est une des notions de base de la méthode B. Elle représente un état spécifié par une partie statique (à l'aide de variables d'état et des propriétés d'invariance) et partie dynamique (à l'aide d'un ensemble d'opérations) [18].

II.2.1. Syntaxe générale :

Une machine abstraite est un sous-système de la spécification du logiciel, caractérisée par :

- ✓ Un en-tête
- ✓ La définition d'un état, désigné par les données et l'invariant que ces données doivent respecter.
- ✓ La description des opérations capables de transformer cet état.

La syntaxe générale d'une AMN est résumée dans la figure suivante :

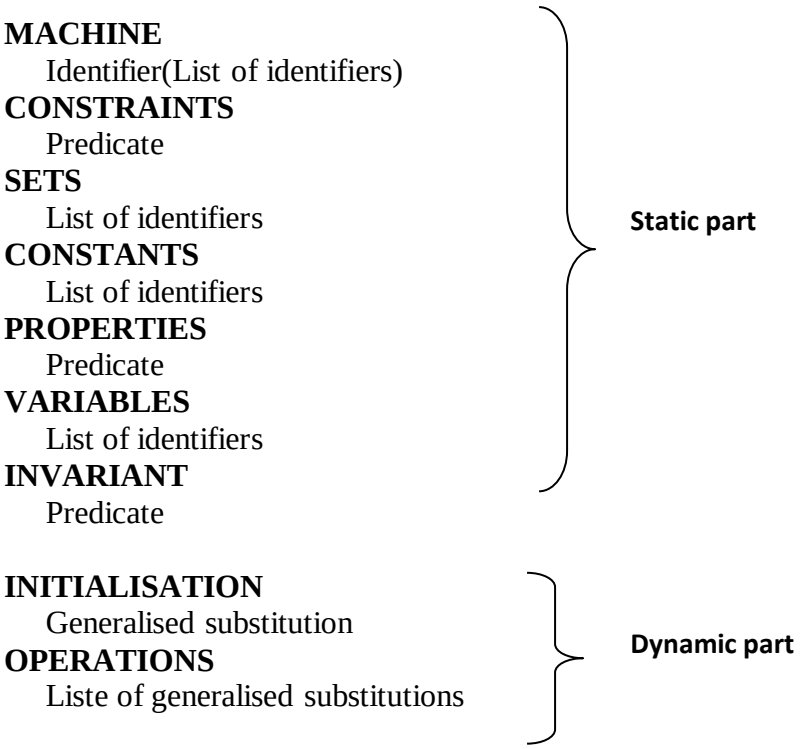


Fig9 : Syntaxe d'une Machine Abstraite [18]

II.2.2. Clauses d'une Machine Abstraite

Clauses	Description
MACHINE	Nom et paramètres de la machine.
CONSTRAINTS	Définitions des propriétés des paramètres de la machine.
SETS	Liste des ensembles abstraits et définition des ensembles énumérés.
CONSTANTS	Liste des constantes de la machine.
PROPRERTIES	Définition des propriétés des constantes et des ensembles.
VARIABLES	Liste des variables d'état de la machine.
INVARIANT	Définition des types et des propriétés des variables.
INITIALISATION	Initialisation des variables d'état.
OPERATIONS	Liste des opérations de la machine

Exemple : Réservation des places

MACHINE
Reservation (max_siege)
CONSTRAINTS
Max_siege : NAT
VARIABLES
nb_siege_disp
INVARIANT
nb_siege_disp : 0.. max-sieges
INITIALISATION
nb_siege_disp :=max_siege
OPERATIONS
Reserver=
PRE
0< nb_siege_dispo
THEN
nb_siege_dispo := nb_siege_dispo – 1
Annuler=
PRE
nb_siege_dispo< max_siege
THEN
nb_siege_dispo := nb_siege_dispo + 1

Fig 9: Exemple d'une Machine Abstraite

II.3. langage pour la partie statique

C'est le langage de description des données et de leurs propriétés. Il est basé sur :

- ✓ La théorie des ensembles : ensembles, relations, fonctions, séquences, etc.
- ✓ La logique des prédicats du 1er ordre .

II.3.1. Typage des données :

Le typage en B est un mécanisme de vérification statique des données et des expressions du langage B. Les types possibles dans le langage B sont [1] :

- ✓ **les types de base** :
  1. BOOL={TRUE,FALSE} avec TRUE≠FALSE
  2. NAT=0.. MAXINT
  3. NAT_1=1.. MAXINT
  4. INT= MININT.. MAXINT
  5. CHAR : l'ensemble des caractères.
  6. STRING : l'ensemble des chaînes de caractères.

- ✓ **les types construits** :
  Soient T, T1, T2, T3, T4 des types. Dans le langage B il existe trois constructeurs de types **:**

  ✓ **Le produit cartésien** : noté '×' ,
  T1 × T2 désigne le produit cartésien des ensembles T1 et T2, c'est-à-dire l'ensemble des paires ordonnées (x_1 ,x_2) dont x_1 est de type T1 et x_2 est de type T2.
  Remarque : L'opérateur '×' étant associatif à gauche, le type T1 × T2 × T3 × T4 désigne en fait le type ((T1 × T2) × T3) × T4.

  ✓ **L'ensemble des parties :** noté 'P' ou 'F' ,
  P(T) désigne l'ensemble des parties de T, c'est-à-dire l'ensemble dont les éléments sont des ensembles d'éléments de T.
  - ✓ P (T) ensemble des parties de T
  - ✓ P_1 (T) ensemble non vide des parties de T
  - ✓ F (T) ensemble fini des parties de T
  - ✓ F_1 (T) ensemble fini non vide des parties de T .

  Exemple : Le type de l'expression {-5, 3, -1, 8} est P(INT).

  ✓ **Le type 'struct'**.,
  Soient :
  - ✓ n un entier supérieur ou égal à 1,
  - ✓ T1, …, Tn des types
  - ✓ $Ident_1$, …, $Ident_n$ des identificateurs.

  Alors le type : struct (Ident1 : T1, …, Identn : Tn) désigne l'ensemble formé par une collection ordonnée de n types Ti appelés champs du record. Chaque champ possède un nom $Ident_i$ appelé label.

  Exemple : Le type de l'expression rec (a : 5, b : TRUE) est
  struct (a : INT, b : BOOL) .

II.3.2. Définition de base d'ensembles

Un ensemble peut être défini explicitement comme suit [18] :

- ✓ **Manière Abstraite** : un ensemble est définit uniquement par son nom : SETS ADR
- ✓ **Manière Concrète** : énuméré les valeurs de l'ensemble :
  SETS FEUX ={rouge, vert, jaune}
- ✓ **Intervalle** : x..y ensemble fini compris entre x et y .

- ✓ **L'ensemble vide** : « Ø » ou « { } »
- ✓ **Ensemble définis par compréhension :** { x|P(x) } ensemble des valeurs de x vérifiant P: {x | x:1..5 ∧ x mod 2=0}= {2,4}
- ✓ **Par composition :** en utilisant les opérateurs ensemblistes classique :

Désignation	Notation
Union	E∪F
Intersection	E ∩ F
Appartenance | sa négation	x ∈ F | x ∉ F
Différence	E\F
Inclusion | sa négation	E ⊆ F | E ⊊ F

II.3.3. Les relations

Une relation est un cas particulier de construction d'ensembles. Elles définie comme u n sous-ensemble du produit cartésien [17] :

$$r \in E1 \leftrightarrow E2 == P(E1 \times E2)$$

✓ **Domaine d'une relation : dom(r)**

L'ensemble des éléments du premier ensemble E1 qui sont effectivement en relation avec des éléments du second ensemble E2.

$$Dom(r) = \{x \mid x \in E1 \wedge \exists y.\ (y \in E2 \wedge (x \rightarrow y) \in r\}$$

✓ **Codomaine d'une relation (range en anglais) : ran (r)**

L'ensemble des points du second ensemble E2 qui sont en relation avec des éléments du premier ensemble E1.

