Spécifications formelles

Spécifications formelles : objectifs

- Montrer comment les techniques de spécification formelles aident à découvrir des problèmes dans la conception d'un système.
- Définir et utiliser des techniques algébriques pour spécifier les interfaces d'un système.

Méthodes formelles

- Les spécifications ou méthodes formelles sont des techniques informatiques utilisant des représentations mathématiques du logiciel et du matériel.
- Elles ne sont utilisées que dans des domaines critiques à cause de leur coût (matériel, en temps et humain).
- Le principal bénéfice de leur utilisation est la réduction du nombre d'erreurs dans le logiciel.

Coût de la spécification formelle

- La spécification formelle demande plus d'effort dans les phases avant du projet.
- Elle réduit les erreurs (incomplétude ou inconsistance) de la spécification des charges.
- Ainsi, le nombre de changements du projet à cause d'un problème de spécification de charges est réduit.

Differentes classes de spécifications formelles

- Algébriques : le système est spécifié en termes d'ensembles, d'opérations et de leur relation.
  Ex. séquentiel : Act One, Larch, OBJ.
  Ex. concurrent : Lotos.

- Basés sur les modèles : le système est spécifié en termes de modèle à états et utilisent des opérations qui changent l'état du sytème.
  Ex. séquentiel : Z, VDM, B.
  Ex. concurrent : CSP, réseaux de Pétri, automates hiérarchiques (*statecharts*).

Spécification formelle d'interfaces

- Les grands systèmes sont décomposés en sous-systèmes qui se composent à travers d'interfaces bien définies.
- La spécification des interfaces des sous-systèmes permet leur développement indépendant.
- Les interfaces peuvent être définies comme des types de données abstraits ou des interfaces de classes.
- L'approche algébrique pour la spécification formelle des interfaces permet pour les opérations de l'interface :
  - de les définir formellement,
  - d'analyser formellement (preuve) leur comportement,
  - de dériver l'implémentation d'un objet ou d'un type en partant de sa définition formelle.

Types abstraits de données

- Manipulation de domaines complexes.
- La manipulation est indépendante de la représentation particulière du domaine.
- On se donne :
  - Un nom pour domaine abstrait
  - Des opérations de construction d'une représentation concrète vers le domaine abstrait
  - Des opérations de test pour les différents constructions.
  - Des opérations d'accès d'un objet abstrait vers les composantes d'une représentation concrète.

Exemple : `jour`

Domaine :

`jour` (jours vus comme des entiers de 1 à 31)

Opérations de construction :

`cons_jour` : $\{x : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\} \rightarrow \texttt{jour}$

Opérations de construction étendue :

`entier_to_jour` : $\texttt{entier} \rightarrow \texttt{jour}$

Opérations d'accès :

`numéro_jour` : $\texttt{jour} \rightarrow \{\texttt{x} : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\}$

Exemple : exemplaire

Domaine :

exemplaire

Opérations de construction :

cons_livre : livre $\rightarrow$ exemplaire

cons_dvd : dvd $\rightarrow$ exemplaire

Opérations de test :

est_livre, est_dvd : exemplaire $\rightarrow$ bool

Opérations d'accès :

ex_livre : $\{x : \text{exemplaire} \mid \text{est\_livre}(x)\} \rightarrow$ livre

ex_dvd : $\{x : \text{exemplaire} \mid \text{est\_dvd}(x)\} \rightarrow$ dvd

Exemple : argent

Domaine :

argent

Opérations de construction :

francs : entier $\rightarrow$ argent

euros : entier $\rightarrow$ argent

Opérations de test :

est_francs, est_euros : argent $\rightarrow$ bool

Opérations d'accès :

val_argent : argent $\rightarrow$ entier

Exemple : Les cartes de couleur

Domaine :

couleur

Constantes :

pique, coeur, carreau, trêfle : couleur

Opérations de test :

est_pique : couleur $\rightarrow$ bool

est_coeur : couleur $\rightarrow$ bool

est_carreau : couleur $\rightarrow$ bool

est_trêfle : couleur $\rightarrow$ bool

Domaine :

figure

Constantes :

as, roi, dame, cavalier, valet, dix, neuf, huit, sept :

Opérations de test :

est_as, est_roi, est_dame : figure $\rightarrow$ bool

est_cavalier, est_valet : figure $\rightarrow$ bool

est_dix, est_neuf, est_huit, est_sept : figure $\rightarrow$ bool

Domaine :

carte

Opérations de construction :

cons_carte : couleur $\times$ figure $\rightarrow$ carte

Opérations d'accès :

figure_carte : carte $\rightarrow$ figure

couleur_carte : carte $\rightarrow$ couleur

Domaine :

main

Opérations de construction :

cons_main : $\{\vec{n} \in \text{carte}^5 \mid \text{Diff}(\vec{n})\} \rightarrow$ main

