Slide 1

Une introduction aux méthodes formelles

Guy Tremblay
Dépt. d'informatique, UQAM
Montréal, Qué.

MGL7260 Exigences et spécifications de systèmes logiciels
UQAM
20 mars 2002

Slide 2

Plan de la présentation :

- Qu'est-ce qu'une spécification?
- Qu'est-ce qu'une méthode formelle?
- Qu'est-ce qu'un langage formel de spécification?
- Pourquoi y-a-t-il plusieurs notations et méthodes?
- À quelles étapes les méthodes formelles peuvent-elles être utilisées?
- Quels sont les principaux bénéfices des méthodes formelles?
- Y-a-t-il des exemples d'applications développées avec des MFs?

Slide 3

1 Prélude : Analyse, spécification et conception

Plusieurs niveaux de description d'un système :

(a) Les besoins et exigences des usagers ;

(b) L'ensemble (l'espace) des solutions possibles ;

(c) Une solution particulière ;

(d) La structure interne de la solution choisie ;

(e) La description des divers modules (interfaces) ;

(f) Les algorithmes utilisés par chaque module ;

(g) Le code final.

Slide 4

Analyse et spécification

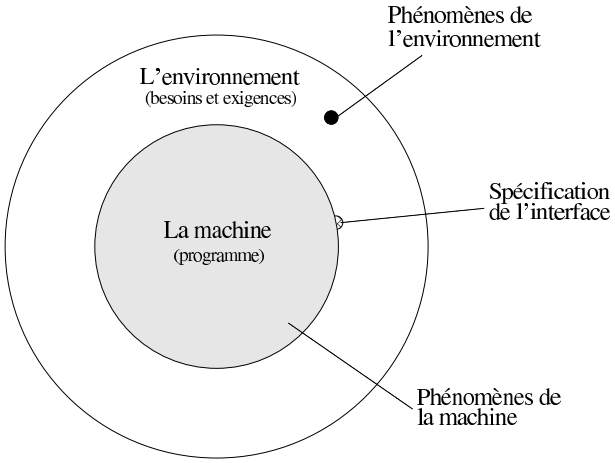
L'étape d'**analyse et spécification** porte sur :

1. Analyse du problème = Analyser et comprendre le problème à résoudre, les besoins et exigences des usagers, les contraintes à respecter = (a)+(b)
   (a) Besoins et exigences
   (b) Espace des solutions possibles
2. Spécification du produit = Décrire le logiciel qui va satisfaire aux besoins et exigences et qui va respecter les contraintes = (c)
   (c) Description d'une solution spécifique

Slide 5

Exigences et besoins vs. spécifications (selon M. Jackson) :

- Exigences et besoins = phénomènes de l'environnement
- Programmes = phénomènes internes à la machine
- Spécifications = phénomènes à l'interface entre l'environnement et la machine



Slide 6

Conception

- Conception architecturale :
  - Quels sont les principaux modules?
  - Comment sont-ils organisés et structurés?
  - Quelles sont les interfaces entre les modules?
- Conception détaillée (procédurale) :
  - Que fait précisément chacun des modules?

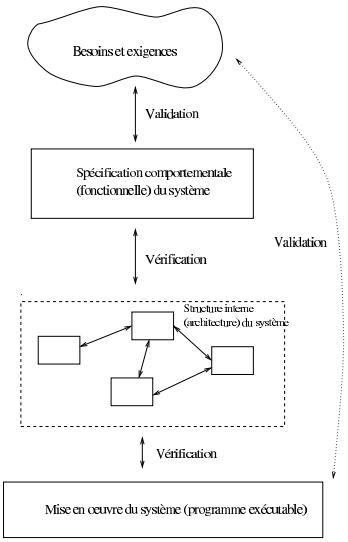
Analyse et spécification vs. conception

Donc, analyse et spécification vs. conception =>

- Analyse et spécification : Concepts et interfaces pour utiliser le système (point de vue de l'*usager*) ;
- Conception : Concepts et interfaces pour construire le système (point de vue du *constructeur*).

Slide 7

Pourquoi une spécification est-elle importante?



Spécification = *contrat à satisfaire*

- Validation : *Are we building the right product?* = Construit-on le bon produit?
- Vérification : *Are we building the product right?* = Construit-on le produit correctement?