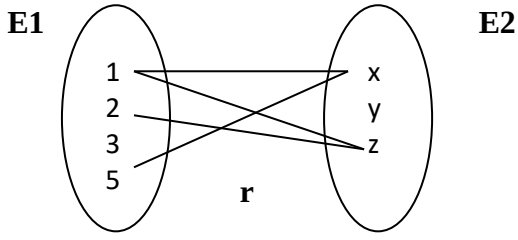
$$Dom(r) = \{y \mid y \in E2 \wedge \exists x.\ (x \in E1 \wedge (x \rightarrow y) \in r\}$$

✓ **L'image d'un ensemble par relation : r[F]**

L'ensemble des éléments de E2 qui sont en relation avec les éléments de F par la relation r.

$$r[F] = \{y \mid y \in E2 \wedge \exists x.\ (x \in F \wedge (x \rightarrow y) \in r\}$$

Exemple :



$$r \in E_1 \leftrightarrow E_2 = \{1 \rightarrow x, 1 \rightarrow z, 2 \rightarrow z, 5 \rightarrow x\}$$

✓dom(r)={1,2,5}
✓ran(r)={x,z}
✓r({2,3})={z}

II.3.3.1) Opérations sur les relations

- ✓ **Identité :** La relation identité sur un ensemble, qui à chaque élément associe le même élément. :**Id (E) ={ x→ x | x∈ E }**
  Exemple : id(E2)={ x→ x, y→ y, z→ z}

- ✓ **Inverse d'une relation : r^{-1} ={ x→ y | y→x∈ r }**
  Exemple : r^{-1} = {x→ 1, z→ 1, z→ 2, x→ 5}

- ✓ **La composition séquentielle : p :q ={ x→ z | ∃ y, x→y∈ p ∧ y→z∈ q }**
  Exemple :
  r1 = {(0 → 2),(1 → 5),(2 → 5),(2 → 7)},
  r2 = {(0 → 0),(2 → −1),(5 → 8),(6 → 9)},
  r1 ; r2 = {(0 → −1),(1 → 8),(2 → 8)}

- ✓ **Le produit direct : p ⊗ q ={ x→ (y,z) | x→y∈ q ∧ x→z∈ q }**
  Exemple :
  r1 ⊗ r2 = {0 → (2,0), 2 → (5,−1), 2 → (7,−1)}

- ✓ **Le produit parallèle : p || q ={ (x,z)→ (y,a) | x→y∈ q ∧ z→a∈ q }**
  Exemple :
  r3 = {(0 → 2),(1 → 5)}
  r4 = {(3 → 4),(6 → −1)},
  r1 || r2 = {(0,3)→(2,4) , (0,6)→(2,-1), (1,3)→(5,4) ,(1,6)→(5,-1)}

- ✓ **Itération : r ∈ E_1 ↔ E_2**

Notation	Signification	Définition
r^n	Itération n fois de r	r^n= r ; r^{n-1} si n>0 r^n= id (E_1) si n=0
r^+	Fermeture transitive	r^+= ∪n (n∈ Nat_1 | r^n)
r^*	Fermeture réflexive transitive	r^*= ∪n (n∈ Nat | r^n)

Exemple : r = {(0→ 2),(1→ 0),(2→ 1)}, E1=E2= {0,1,2}
r^2 = r ; r^{2-1} = r ; r = {(0 → 1),(1 → 2),(2 → 0)}
r^3= r ; r^{3-1} = r ; r^2 = {(0 → 0),(1 → 1),(2 → 2)}
r^4= r ; r^3 ={(0→ 2),(1→ 0),(2→ 1)}= r
r^*= r^0∪r∪r^2∪r^3= {(0→0),(0→1),(0→2),(1→0),(1→1),(1→2),(2→0),(2→1),(2→2)}

II.3.4. Les fonctions : Les fonctions sont des relations dont chaque élément a une image au plus.

Notation	Signification	Définition
E1⇸E2	Fonction partielle	$\{r \mid r \in E_1 \leftrightarrow E_2 \wedge \forall x, y, z \ (x \rightarrow y \in r \wedge x \rightarrow z \in r \Rightarrow y=z)\}$
E1→ E2	Fonction Totale	$\{f \mid f \in E_1 \nrightarrow E_2 \wedge dom(f)= E1\}$
E1 ⤔ E2	Injective partielle	$\{f \mid f \in E_1 \nrightarrow E_2 \wedge f^{-1} \in E_2 \nrightarrow E_1\}$
E1 ↣ E2	Injective totale	$\{f \mid f \in E_1 \rightarrowtail E_2 \wedge dom(f)= E1\}$
E1 ⤖E2	Surjective partielle	$\{f \mid f \in E_1 \nrightarrow E_2 \wedge ran(f)= E2\}$
E1 ↠E2	Surjective totale	$\{f \mid f \in E_1 \twoheadrightarrow E_2 \wedge dom(f)= E1\}$
E1 ⤕ E2	Bijective partielle	$\{f \mid f \in E_1 \rightarrowtail E_2 \wedge E_1 \twoheadrightarrow E_2\}$
E1⤖ E2	Bijective totale	$\{f \mid f \in E_1 \rightarrowtail E_2 \wedge E_1 \twoheadrightarrow E_2\}$

II.4. Exercices

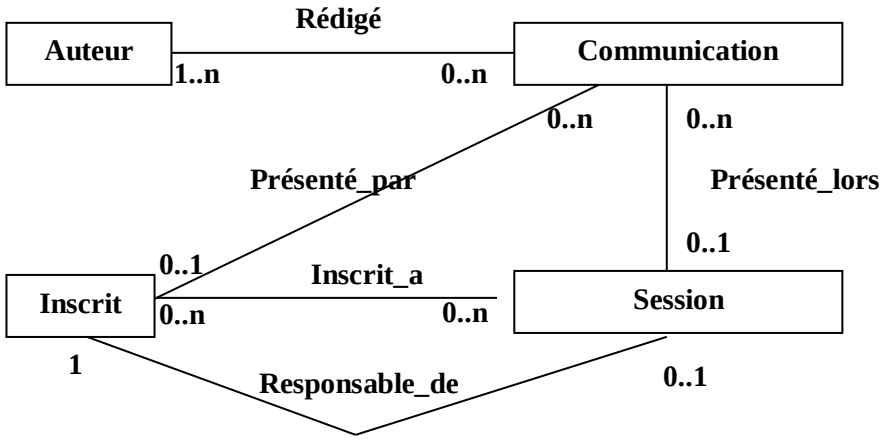
Exercice N° 6 : Étant donné un ensemble personnes, on cherche à modéliser :

- ✓ Les ensembles femmes, hommes
- ✓ Les relations epoux, mere, pere, parent, enfant, grand-parent, ancetre, frere, sœur, frere-sœur, enfant

Donner un modèle mathématique qui exprime les expressions informelles suivantes :

1. Chaque personne est un homme ou une femme.
2. Nulle personne n'est à la fois homme et femme.
3. **epoux** = {Chaque femme a au plus un époux }
4. **mere** ={La mère d'une est une femme mariée}
5. **père**={ Le père est l'époux de la mère.}
6. **Parent= ?**
7. **Enfant= ?**
8. **Ancetre= ?**

Exercice N° 7 : En utilisant les expressions d'ensemble et la logique de 1er ordre , spécifier les relation et les ensembles décrits sur le diagramme suivants :



II.5. Langage des substitutions généralisées

II.5.1.Définitions des opérations :

Dans la partie dynamique, deux clauses sont définies :

- ✓ Clause **INITIALISATION** : qui est une substitution généralisée
- ✓ Clause **OPERATIONS** : une liste d'opérations.