Opérations de construction étendue :

cartes_to_main : $\text{carte}^5 \rightarrow$ main

Opérations d'accès :

carte1 : main $\rightarrow$ carte

$\vdots$

carte5 : main $\rightarrow$ carte

Domaine :

carte_tarot

Constantes :

excuse : carte_tarot

Opérations de construction :

cons_tarot_simple : carte $\rightarrow$ carte_tarot

cons_atout : $\{n$: entier $\mid 1 \leq n \leq 21\} \rightarrow$ carte_tarot

Opérations de construction étendue :

entier_to_carte_tarot : entier $\rightarrow$ carte_tarot

Opérations de test :

est_excuse, est_carte_simple, est_atout : carte_tarot $\rightarrow$

Opérations d'accès : (Première possibilité)

carte_carte_tarot : $\{x$: carte_tarot $\mid$ est_carte_simple(x)

$\rightarrow$ carte

atout_de_tarot : $\{x$: carte_tarot $\mid$ est_atout(x)$\} \rightarrow$

$\{n$: entier $\mid 1 \leq n \leq 21\}$

Opérations d'accès : (Deuxième possibilité)

couleur_de_tarot : $\{x : \texttt{carte\_tarot} \mid \texttt{est\_carte\_simple(x)}\}$
$\rightarrow \texttt{couleur}$

figure_de_tarot : $\{x : \texttt{carte\_tarot} \mid \texttt{est\_carte\_simple(x)}\}$
$\rightarrow \texttt{figure}$

atout_de_tarot : $\{x : \texttt{carte\_tarot} \mid \texttt{est\_atout(x)}\} \rightarrow$
$\{n : \texttt{entier} \mid 1 \leq \texttt{n} \leq 21\}$

Relation entre les deux solutions

couleur_de_tarot(c) peut s'obtenir comme
couleur_carte(carte_carte_tarot(c))

figure_de_tarot(c) peut s'obtenir comme
figure_carte(carte_carte_tarot(c))

Forme générale d'un TDA

Domaine :

dom (éventuellement un produit cartésien)

Constantes :

$c_1, \ldots, c_n : dom$

Opérations de construction :

$cons\_dom_1 : \{x : d_1 \mid P_1(x)\} \rightarrow dom$

$\vdots$

$cons\_dom_m : \{x : d_m \mid P_m(x)\} \rightarrow dom$

d_i est un domaine, $P_i(\_)$ est une propriété

Opérations de construction étendue :

$dom_1\_to\_dom : d_1 \rightarrow dom$

$\vdots$

$dom_m\_to\_dom : d_m \rightarrow dom$

Opérations de test :

$est\_c_1, \ldots, est\_c_n : dom \rightarrow bool$

$\vdots$

$est\_op_1, \ldots, est\_op_m : dom \rightarrow bool$

Opérations d'accès :

$acces_1 \ : \{x : dom \mid est\_op_1(x)\} \to \{x : d_1 \mid P_1(x)\}$

$\vdots$

$acces_m \ : \{x : dom \mid est\_op_m(x)\} \to \{x : d_m \mid P_m(x)\}$

Autres opérations :

$\vdots$

Notation

$\{x : dom \mid \texttt{true}\} \to dom'$ est équivalent à

$\{x : dom\} \to dom'$ est équivalent à

$dom \to dom'$

Quelques variantes

On peut omettre ou rajouter quelques items selon les cas :

- Si la propriété $P_i(x)$ est toujours $true$, alors on l'écrit pas.
- Si le domaine est construit à partir d'une seule constante ou d'une seule opération de construction, alors inutile d'écrire l'opération de test car elle est équivalente à $est(x) = true$.
- Si le domaine est construit à partir de m opérations de construction, alors $m - 1$ opérations de test suffisent.
- Si un domaine d_i est un produit cartésien $a_1 \times \ldots \times a_k$, alors on peut remplacer l'opération d'accès

  $$acces_i \ : \{x : dom \mid est\_op_i(x)\} \to \{x : d_i \mid P_i(x)\}$$

  par k opérations de la forme

$acces_{i_1} \ : \{x : dom \mid \ est\_op_i(x)\} \to$
$\{y : a_1 \mid y = proj_1(x)$ et $x : d_i$ et $P_i(x)\}$