Slide 8

2 Qu'est-ce qu'une "Méthode formelle"?

"[Les méthodes formelles sont des] techniques basées sur les mathématiques pour décrire des propriétés [de] systèmes [informatiques]. [Elles] fournissent un cadre [pour] spécifier, développer, et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc*."* [Wing90]

Éléments clés d'une méthode formelle :

- Langage *formel* pour l'écriture de spécifications
- Règles pour évaluer la validité/qualité des spécifications
- Stratégies et règles pour raffiner (mettre en oeuvre) les spécifications et vérifier ces raffinements

Fondation sur laquelle *tout* repose = spécification formelle

Slide 9

3 Qu'est-ce qu'un langage *formel* de *spécification?*

Langage avec syntaxe et sémantique bien définies :

- Syntaxe = EBNF ; diagrammes syntaxiques ; etc.
- Sémantique = algèbres ; automates et systèmes de transitions ; fonctions, relations et prédicats ; etc.

Doit permettre de décrire le comportement d'un composant logiciel

- en décrivant ses propriétés importantes
- de façon abstraite, sans détails inutiles
- sans dire comment il est réalisé (non-algorithmique)

Un langage de programmation *n'est pas* un langage de spécification :

- trop algorithmique (opérationnelle)
- pas assez abstrait (tableaux, pointeurs, etc.)

Slide 10

Partie vraiment difficile des spécifications :

!= mathématiques (ensembles, fonctions)

= modélisation (comprendre le problème et modéliser une solution)

Les notions mathématiques pour les spécifications basées sur l'approche de modélisation abstraite sont relativement simples :

- Logique propositionnelle :

```
 p => q  <=>  ~p | q
~(p & q) <=>  ~p | ~q
```

- Logique des prédicats :

```
 SOME( i: nat :: 2*i = i )
~SOME( i: nat SUCH THAT i ~= 0 :: 2*i = i )
 ALL( i: nat :: i < i+1 )
 ALL( i: nat, j: nat SUCH i < j :: i+1 < j+1 )
```

Slide 11

Un exemple (de M. Jackson) qui montre comment la formalisation à l'aide d'expressions logiques aide à rendre *explicite* certaines propriétés implicites.

Soit les deux affiches suivantes au pied d'un escalier mobile :


```
ALL( p: personne ::
     veut_prendre_escalier(p)
     =>
     SOME( s: souliers :: porte_sur_ses_pieds(p, s) )

ALL( p: personne, c: chien ::
     veut_prendre_escalier_avec(p, c)
     =>
     porte_dans_ses_bras(p, c) )
```

Slide 12

- Ensembles = collections *non-ordonnées* d'éléments

```
e1: set{nat}
e1 = {20, 30}
add(20, e1) = e1
size(e1 U e1) = 2
e1 U {30..32} = {20, 30, 31, 32}
{x: nat SUCH THAT x IN e1 :: x+1} = {21, 31}
```

- Séquences = suites ordonnées d'éléments :

```
s1: sequence{nat}
s1 = [20, 30]
length(s1) = 2
length(s1 || s1) = 4
add(10, s1) = [10, 20, 30]
s1 || [30..32] = [20, 30, 30, 31, 32]
```

Slide 13

- Dictionnaires = associations entre clés et définitions :

```
d1: map{string, nat}
d1 = {["Colin", 16], ["Zoe", 10]; 0}
d1["Zoe"] = 10
d1["Mathieu"] = 0
domain(d1) = {"Colin", "Zoe"}
range(d1) = {0, 10, 16}
```

Slide 14

Spécification par opposition à implémentation

Généralement, une spécification formelle décrit ce qui doit être fait (quoi?) sans dire comment cela sera fait (comment?)

Deux exemples (en Spec) :

- Une fonction (avec plusieurs résultats acceptables) pour calculer la racine carrée d'un nombre réel :

```
r = racine@racine_carree{0.1}(4.0)
=>
1.9748417 < r < 2.0248456
```

- Une fonction pour trier une séquence de nombres :

```
s = [20, 10, 30, 20]
=>
trier@tris{nat}(s) = [10, 20, 20, 30]
```

Slide 15

```
FUNCTION racine_carree
         {precision: real SUCH THAT precision > 0.0}

  MESSAGE racine( x: real )
    WHEN x >= 0.0
      REPLY( r: real )
        WHERE r >= 0.0 & presque_egaux( r^2, x )
    OTHERWISE
      REPLY EXCEPTION racine_imaginaire

  -- Concept auxiliaire.
  CONCEPT presque_egaux( v1 v2: real )
    VALUE( b: boolean )
      WHERE b <=> abs(v1 - v2) <= precision
END
```