Les opérations ont la forme suivante :

nom op(p) = substitution généralisée

II.5.2. Notations

- wp(S,Q) pour weakest precondition. C'est une fonction à deux arguments :
  1. Un programme S
  2. Un prédicat Q
- Pour un S fixe, on peut voir wp(S,Q) comme une fonction à un seul argument Q : **wps(Q).**
- La fonction wps est appelé transformateur de prédicats : c'est une fonction qui associe à tout prédicat Q la plus faible pré-condition P telle que P {S} Q.
- Wps(Q) sera notée [S]Q.
- [S]Q représente la plus faible pré condition pour que après n'importe quelle exécution de S, la propriété Q soit vérifiée. [18]

II.5.3. Substitution généralisée

- Toutes substitutions est vue comme un transformateur de prédicat.
- Les substitutions généralisées ont une notation concise qui facilite les formules et les calculs.
- La substitution généralisée est complètement déterminée par la définition de la formule[S]P. [17]

II.5.4. Langage des substitutions généralisées :

✓**Affectation (substitution simple)**

Syntaxe : x :=E **cas particulier** : f(x) :=y
Sémantique : [x :=e]P ≡ P(e/x)
P(e/x) représente le résultat de la substitution des occurrences libres de x dans P par e (e renomme les x libres dans P).

Exemple : [n :=n+3]n>2 ≡ n+3>2
≡ n>-1

✓**Substitution multiple :**

Syntaxe : $x_1,x_2 := e_1,e_2$.
Cas général : $x_1,\ldots,x_n := e_1,\ldots,e_n$.
Sémantique : $[x_1,x_2 := e_1,e_2]P \equiv P(e_1,e_2/ x_1,x_2)$
$P(e_1,e_2/ x_1,x_2)$ représente le résultat de la substitution dans P de toutes les occurrences libres de x_1 par e_1 et simultanément de toutes les occurrences libres de x_2 par e_2 .

Exemple : [x,y :=a,b]x=y ≡ a=b

✓**Vide**
Syntaxe : skip
Sémantique : [skip]P ≡ P.

✓**Composition séquentielle**
Soit S_1 ; S_2 deux substitutions.
La composition séquentielle permet de faire une séquence de substitutions.
Syntaxe : S_1 ; S_2 .
Sémantique : $[S_1 ; S_2]P \equiv [S_1]([S_2]P)$

Exemple : [b :=a ; y :=b]x=y ≡ [b :=a]([y :=b] x=y)
≡ [b :=a](x=b)
≡ x=a

✓**Alternative**

Syntaxe : IF B THEN S1 ELSE S2 END
Sémantique : [IF B THEN S1 ELSE S2 END]P ≡ B⇒[S1]P ∧ ⅂B⇒[S2]P

Exemple : [if x>0 then y:=x else y:=-x] y>5
≡ x>0 ⇒[y:=x] y>5 ∧ x≤0⇒[y:=-x] y>5
≡ x>0 ⇒x>5 ∧ x≤0⇒x< -5
≡ (x≤0 ∨ x>5) ∧ (x>0 ∨ x< -5)
≡ x < -5 ∨ x>5

✓**Pré-conditionnée**

Syntaxe : PRE Q THEN S END
Sémantique : [Q | S]P ≡ Q ∧ [S]P

Exemple : [x>0 | x:=x-1] -1≤x≤1
≡ x>0 ∧ -1≤ x-1≤1
≡ x>0 ∧ 0≤ x≤2
≡ 0< x≤2

✓**Choix borné**

Syntaxe : CHOICE S1 OR S2 OR ….OR Sn END
Sémantique : [CHOICE S1 ∨ S2 ∨ ….∨ Sn]P ≡ [S1]P ∧ [S2]P ∧…∧ [Sn]P

Exemple : [CHOICE x :=x-1 ∨ x :=x+1] 1≤x≤10
≡ [x :=x-1] 1≤x≤10 ∧ [x :=x+1] 1≤x≤10
≡ 1≤ x-1≤10 ∧ 1≤ x+1≤10
≡ 2≤ x0≤11 ∧ 0≤ x≤9
≡ 2≤ x0≤9

✓**Choix non borné**

Syntaxe : ANY z WHERE Q THEN S END
Sémantique : [ANY z WHERE Q THEN S END]P ≡ ∀ z (Q⇒[S]P)

Exemple : [@z.(z>0 ⇒x :=x+z)] x≥3
≡ ∀ z .([z>0 ⇒[x :=x+z] x≥3)
≡ ∀ z .(z>0 ⇒ x :=x+z ≥3)

✓ **Choix Gardée**

Syntaxe : SELECT Q THEN S END
Sémantique : [Q⇒S]P ≡ Q⇒[S]P

Exemple : [x>0 ⇒y :=y+x] y>0
≡ x>0 ⇒y+x>0
≡ x≤0 ∨ y+x>0

II.6.Exercices

Exercice N° 8 : Calculer les substitutions suivantes

- [x :=y+5]x>y
- [i :=i+1]i≤1
- [x,y :=y+1,x]x*y
- [x :=z+1 ;y :=x+z]y ∈0..5

Exercice N° 9 : En utilisant le langage B, écrire une machine abstraite qui permet de spécifier une opération d'allocation d'un mot dans la mémoire. Elle retourne l'adresse de l'emplacement alloué, s'il existe.

Chapitre IV :
Test et Qualité de logiciels

L'objectif de l'assurance qualité de mettre en œuvre un ensemble de dispositions qui vont être prises tout au long cycle de développement d'un logiciel pour accroître les chances d'obtenir un logiciel qui corresponde à sa spécification (ses objectifs). La phase de test est l'une des activités de l'assurance qualité. Son objectif est de minimiser la probabilité d'apparition d'une erreur lors de l'utilisation du logiciel.

I .Test

Le test est l'évaluation d'une application ou d'un module par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus [19].

I.1. Phases du test

1. **Test unitaire : premier niveau**
   Le test unitaire permet d'assurer que chaque unité de programme (méthode, classe, composant) conformes à sa spécification, indépendamment du reste du système. Les tests unitaires sont réalisés par les développeurs des modules.
   Cible : les méthodes et les classes. [20]

2. **Test d'intégration : deuxième niveau**
   Le but et de tester les interfaces entre modules . Ce test porte principalement sur la vérification des enchaînements entre modules, la circulation des données, les aspects dynamiques, les séquences d'événements prévus et les reprises en cas d'interruption. Ils sont réalisés par les développeurs des modules ou autres développeurs [20]
   Cible : les composants et le système global.

3. **Test système : niveau trois.**
   Sert à tester l'application dans des conditions opérationnelles (son futur site d'exploitation) . Ils sont réalisés par les développeurs des modules [20].
   Cible : l'application finale sur le site d'exploitation.