$\vdots$

$acces_{i_k} \ : \{x : dom \mid \ est\_op_i(x)\} \to$
$\{y : a_k \mid y = proj_k(x)$ et $x : d_i$ et $P_i(x)\}$

```
(* type couleur *)

#type couleur = Pi | Co | Ca | Tr;;


(* Constructeurs pour les constantes de couleur *)

#let pique   = Pi;;
#let coeur   = Co;;
#let carreau = Ca;;
#let trefle  = Tr;;
```



```
(* operations de test du type couleur *)

#let est_pique (f) =
     match f with Pi -> true | _ -> false;;

#let est_coeur (f) =
     match f with Co -> true | _ -> false;;

#let est_carreau (f) =
     match f with Ca -> true | _ -> false;;

#let est_trefle (f) =
     match f with Tr -> true | _ -> false;;
```



```
(* type figure *)

#type figure =
 As | Roi | Dame | Cav | Valet | Dix | Neuf | Huit | Sept

(* Constructeurs pour les constantes de figure *)

#let as        =As;;
#let roi       =Roi;;
#let dame      =Dame;;
#let cavalier  =Cav;;
#let valet     =Valet;;
#let dix       =Dix;;
#let neuf      =Neuf;;
```



```
#let huit      =Huit;;
#let sept      =Sept;;


(* operations de test du type figure *)

#let est_as (f) =
     match f with As -> true | _ -> false;;

#let est_roi (f) =
     match f with Roi -> true | _ -> false;;

#let est_dame (f) =
     match f with Dame -> true | _ -> false;;
```

```
#let est_cavalier (f) =
     match f with Cav -> true | _ -> false;;

#let est_valet (f) =
     match f with Valet -> true | _ -> false;;

#let est_dix (f) =
     match f with Dix -> true | _ -> false;;

#let est_neuf (f) =
     match f with Neuf -> true | _ -> false;;

#let est_huit (f) =
```



```
     match f with Huit -> true | _ -> false;;

#let est_sept (f) =
     match f with Sept -> true | _ -> false;;

(* type carte *)

#type carte = C of figure * couleur;;


(* operation de construction du type carte *)

#let cons_carte(f,c) = C(f,c);;
```



```
(* operations d'acc\'es du type carte *)
#let figure_carte(e) = match e with C(f,c)->f;;
#let couleur_carte(e) = match e with C(f,c)->c;;

(* type carte_tarot *)

#type carte_tarot =
   Ex | S of carte | A of int;;

(* operations de construction etendue du type carte_taro

#let excuse               = Ex;;
#let cons_tarot_simple(c) = S(c);;
```



```
#let cons_atout (n)       = A(n);;

(* operations de construction du type carte_tarot *)

#let int_to_carte_tarot (n) =
     if 1 <= n & n <= 21
        then cons_atout (n)
        else failwith"numero d'atout invalide";;


(* operations de test du type carte_tarot *)

#let est_excuse (c) =
     match c with Ex -> true | _ -> false;;
```

```
#let est_carte_simple (c) =
     match c with S(_) -> true | _ -> false;;

#let est_atout (c)        =
     match c with A(_)  -> true | _ -> false;;



(* operations d'acces du  type carte_tarot *)

#let couleur_de_tarot (e) =
     match e with
     S(C(_,c)) -> c
```



```
     | _       -> failwith "Pas une carte simple";;


#let figure_de_tarot (e) =
     match e with
     S(C(f,_)) -> f
     | _       -> failwith "Pas une carte simple";;


#let atout_de_tarot (e) =
     match e with
     A(x) -> x
     | _  -> failwith "Pas un atout";;
```


Les profils de symboles de la signature

```
pique:                          -> couleur
coeur:                          -> couleur
carreau:                        -> couleur
trefle:                         -> couleur
est_pique:       couleur        -> bool
est_coeur:       couleur        -> bool
est_carreau:     couleur        -> bool
est_trefle:      couleur        -> bool
as:                             -> figure
roi:                            -> figure
dame:                           -> figure
cavalier:                       -> figure
```



```
valet:                          -> figure
dix:                            -> figure
neuf:                           -> figure
huit:                           -> figure
sept:                           -> figure
est_as:          figure         -> bool
est_roi:         figure         -> bool
est_dame         figure         -> bool
est_cavalier:    figure         -> bool
est_valet:       figure         -> bool
est_dix:         figure         -> bool
est_neuf:        figure         -> bool
est_huit:        figure         -> bool
est_sept:        figure         -> bool
```

```
cons_carte:         figure * couleur -> carte = <fun>
figure_carte:       carte            -> figure = <fun>
couleur_carte:      carte            -> couleur = <fun>
excuse:                              -> carte_tarot
cons_tarot_simple: carte             -> carte_tarot = <fu
cons_atout:         int              -> carte_tarot = <fu
int_to_carte_tarot:int               -> carte_tarot = <fu
est_carte_simple:   carte_tarot      -> bool = <fun>
est_atout:          carte_tarot      -> bool = <fun>
figure_de_tarot:    carte_tarot      -> figure = <fun>
couleur_de_tarot:   carte_tarot      -> couleur = <fun>
atout_de_tarot:     carte_tarot      -> int = <fun>
```


Codage d'autres opérations d'un TDA

- Abstrait : on utilise uniquement les opérations abstraites définies dans le TDA.
- Concret : on utilise les primitives du langage de programmation associées au domaines concrets utilisés dans le TDA.