Slide 16

```
FUNCTION tris{t: type SUCH THAT Subtype(t, total_order{t})}
  MESSAGE trier( s1: sequence{t} )
    REPLY( s2: sequence{t} )
      WHERE est_permutation(s1, s2),
            en_ordre(s2)

  -- Concepts auxiliaires.
  CONCEPT est_permutation( s1 s2: sequence{t} )
          VALUE( b: boolean )
    WHERE b <=> ALL(x: t :: nb_occs(x, s1) = nb_occs(x, s2)

  CONCEPT nb_occs( x: t, s: sequence{t} ) VALUE( n: nat )
    WHERE n = NUMBER( i: nat SUCH THAT i IN domain(s)
                                     & s[i] = x :: i )

  CONCEPT en_ordre( s: sequence{t} ) VALUE( b: boolean )
    WHERE b <=> ALL( i: nat SUCH THAT 1 <= i < length(s) ::
                     s[i] <= s[i+1] )
END
```

Slide 17

4 Pourquoi y-a-t-il plusieurs notations et méthodes différentes?

Il existe plusieurs styles différents de spécification :

- Modélisation abstraite
- Approche algébrique (types abstraits)
- Algèbre de processus
- Logique temporelle
- . . .

Situation semblable à celle des langages de programmation :

- Domaines d'applications différents
- Styles (paradigmes) d'utilisation différents
- Pouvoirs expressifs différents

Toutes les notations n'ont pas une allure aussi *mathématique* que Z

Slide 18

Exemple : la bibliothèque de H. Saiedian en Spec plutôt qu'en Z

```
MACHINE LibSys{ books: set{Book} }
  STATE( available: set{Book},
         borrowed:  map{Book, User} )
  INVARIANT
    available U domain(borrowed)              = books,
    intersection(available, domain(borrowed)) = {}
  INITIALLY
    available        = books,
    domain(borrowed) = {}

  MESSAGE CheckOut( u: User, b: Book )
    WHEN b IN available
      REPLY OK
      TRANSITION
        available = *available - {b},
        borrowed  = bind(b, u, *borrowed)
    OTHERWISE
      REPLY EXCEPTION Book_not_available
```

Slide 19

```
  MESSAGE Return( u: User, b: Book )
    WHEN borrowed[b] = u
      REPLY OK
      TRANSITION
        available = add(b, *available),
        borrowed  = remove(b, *borrowed)
    OTHERWISE
      REPLY EXCEPTION Invalid_return

  MESSAGE CheckedOutBooks( u: User )
    REPLY( books: set{Book}, count: nat )
     WHERE books = {b: Book SUCH THAT borrowed[b] = u :: b},
           count = size(books)

  CONCEPT Book: type
  CONCEPT User: type
END
```

Slide 20

Exemple : une spécification Spec d'un type abstrait (*collection de valeurs*) `pile`

```
  TYPE pile{T: type}
    MODEL( elems: sequence{T} )
    INVARIANT true

    MESSAGE vide
      REPLY( p: pile{T} ) WHERE p.elems = []

    MESSAGE estVide( p: pile{T} )
      REPLY( b: boolean ) WHERE b <=> p.elems = []

    MESSAGE empiler( p: pile{T}, x: T )
      REPLY( p2: pile{T} ) WHERE p2.elems = add(x, p.elems)

    MESSAGE depiler( p: pile{T} SUCH THAT ~estVide(p) )
      REPLY( p2: pile{T} )
        WHERE p2.elems = p.elems[2..length(p.elems)]

    MESSAGE sommet( p: pile{T} SUCH THAT ~estVide(p) )
      REPLY( x: T ) WHERE x = p.elems[1]
  END
```

Slide 21

Exemple : une spécification algébrique (Larch) d'un type abstrait `Pile`

```
Pile(T, Pile): trait
  introduces
    vide:            -> Pile
    estVide: Pile    -> Bool
    empiler: Pile, T -> Pile
    depiler: Pile    -> Pile
    sommet:  Pile    -> T
  assert
    Pile generated by vide, empiler
    Forall x: T, p: Pile
      estVide(vide)          == true;
      estVide(empiler(p, x)) == false;
      sommet(empiler(p, x))  == x;
      depiler(empiler(p, x)) == p;
```

Slide 22

Exemple : une spécification OCL d'une classe d'objets `Pile`

```
Pile
0 < taille_max
and
elems->size <= taille_max


Pile::Pile()
post:  est_vide(result)

Pile::est_vide(): Boolean
post:  result = elems->notEmpty

Pile::empiler( Object o )
post:  elems = elems@pre->prepend(o)

Pile::depiler()
pre:  elems->notEmpty
post:  elems = elems@pre->subSequence(2, elems@pre->size)

Pile::sommet(): Object
pre:  elems->notEmpty
post:  result = elems->first
```