4. **Le test d'acceptation : dernière étape**
   Cette étape sert a testé l'application dans des conditions précisé par les utilisateurs (les clients) . Ce type de test est réalisé par le client.[20]
   Remarque : Les tests d'acceptation révèlent souvent des omissions ou des erreurs dans la définition de besoins (erreurs de validation et non de vérification)

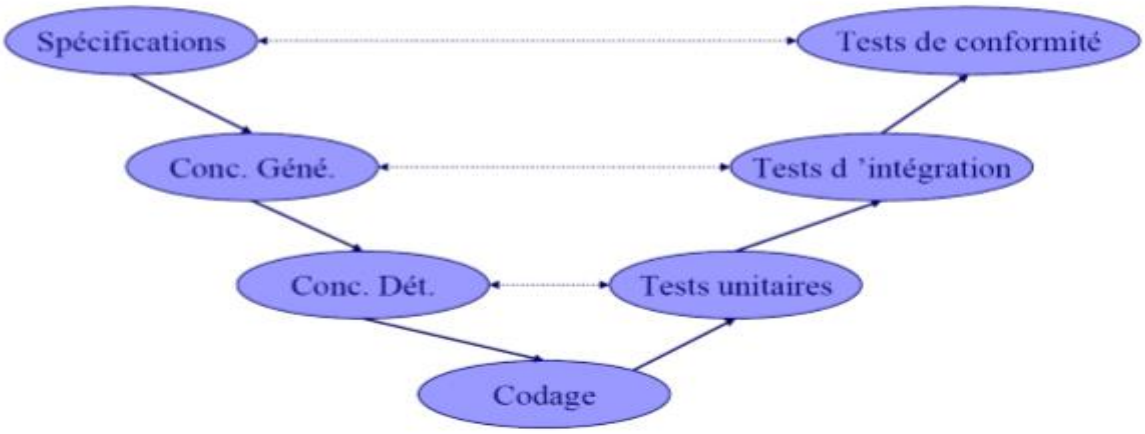


Fig 10 : Phases de test [20]

I.2. Processus simplifié de test

Le processus de teste est composé de cinq phases [22] :

- **Etape 1 : Définition d'un t un cas test (CT) :**
  Nous commençons tout d'abord par la précision scénario que l'on veut appliquer.Et estimer le résultat attendu du CT (Oracle[4])
- **Etape 2 : Concrétisation du CT**
  Le CT est concrétisé par définition des données de test (DT) .
- **Etape 3 : Concrétisation de l'Oracle**
  Déterminer le résultat attend pour ce CT.
- **Etape 4 : Exécution**
  Exécuter le script de test correspond au CT sur les DT.
- **Etape 5** : **Comparaison et évaluation**
  Compare le résultat de l'exécution (obtenu) à la prédiction de l'Oracle.

[4] Un oracle est un mécanisme permettant de décider la réussite d'un scénario de test, c'est à dire de déterminer si les réponses obtenues à l'exécution du test correspondent bien à ce que requiert le scénario.

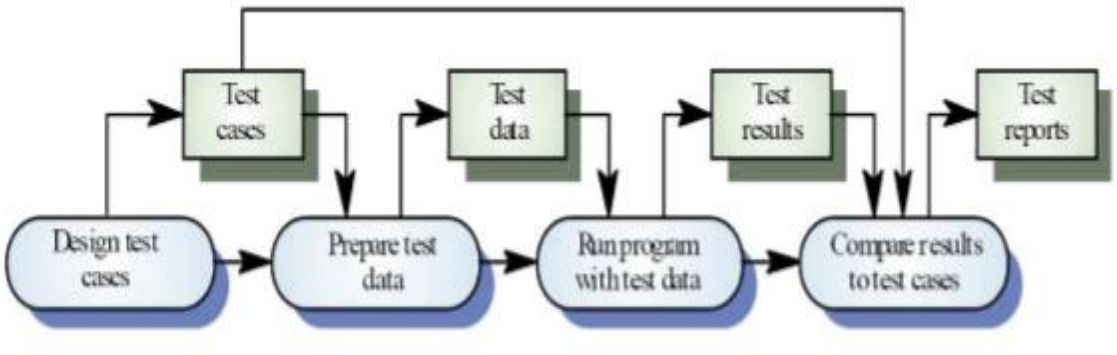


Figure 11 : Processus de test [22]

Exemple : : tri de tableaux d'entiers avec suppression des éléments répétés
Interface : int[] my-sort (int[] vec)

Etape 1 : Quelques cas de tests (CT) et leurs oracles

Cas de test	Oracle
Cas N°1 : tableau d'entiers non redondant	Le tableau trié
Cas N°2 : tableau vide	Le tableau vide
Cas N°3 : tableau avec des entiers redondants	Le tableau est trié sans redondance

Etape 2: concrétiser le cas test (DT)

Données de test
Cas N°1 : vec = [5,3,15]
Cas N°2 : vec = []
Cas N°3 : vec = [10, 20, 30, 5, 30,0]

Etape 3: concrétiser l'oracle

Données de test
Cas N°1 : res= [3, 5,15]
Cas N°2 : res = []
Cas N°3 : res = [0,5,10,20,30]

Etape 4 & Etape 5: exécuter le script sur un cas des tests et comparer le résultat obtenu par le résultat attendu (étape 3).

I.4. Sélection de tests :

Deux grandes familles de sélection de tests

- **Test fonctionnel ou boîte noire :** le cas de test est conçu à partir des spécifications du logiciel (use-case). Le concepteur du test n'a pas accès au code source ou a choisi de ne pas regarder la façon dont le logiciel a été écrit : c'est pour ce là qu'on parle également de test en boîte noire.
  ⇒Uniquement en fonction des entrées et des sorties

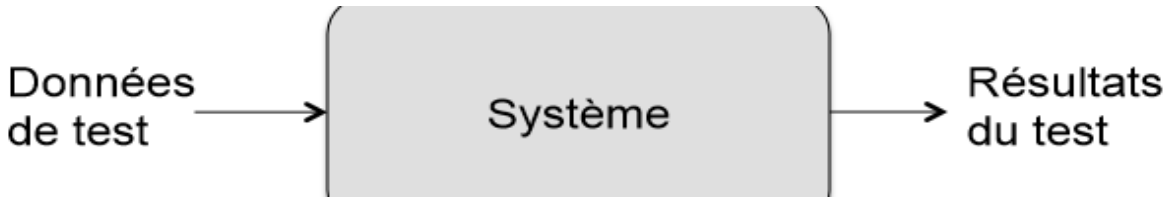


Fig12 : Test Fonctionnel [21]

- **Test structurel ou boîte blanche :** les cas de test sont conçus à partir du code source (on parle également de test en boîte blanche). Les données de tests sont alors produites après analyse du code.
  ⇒Les données de tests sont produites après analyse du code

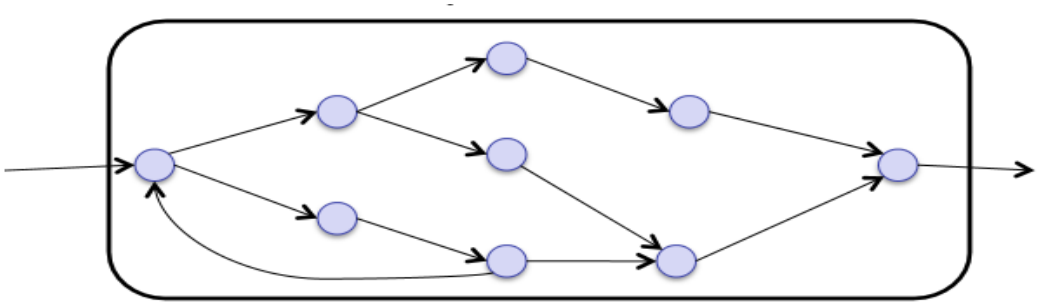


Fig13 : Test Structurel [21]

II. Qualité

II.1.Définition :

La qualité sert à obtenir de la satisfaction durable du client, en garant ses besoins et attentes, au sein d'une entreprise s'engageant à améliorer constamment son rendement et son efficacité [23].

II.2. Différent types et niveaux de qualité

En général; il existe une double vision pour la qualité:

- Vision du développeur (organisme).
- Vision du client.