Codage abstrait

```
#let points (c) =
    if est_excuse(c)
    then 4.5
    else if est_carte_simple (c)
         then let f = figure_de_tarot(c) in
              if est_roi(f)
              then 4.5
              else if est_dame(f)
                   then 3.5
                   else if est_cavalier(f)
                        then 2.5
                        else if est_valet(f)
```



```
                                then 1.5
                                else 0.5
          else let n = atout_de_tarot(c) in
               if n=1 or n=21
                  then 4.5
                  else 0.5;;

val points : carte_tarot -> float = <fun>
```

Codage concret

```
#let points(c) = match c with
    Ex             -> 4.5
  | S(C(Roi,_))    -> 4.5
  | S(C(Dame,_))   -> 3.5
  | S(C(Cav,_))    -> 2.5
  | S(C(Valet,_)) -> 1.5
  | S(C(_,_))      -> 0.5
  | A(1)           -> 4.5
  | A(21)          -> 4.5
  | _              -> 0.5;;
val points : carte_tarot -> float = <fun>
```


Exemple : un seul TDA et deux implémentations possibles

Domaine :

pile

Constantes :

pile_vide : pile

Opération de construction :

cons_pile : entier $\times$ pile $\rightarrow$ pile

Opération de test :

est_pile_nv : pile $\rightarrow$ bool

Opérations d'accès

premier_pile : $\{x : \text{pile} \mid \text{est\_pile\_nv}(x)\} \rightarrow$ entier

reste_pile : $\{x : \text{pile} \mid \text{est\_pile\_nv}(x)\} \rightarrow$ pile

Exemple : une première implémentation

```
#type pile = intlist;;

#let pile_vide = [];;

#let cons_pile(a,p) = a::p;;

#let est_pile_nv(p) =
     match p with a::l -> true | _ -> false;;

#let premier_pile(p) = match p with
     a::l -> a
    | _   -> failwith"Pile Vide";;
```

```
#let reste_pile(p) = match p with
     a::l -> l
    | _   -> failwith"Pile Vide";;
```


Les profils de symboles de la signature

```
pile_vide:                    -> intlist
cons_pile :    int * intlist -> intlist
est_pile_nv :  intlist       -> bool
premier_pile : intlist       -> int
reste_pile :   intlist       -> intlist
```


Exemple : une autre implémentation

```
#type pile = Pile_Vide | P of int * pile;;

# let pile_vide = Pile_Vide;;

# let cons_pile(t,r) = P(t,r);;

# let est_pile_nv(p) = match   p with
     Pile_Vide -> false
    | _        -> true ;;

# let premier_pile(p) = match p with
      P(t,r) -> t
```



```
    | _      -> failwith "Pas une pile non vide";;

# let reste_pile(p) = match p with
      P(t,r) -> r
    | _      -> failwith "Pas une pile non vide";;
```

Les profils de symboles de la signature

```
pile_vide:                -> pile
cons_pile :    int * pile -> pile
est_pile_nv :  pile       -> bool
premier_pile : pile       -> int
reste_pile :   pile       -> pile
```


Exemple : encore une autre implémentation

```
#type pile_nv = {tete:int; reste:ma_pile} and
      ma_pile = Pile_Vide | P of pile_nv;;

type pile_nv = { tete : int; reste : ma_pile; } and
     ma_pile = Pile_Vide | P of pile_nv


# let cons_pile_nv(t,r) = { tete=t ; reste = r};;
val cons_pile_nv : int * ma_pile -> pile_nv = <fun>

# let tete_pile_nv(p) = p.tete;;
val tete_pile_nv : pile_nv -> int = <fun>
```



```
# let reste_pile_nv(p) = p.reste;;
val reste_pile_nv : pile_nv -> ma_pile = <fun>

# let pile_vide =Pile_Vide;;
val pile_vide : ma_pile = Pile_Vide

# let cons_pile(a,p) = P(cons_pile_nv(a,p));;
val cons_pile : int * ma_pile -> ma_pile = <fun>

# let est_pile_nv(p) =
           match   p with Pile_Vide -> false | _ -> true
val est_pile_nv : ma_pile -> bool = <fun>
```



```
# let premier_pile(p) = match p with
     Pile_Vide -> failwith"Pile Vide"
   | P(x)      -> x.tete;;
val premier_pile : ma_pile -> int = <fun>

# let reste_pile(p) = match p with
   Pile_Vide -> failwith"Pile Vide"
 | P(x) -> x.reste;;
val reste_pile : ma_pile -> ma_pile = <fun>
```