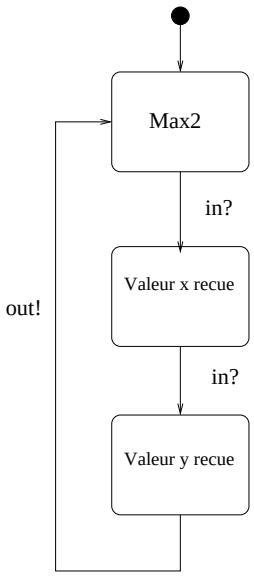
Slide 23

Exemple : un processus réactif LOTOS pour trouver le maximum de deux nombres

```
PROCESS Max2[in, out]: NOEXIT :=
  in ?x: nat;
  in ?y: nat;
  ( [x <= y] -> out !y; Max2[in, out]
    []
    [y <= x] -> out !x; Max2[in, out]
  )
ENDPROC
```

Slide 24

Cette spécification (textuelle) est équivalente au diagramme de transitions (automate à états finis) suivant :



Slide 25

Obligations de preuve

L'application d'une *méthode* formelle (par ex., style VDM) nous oblige aussi à vérifier la qualité *intrinsèque* de cette spécification :

- Vérifier que l'état initial *établit* l'invariant
- Vérifier que chaque opération *préserve* l'invariant
- Vérifier que les réponses indiquées peuvent effectivement être produites

Cette dernière étape nécessite une connaissance du domaine d'application :

```
MESSAGE racine_carre( n: nat ) REPLY( r: nat )
  WHERE r^2 = n
```

Autre étape importante . . . mais *informelle* = valider "cette spécification" auprès des clients/usagers

=> Retraduire en langue naturelle et/ou à l'aide de notations graphiques ce qui a été formalisé

Étapes ultérieures de développement = vérifier que la conception et la mise en oeuvre satisfont cette spécification

Slide 26

5 À quelles étapes les spécifications et les méthodes formelles peuvent-elles être utilisées?

- Analyse et spécification :
  - Description formelle des concepts clés du problème (domaine d'application)
  - Spécification formelle de la solution (comportement du système)
  - Prototypage
- Conception architecturale et détaillée :
  - Spécification formelle de l'interface des modules.
  - Méthode rigoureuse de raffinement
- Codification :
  - Preuves de programmes
  - Synthèse automatique du code
  - Utilisation d'assertions
- Tests
  - Génération automatique des tests

6 Quels sont les principaux bénéfices d'une spécification formelle?

"Devoir en arriver à une meilleure compréhension de l'élément à spécifier en forçant l'analyste à être abstrait tout en étant précis à propos des propriétés du système peut être plus gratifiant que d'avoir le document de spécification lui-même." [Wing90]

Slide 27

- Impact positif de l'*effort de formalisation*.
- Spécifications explicites, précises, non-ambiguës.
- Base pour la mise en oeuvre et pour des vérifications formelles.
- Outils (manipulation, analyse, simulation).
- Base pour le développement des tests.

7 Y-a-t-il des exemples d'applications développées à l'aide de méthodes formelles?

Slide 28

- Réingénierie du système CICS (IBM).
- Composants matériels (Inmos).
- Appareils électroniques (Tektronik).
- Outils pour l'analyse structurée.
- Contrôle du métro (Paris).
- Contrôle de trains (SNCF).
- Contrôle du traffic aérien (UK et USA).
- Contrôle de réacteurs nucléaires (Ontario, USA).
- Navette spatiale de la NASA (contrôleur de vol, GPS).
- Contrôle pour machine de radiothérapie.

Slide 29

8 Conclusion

- Les méthodes formelles peuvent être utiles et peuvent jouer un rôle important dans le développement de systèmes critiques
- Peuvent apparaître coûteuses à première vue (nouvelle approche) mais peuvent avoir pour effet de réduire les coûts des phases subséquentes (mise en oeuvre, tests)
- ... mais ne sont pas la panacée (le *silver bullet*) = il faut savoir les utiliser à bon escient (*lightweight formal methods*)

- Pour en savoir plus :
  - Cours MGL7160 (Hiver 2003).
  - "Modélisation et spécification formelle des logiciels", G. Tremblay, Loze-Dion Éditeurs, 2000.
  - "The Object Constraint Language — Precise Modeling with UML", J. Warmer and A. Kleppe, Addison-Wesley, 1999.