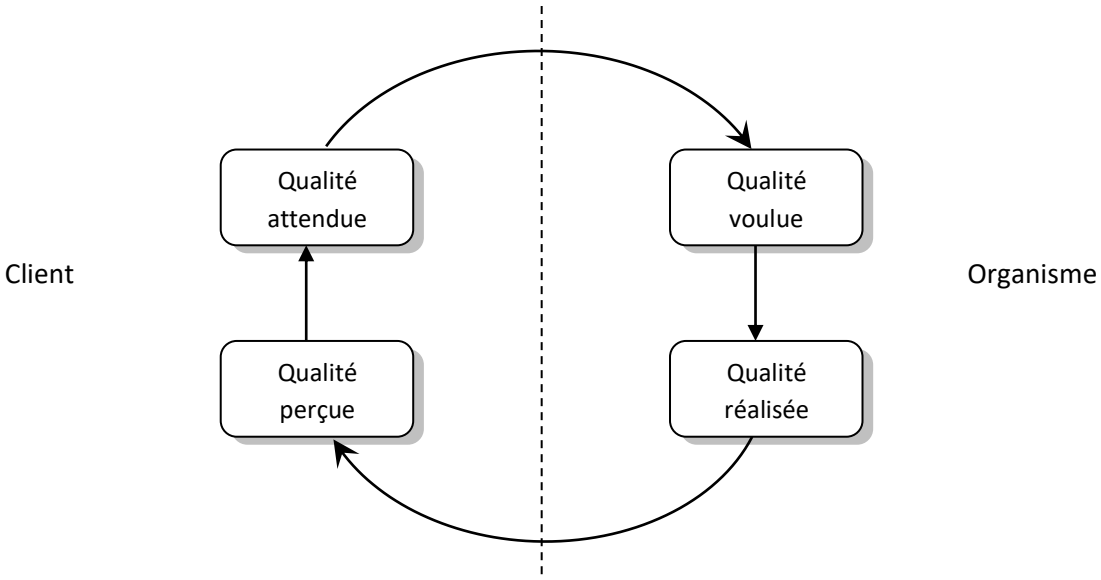


Fig14 : Différent type de qualité [24]

Du point de vue de l'organisme, nous distinguons **[24]**:

- **La qualité voulue :** c'est la première vue de l'organisme. Elle représente la qualité que l'entreprise souhaite atteindre pour répondre à la qualité attendue. C'est la prestation qu'il veut assurer à ses clients.
- **La qualité réalisée :** représente la qualité effectivement réalisée par l'organisme.
-

Du point de vue du client, nous distinguons : [24]

- **La qualité attendue :** c'est la qualité souhaitée par le client (la réponse à ses besoins et attentes) .
- **La qualité perçue :** c'est la qualité ressentie de façon plus ou moins confuse par les clients à partir de leurs besoins et attentes. Elle est l'expression de la satisfaction des clients .

II.3. Les aspects standards de la qualité

- **Facteurs de qualité :** caractéristique du logiciel qui contribue à sa qualité. Ils concernent les caractéristiques d'utilisation liées à : l'environnement d'exploitation et l'environnement de suivi et de maintenance. Les facteurs traduisent la vision externe que peut en avoir le demandeur. [27]
- **Critères de qualité :** Attribut du logiciel par l'intermédiaire duquel un facteur peut être évalué .Ils sont :
  1. Orientés réalisateur.
  2. Les composantes des facteurs de qualité.
  3. Reliés à des métriques.

Les critères de qualités concernent les caractéristiques d'utilisation en fonction d'une vision interne (structure du logiciel).

- **Métriques de qualité:** Les critères de qualité sont reliés à des métriques qui sont des mesures à posteriori.

Il existe trois types de métrique pour mesurer les critères :

1. **Présence :** Cette métrique identifie, qu'un attribut soit présent dans un composant ou non. elle est décrite par une valeur de type booléen.
2. **dénombrement :** Cette métrique est employée pour indiquer des valeurs exactes. Elle est décrite par une variable de type entier.
3. **Ratio :** Cette métrique est employée pour décrire des pourcentages.

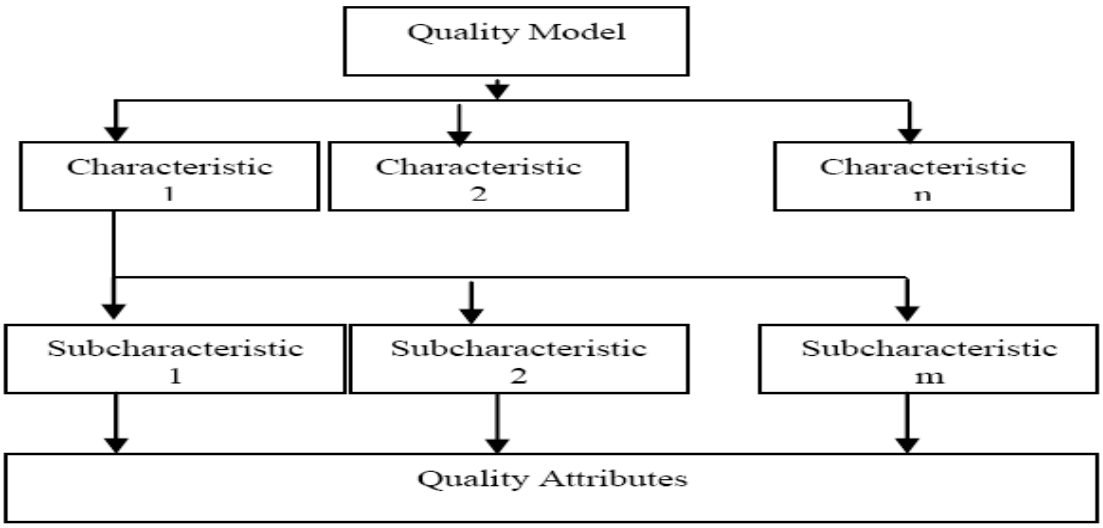


Figure 15 : Les aspects standards de la qualité [27]

II.4. Modèles de qualité

Un modèle de qualité représente l'ensemble des caractéristiques et les relations entre eux qui fournissent la base pour la spécification et l'évaluation des exigences de qualité [26] .

On peut trouver plus de 300 standards élaborés et certifiés par plus de 50 organisations différentes. Le modèle ISO / IEC 9126-1 est le modèle le plus populaire. Il spécifie les exigences pour le système de gestion de la qualité au sein d'une entreprise. ISO 9126 est un modèle de qualité générique et peut être appliqué à tout produit logiciel en l'adaptant à un but bien précis. [26]

II.5. Le Standard ISO 9162

Le standard ISO 9126 définit six facteurs de qualité : Fonctionnalité, Fiabilité, Utilisabilité, Rendement, Maintenabilité et la Portabilité [26] :

1. **Capacité fonctionnelle (Functionality) :** cette caractéristique permet de vérifier si l'application (ou une partie de l'application) en question satisfait aux exigences fonctionnelles.
2. **Fiabilité (Reliability)** : La capacité d'un système à maintenir le niveau de service spécifié dans des conditions bien déterminé.
3. **Facilité d'utilisation (Usability)** : la capacité d'une application à être compris, appris et réutilisable pour les utilisateurs, selon des événements spécifiques d'utilisation.
4. **Rendement (Efficiency) :** la capacité d'une application à fournir le niveau attendu de performances, en fonction des ressources utilisées selon des conditions fixées.
5. **Maintenabilité (Maintainability) :** la capacité de modifier une application. Les modifications impliquent les corrections, les améliorations et l'adaptation lors d'une modificationd'environnement, d'exigences ou de spécifications fonctionnelles.
6. **Portabilité (Portability) :** la capacité d'exécution d'une application sur des environnements hétérogène.

- A chaque facteur de qualité un ensemble de critères est défini :

Caractéristiques (Facteurs)	Sous_Caractéristiques (Critères)
Fonctionnalité	Exactitude , Sécurité, Aptitude, Interopérabilité, Conformité
Fiabilité	Maturité, Tolérance de fautes, Récupération
Utilisabilité	Configurabilité , Compréhension ,Facilité d'exploitation
Efficacité	Comportement de temps, Comportement de Ressource , Scalabilité
Maintenabilté	Facilité d'analyse, Facilité de modification , Stabilité, Facilité de test.
Portabilité	Déployabilité , Adaptabilité, Réutilisabilité

Tableau 1 : le standard de qualité ISO 9126 [26]

Selon le standard "ISO/IEC 9126", les attributs constituent le niveau le plus bas de la hiérarchie et sont mesurables par des métriques. Par exemple les métriques associées aux attributs auditabilité et contrôlabilité du critère sécurité sont définies comme suit [27] :

Nom	But	Formule de mesure	Interprétation de la valeur mesurée	Type de mesure
Auditabilité	Évaluer le taux d'accès que le système a enregistrés dans la base d'accès.	X= A / B A= nombre "de tentatives d'accès au système et aux données » enregistré dans la base d'accès. B= nombre "de tentatives d'accès au système et aux données » faites durant l'évaluation.	0<=X<=1 Plus proche à 1 est le meilleur.	A : dénombrement B : dénombrement
Contrôlabilité	Compte le nombre d'opérations illégales détectées	X= A / B A= nombre d'opérations illégales détectées. B= nombre d'opérations illégales spécifiées	0<=X<=1 Plus proche à 1 est le meilleur.	A : dénombrement B : dénombrement

Annexe : Solution des Exercices

Exercice N°1 :

1. {P} x :=3 {x+y>0}

{ ?? } x := 3 { x+y>0}
wp(x := 3, x+y>0)= [3/x](x+y>0)
= 3+y>0
➔**{3+y>0} x :=3 {x+y>0}**

_____ aff
{3+y>0} x :=3 {x+y>0}

2. {P} x :=3*y+z {x*y-z>0}

{ ?? } x :=3*y+z {x*y-z>0}
wp(x := 3*y+z, x*y-z >0)= [3*y+z /x](x*y-z >0)
= (3*y+z)*y-z >0
= 3*y^2+z*y-z >0

➔**{3*y^2+z*y-z >0} x :=3*y+z {x*y-z>0}**

_____ aff
{3*y^2+z*y-z >0} x :=3*y+z {x*y-z>0}

3. {P} x :=x+1 ;y :=x+y {y>5}

wp(**x :=x+1 ;** y :=x+y, **y>5**) = wp(**x :=x+1 , wp(**y :=x+y, **y>5)**)
= wp(**x :=x+1 ,** [**x+y**/y](**y>5**))
= wp(**x :=x+1 , x+y>5**)
= [**x+1/x](** **x+y>5**)
= x+y>4

{ x+y>4} x :=x+1{ x+y>5} { x+y>5} y :=x+y {y>5} seq
{ x+y>4} x :=x+1 ;y :=x+y {y>5}

Exercice N°2 :

{a=x && y=b}
x :=x-y ;
{Q''} **PR2**
y :=x+y ;
{Q'} **PR1**
x :=y-x ;
{y=a && x=b}

I) PR1

{Q'} x :=y-x ; **{y=a && x=b}**

wp(x :=y-x , y=a && x=b)= [y-x/x](y=a && x=b)
= y=a && y-x=b
affect➔{ y=a && y-x=b } x :=y-x { y=a && x=b }

{a=x && y=b}
x :=x-y ;
{Q''} **PR2**
y :=x+y ;
{ y=a && y-x=b }
x :=y-x ;
{y=a && x=b}

$$\frac{}{\{ y=a \;\&\&\; y-x=b \}\; x := y-x \;\{ y=a \;\&\&\; x=b \}}\ \text{aff}$$

Exercice N°3 :

Solution

{x≥-100 && x≤ 100}
Q'' PR2
If x<0 then x :=x+100 ;
Q' PR1
y :=x*2 ;
{y≥0 && y≤ 300}

I) PR2

{Q'} y :=x*2 ; **{** y≥0 && x≤ 300**}**

wp(y :=x*2 , y≥0 && y≤ 300)= [x*2 /y](y≥0 && y≤ 300)
= x≥0 && x≤ 150
affect➔{ x≥0 && x≤ 150} y :=x*2 { y≥0 && y≤ 300}

Q'' PR2
If x<0 then x :=x+100 ;
{x≥0 && x≤ 150}
y :=x*2 ;
{y≥0 && y≤ 300}

$$\frac{}{\{ x\geq 0 \;\&\&\; x\leq 150 \}\; y := x*2 \;\{ y\geq 0 \;\&\&\; y\leq 300 \}}\ \text{aff}$$

I) PR2

Q'' if x<0 then x :=x+100 else skip ; { x≥0 && x≤ 150}
wp(if x<0 then x :=x+100 else skip, x≥0 &&x≤150)
if ⇒ x<0 && wp(x :=x+100 , x≥0 &&x≤150) || x≥0 && wp(skip, x≥0 &&x≤150)
⇒ x<0 && (x| x+100)[x≥0 &&x≤150] || x≥0 && x≥0 && x≤150)
Q''= (x<0 && x≥-100 && x≤50) || (x≥0 && x≥0 && x≤150)

Cons : (x<0&& x≥-100) ⇒ (x<0 && x≥-100 && x≤50)
Cons: (x≤150 && x≥0) ⇒ (x≥0 && x≥0 && x≤150)

Q''= (x≥-100 && x<0) || (x≥0 && x≤150)
Cons : Q''= x≥-100 && x≤150

(x≥-100 && x≤150)
If x>0 then x :=x+100 ;
{ x≥0 && x≤ 150}
y :=x*2 ;
{y≥0 && y≤ 300}

(x≤0&& x≥-100) x :=x+100 { x≥0 && x≤ 150} (x≤150 && x≥0) skip { x≥0 && x≤ 150} if
(x≥-100 && x≤150) if x>0 then x :=x+100 else skip ; { x≥0 && x≤ 150}

.....(I)

x≥-100 && x≤ 100 ⇒ x≥-100 && x≤150 && (I) ConsG
{ x≥-100 && x≤ 100 } if x>0 then x :=x+100 else skip ; { x≥0 && x≤ 150}

.....(II)

Séquence

x≥-100 && x≤ 100 ⇒ x≥-100 && x≤150 && (I) ConsG aff
{ x≥-100 && x≤ 100 if x>0 then x :=x+100 else skip ;{ x≥0 && x≤ 150} { x≥0 && x≤ 150} } y :=x*2 { y≥0 && y≤ 300} seq
{ x≥-100 && x≤ 100 if x>0 then x :=x+100 else skip ; y :=x*2 { y≥0 && y≤ 300}

Le triplet est vrai.

Exercice N°4 :

{x≥0 && y≥0 }

q :=0 ;
r :=x ;

while r≥y do
r :=r-y ;
q :=q+1 ;
end ;

PR2
PR3
PR1

{ x=q*y+r && 0≤r<y }

- **PR1**

L'Invariant :

Hypothèse : I :: x=q*y+r && 0≤r

Exemple : x=5 y=2

q	**0**	**1**	**2**
r	**5**	**3**	**1**
x=q*y+r && 0≤r	✓ **5=0**2+5** ✓ **0≤5**	✓ **5=1**2+3** ✓ **0≤3**	✓ **5=2**2+1** ✓ **0≤3**

Hypothèse vrais

{ x=q*y+r && 0≤r }
while r≥y do
r :=r-y ;
q :=q+1 ;
end ;
{ x=q*y+r && 0≤r<y }

$$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ \text{tant que b faire S}\ \{I \wedge \neg b\}}\ \text{loop}$$

???????

{ x=q*y+r && 0≤r && r≥y }
r :=r-y ;
q :=q+1 ;
{ x=q*y+r && 0≤r }

Wp (r :=r-y ; q :=q+1, **x=q*y+r && 0≤r) = Wp (**r :=r-y ; **x=** (q+1) ***y+r && 0≤r)**
= x= (q+1) ***y+** r-y **&& 0≤** r-y **)** = **x =** q ***y+** r **&& r≥y**

Sous Arbre 1

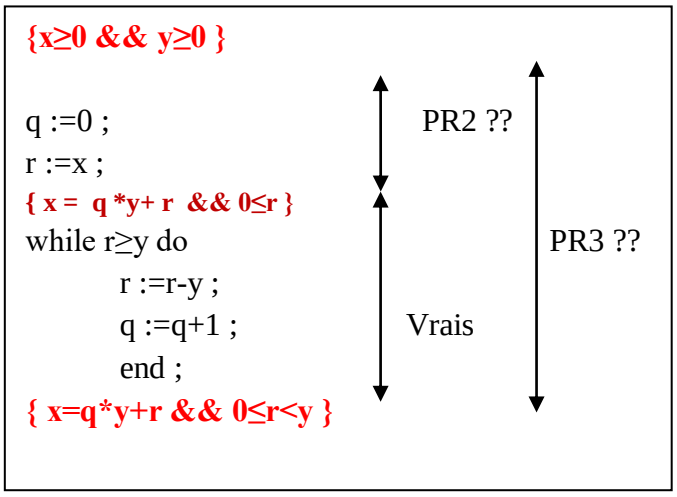
$$\frac{\{\ x = q*y+r\ \&\&\ r\geq y\ \}\ r:=r-y\ \{\ x=(q+1)*y+r\ \&\&\ 0\leq r\ \}\ \text{Aff} \qquad \{\ x=(q+1)*y+r\ \&\&\ 0\leq r\ \}\ q:=q+1\ \{\ x=q*y+r\ \&\&\ 0\leq r\ \}\ \text{aff}}{\{\ x = q*y+r\ \&\&\ r\geq y\ \}\ r:=r-y\ ;\ q:=q+1\ ;\ \{\ x=q*y+r\ \&\&\ 0\leq r\ \}}\ \text{seq}$$

{ x=q*y+r && 0≤r && r≥y } => x = q ***y+** r **&& r≥y**

Sous Arbre2

$$\frac{\{\ x=q*y+r\ \&\&\ 0\leq r\ \&\&\ r\geq y\ \}\ \Rightarrow\ x = q*y+r\ \&\&\ r\geq y \qquad \text{Sous Arbre I}}{\{\ x=q*y+r\ \&\&\ 0\leq r\ \&\&\ r\geq y\ \}\ r:=r-y\ ;\ q:=q+1\ ;\ \{\ x=q*y+r\ \&\&\ 0\leq r\ \}}\ \text{CONSG}$$

$$\frac{}{\{\ x = q*y+r\ \&\&\ 0\leq r\ \}\ \text{while}\ (\ r\geq y)\ \ r:=r-y\ ;\ q:=q+1\ ;\ \{\ x=q*y+r\ \&\&\ \mathbf{0\leq r<y}\ \}}\ \text{LOOP}$$



PR2 ??

{x≥0 && y≥0 }
q :=0 ;
r :=x ;
{ x = q *y+ r && 0≤r }
Wp (q :=0 ; r :=x, **x=q*y+r && 0≤r) = Wp (**q :=0 , **x=q*y+x && 0≤x)**
= (x=x && 0≤x)

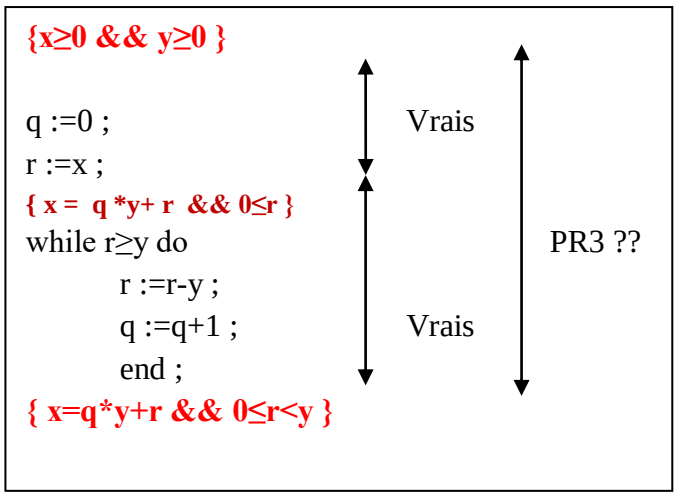
Sous Arbre 3

{x≥0 && y≥0 } =>{x=x && 0≤x } Aff — **{ x≥0 && y≥0 }** q :=0 **{ x=q*y+x && 0≤x }** Consg

aff — **{ x=q*y+x && 0≤x }** r :=x { x=q*y+r && 0≤r }

{x≥0 && y≥0 } q :=0 **{ x=q*y+x && 0≤x}**

seq — **{ x≥0 && y≥0 }** q :=0 ; r :=x1 ; { x=q*y+r && 0≤r }



PR3 : Séquence Sous arbre 2 et sous arbre 3

Sous Arbre 3 Sous Arbre 2

_____ seq

{ **x≥0 && y≥0** } q :=0 ; r :=x1 ; while r≥y do r :=r-y ; q :=q+1 ; { x=q*y+r && 0≤r<**y** }

II)PR2

{Q''} y :=x+y ; **{ y=a && y-x=b }**
wp(y :=x+y , **y=a && y-x=b**)= [x+y/y](**y=a && y-x=b**)
= x+y =a && y =b
affect➔{ x+y =a && y =b } y :=x+y { y=a && y-x=b }

{a=x && y=b}
x :=x-y ;
{ x+y =a && y =b }
PR2
y :=x+y ;
{ y=a && y-x=b }
x :=y-x ;
{y=a && x=b}

_____ **aff**
{ x+y =a && y =b } y :=x+y { y=a && y-x=b }

III)PR3

(PR1) and (PR2) => { x+y =a && y =b } y :=x+y ; x :=y-x { y=a && x=b }

_____ **Aff** _____ **aff**
{ x+y =a && y =b } y :=x+y { y=a && y-x=b } **{ y=a && y-x=b } x :=y-x { y=a && x=b }**
_____ **seq**
{ x+y =a && y =b } y :=x+y ; x :=y-x ; { y=a && x=b }

IV)PR4

{Q''} x :=x-y ;{ x+y =a && y =b }
wp(x :=x-y , x+y =a && y =b)= [x-y /x](x+y =a && y =b)
= x =a && y =b
affect➔{ x =a && y =b } x :=x-y ;{ x+y =a && y =b }

{a=x && y=b}
x :=x-y ;
{ x+y =a && y =b }
y :=x+y ;
{ y=a && y-x=b }
x :=y-x ;
{y=a && x=b}

_____ **aff**
{ x =a && y =b } x :=x-y ;{ x+y =a && y =b }

V) PR5

(PR3) and (PR4) => { x =a && y =b } x :=x-y ;y :=x+y ; x :=y-x { y=a && x=b }

$$\dfrac{\dfrac{\dfrac{}{\{ x+y =a \;\&\&\; y =b \}\ y :=x+y\ \{ y=a \;\&\; y-x=b \}}\ \text{Aff}\quad \dfrac{}{\{ y=a \;\&\; y-x=b \}\ x :=y-x\ \{ y=a \;\&\; x=b \}}\ \text{aff}}{\{ x =a \;\&\&\; y =b \}\ x :=x-y\ ;\{ x+y =a \;\&\&\; y =b \}}\ \text{aff}\quad \dfrac{}{\{ x+y =a \;\&\&\; y =b \}\ y :=x+y\ ;\ x :=y-x\ ;\ \{ y=a \;\&\&\; x=b \}}\ \text{seq}}{\{ x =a \;\&\&\; y =b \}\ x :=x-y\ ;y :=x+y\ ;\ x :=y-x\ \ \{ y=a \;\&\&\; x=b \}}\ \text{seq}$$

Exercice N° 6 :

Donner un modèle mathématique qui exprime les expressions informelles suivantes :

1. Chaque personne est un homme ou une femme.
   hommes ⊆ personnes
   femmes ⊆ personnes
   personnes ⊆ hommes ∪ femmes

2. Nulle personne n'est à la fois homme et femme.
   hommes ∩ femmes = ∅

3. epoux = {Chaque femme a au plus un époux }
   epoux ∈ femmes ⇸ hommes
   epouse = $epoux^{-1}$

4. mere ={La mère d'une est une femme mariée}
   mere ∈ personnes ⇸ dom(epoux)

5. père={ Le père est l'époux de la mère.}
   pere = mere; epoux

6. Parent= ?
   parent = mere ∪ père

7. Enfant= ?
   enfant = parent−1

8. Ancetre= ?
   ancetre = parent; parent∗

9. Démontrer mere = pere; epoux−1.
   pere;epoux−1
   (mere; epoux); epoux−1=
   mere; (epoux; epoux−1)=
   mere; id(dom(epoux)) =
   mere

Exercice N° 7 : En utilisant les expressions d'ensemble et la logique de 1er ordre , spécifier les relation et les ensembles décrits sur le diagramme suivants :

SETS INSCRIT,AUTEUR, SESSION, COMMUNICATION

VARIABLES inscrits, session, communication, auteur, inscrit_a,responsable_de, redige, presente_par, presente_lors,

INVARIANTS communication⊆ COMMUNICATION∧ inscrit⊆ INSCRIT ∧ auteur⊆ AUTEUR ∧ inscrit_a ∈ inscrit ↔ session ∧ responsable_de ∈ inscrit >⇸ session ∧ a_redige ∈ auteur ↔ communication∧ rang(a_redige)=communication ∧ presente_par ∈ communication ⇸inscrit ∧ presente_lors ∈ communication ⇸session ∧

Exercice N° 8: Calculer les substitutions suivantes

- [x :=y+5]x>y
  ≡ y+5>y
  ≡ 5>0
  ≡ true

- [i :=i+1]i≤1
  ≡ i+1≤1
  ≡ i≤0

- [x,y :=y+1,x]x*y
  ≡ (y+1)*x

- [x :=z+1 ;y :=x+z]y ∈0..5
  ≡[x :=z+1]([y :=x+z]y ∈0..5)
  ≡[x :=z+1](x+z ∈0..5)
  ≡z+1 +z ∈0..5
  ≡2z ∈-1..4

Exercice N° 9 : En utilisant le langage B, écrire une machine abstraite qui permet de spécifier une opération d'allocation d'un mot dans la mémoire. Elle retourne l'adresse de l'emplacement alloué, s'il existe.

MACHINE MEMORY
SETS
ADDRESSES //ensemble abstrait d'adresses
CONSTANTS
mem,null
PROPERTIES
mem ⊆ ADDRESSES ∧ null ∈ ADDRESSES ∧ null ∉ mem
VARIABLES
free
INVARIANT

free ⊆ mem

INITIALISATION

free :=mem

OPERATIONS

rr ← alloc =

IF free ≠∅ THEN

ANY vv WHERE vv ∈ free

THEN free,rr :=free −{vv},vv

END

ELSE

rr :=null

END;

[1] Renaud Keriven et Pascal Monasse, '***La Programmation Pour. . .***', Cours de l'École des Ponts ParisTech - 2020/2021

[2] C. Recanati , '***Programmation Impérative Polycopié de cours n° 1***', Cours de Institut Galilée- 2006/2007

[3] Fatima Amounas, '***Algorithmique & Programmation II***', Support du cours , Université Moulay Ismail- 2020/2021

[4] *https://sites.google.com/site/cours1900/master-01-cours-1/paradigmes-des-langages-de-programmation* , Consulté le 02/01/2023 à 14h00.

[5] Peter Van Roy **,** '***Les principaux paradigmes de programmation***' , conférence UPMC, Université catholique de Louvain Louvain-la-Neuve, Belgium, 2008.

[6] Raphael Grevisse Yende, '***Support De Cours De Genie Logiciel',*** HAL open source, Id: cel-01988734 , https://hal.science/cel-01988734, Submitted on 22 Jan 2019

[7] https://complex-systems-ai.com/algorithmique/terminaison-et-correction/ **,** Consulté le 02/01/2023 à 14h00.

[8] : Stefano Guerrini, '*Sémantique des Langages de Programmation Introduction*' , support de cours , LIPN - Institut Galilée, Université Paris Nord 13, 2009/2010.

[9] Bernd Kortmann, **'*Semantics: Word and sentence meaning'***, English Linguistics pp 143–171 , 2020.

[10] Pierre Courtieu , '***Outils de preuve et vérification'*** , https://cedric.cnam.fr/~courtiep/downloads/hoare.pdf, 30 octobre 2008

[11] Mike Gordon , '***Background reading on Hoare Logi'c***, , MJCG February 2, 2016

[12] Jonathan Aldrich , '***Lecture Notes: Hoare Logic'***, 15-819O: Program Analysis Jonathan Aldrich March 2013

[13] Colette Johnen, '***cours logique de Hoare'***, https://www.labri.fr/perso/johnen/pdf/IUT-Bordeaux/MF/Cours-Logique-Hoare.pdf

1. [14] 1074-1995 - IEEE Standard for Developing Software Life Cycle Processes, DOI: **10.1109/IEEESTD.1996.79663** ICS Code: **35.080 - Software**

[15] hal.archives-ouvertes.fr , '***EB4 : Vers une méthode combinée de spécification formelle des systèmes d'information'***, https://hal.archives-ouvertes.fr/hal-01124923/file/RC658.pdf

[16] https://www.atelierb.eu/outil-atelier-b/atelier-b-4/ , Consulté le 05/02/2023 à 21h00.

[17] Jean-Raymond Abrial, '***Atelier B ,Manuel Utilisateur, Version 4.0'*** , sous License Creative Commons – Paternité, janvier 2002.

[18] Christian Attiogb, '***La Méthode B' ,*** Support de Cours , Université de Nantes, 2004.

[19] '***IEEE Standard Glossary of Software Engineering Terminology'*** , Computer *Society* of the IEEE , Approved September 28,1990

[20] Patrick Felix, ***'Test et Validation du Logiciel',*** Support de Cours , IUT –Bordeaux 1 , 2009.

[21] Charles G- Colombiano K, '***Cas de test en génie logiciel'***, Support de Cours 2014.

[22] http://agile.csc.ncsu.edu/SEMaterials/AgileTesting.pdf, Consulté le 18/12/2022 à 21h00.

[23] Blanca Gil, Witold Suryn, The analysis and evaluation of ISO/IEC9126–3 internal quality measures applicability: state-of-the-art 2006. Technical report, École de technologie supérieure, Québec, 2006.

[24] Augustin RADU, Evaluation de la Qualité de Service par l'utilisateur final dans les systèmes mobiles, thèse de doctorat, Université de Marne-La-Vallée 2004.

[25] Alexandre.A et Eduardo.S et Silivio.R, Quality Attributes for a Component Quality Model, Federal University of Pernambuco and C.E.S.A.R,2006.

[26]Blanca Gil, Witold Suryn, The analysis and evaluation of ISO/IEC9126–3 internal quality measures applicability: state-of-the-art 2006. Technical report, École de technologie supérieure, Québec, 2006.

[27]ISO/IEC, IS 9126-3, Software engineering –Product quality – Part 3: Internal metrics, Technical report, ISO/IEC JTC1/SC7, 2002.