UPMC Sorbonne Universités – Master Informatique – STL

Cours Composant
2. Langage de spécification

©Frédéric Peschanski

Sorbonne Universités – UPMC – LIP6

1er février 2014

Plan du cours

- Spécifications algébriques
  - Introduction
  - Principes
  - Complétude et cohérence
  - Interactions

Spécifications formelles

Besoins

Langage de spécification permettant de décrire ce que doit faire un composant logiciel :

- de façon précise et non-ambigüe ($\Rightarrow$ formalisation)
- indépendemment des implémentations
- de façon **cohérente** et **vérifiable**
- sans surspécifier (notion de **complétude**)

Approche formelle « légère »

- utilisation de spécifications algébriques, lointain héritier des types de données abstraits (ADT)
- \+ relativement simples d'utilisation, base des **contrats**
- \- pas d'ordre supérieur

Principes

Fondations

- Ensembles et fonctions du premier ordre
- Logique typée du premier ordre
- Notion d'**observation** (vision algébrique)

Spécification

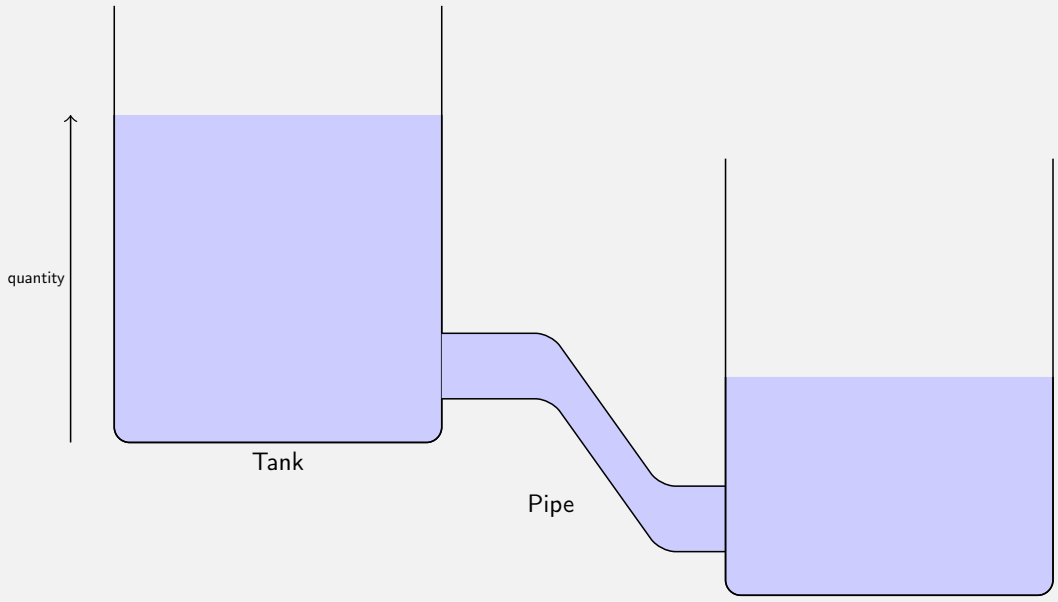
- Brique élémentaire : le **Service** devant être proposé par le fournisseur à ses clients
- indépendant des implémentations
  - plusieurs implémentations du même service
  - une meme implémentation peut requérir/fournir plusieurs services
- Cahier des charges minimal (et formel) pour les implémentations

Format des spécifications

Service : *nom du service spécifié*
Types : *dépendances des types élémentaires*
Require : *dépendances de services*
Observators :
fonctions d'observation de l'état
...
Constructors :
fonctions de construction
...
Operators :
fonctions de modification
...
Observations :
Axiomes d'observation
...

Types élémentaires : **boolean**, **int**, **double**, **String**, etc.

Exemple : les « cuves »



Service « cuve »

Questions

- Quel est le nom du service ?
- Que veut-on observer sur chaque fournisseur du service ?

Service : Tank
Types : **boolean**,**double**
Observators :
quantity : [Tank] $\rightarrow$ **double**
empty : [Tank] $\rightarrow$ **boolean**

Signature des observateurs

nomObs : [Service] $\times T_1 \times \ldots \times T_n \rightarrow T$

Remarque : [Service] signifie « ce Service » (*this*)
$\Rightarrow$ état courant

Constructeurs

Question

- Comment construire une Cuve ?

Service : Tank
Types : **boolean**,**double**
Observators :
quantity : [Tank] $\rightarrow$ **double**
empty : [Tank] $\rightarrow$ **boolean**
Constructors :
init : $\rightarrow$ [Tank]

Signature des constructeurs

init : $T_1 \times \ldots \times T_n \rightarrow$ [Service]

Operateurs

Question

- Comment manipuler une Cuve ?

Service : Tank
Types : **boolean**,**double**
Observators :
quantity : [Tank] $\rightarrow$ **double**
empty : [Tank] $\rightarrow$ **boolean**
Constructors :
init : $\rightarrow$ [Tank]
Operators :
fill : [Tank] $\times$ **double** $\rightarrow$ [Tank]
pump : [Tank] $\times$ **double** $\rightarrow$ [Tank]

Signature des opérateurs

nomOp : [Service] $\times T_1 \times \ldots \times T_n \rightarrow$ [Service]

Fonctions partielles

Questions

- Quels sont les domaines de définition des observateurs, constructeurs et opérateurs ?
- Quelle est l'approche de spécification ?
  - favoriser le client : approche **permissive**
  - favoriser le fournisseur : approche **défensive**

Exemple approche **défensive**

Service : Tank
Types : **boolean**,**double**
Observators :
quantity : [Tank] $\rightarrow$ **double**
empty : [Tank] $\rightarrow$ **boolean**
Constructors :
init : $\rightarrow$ [Tank]
Operators :
fill : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : fill(B,q) **require** $q \geq 0$
pump : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : pump(B,q) **require** $q \geq 0 \wedge \text{quantity}(B) - q \geq 0$

Fonctions partielles

Questions

- Quels sont les domaines de définition des observateurs, constructeurs et opérateurs ?
- Quelle est l'approche de spécification ?
  - favoriser le client : approche **permissive**
  - favoriser le fournisseur : approche **défensive**

Exemple approche **permissive**

Service : Tank
Types : **boolean**,**double**
Observators :
quantity : [Tank] $\rightarrow$ **double**
empty : [Tank] $\rightarrow$ **boolean**
Constructors :
init : $\rightarrow$ [Tank]
Operators :
fill : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : fill(B,q) **require** $q \geq 0$
pump : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : pump(B,q) **require** $q \geq 0$

Résumé : fonctions

Signature des fonctions

nomFunct : $T_1 \times \ldots \times T_n \rightarrow T$
precondition nomFunct$(x_1, \ldots, x_n)$ **require** $\text{pre}_{\text{nomFunct}}(x_1, \ldots, x_n)$

où $\text{pre}_{\text{nomFunct}}(x_1, \ldots, x_n) \in \{\text{true}, \text{false}\}$

$\Rightarrow$ formule logique du premier ordre avec $x_1, \ldots, x_n$ des variables quantifiées universellement sur leur domaine $T_1, \ldots, T_n$

Remarque par défault, $\text{pre}_{\text{nomFunct}}(x_1, \ldots, x_n) \Leftrightarrow$ **true**

Observations : définitions

Définitions préliminaires

Expression bien formée composition de fonctions respectant les types de domaines de définitions
(ex. : $pump(fill(B, q_1), q_2)$)

Expression d'observation expression bien formée dont la fonction de niveau principal est un observateur
(ex. : $quantity(pump(fill(B, q_1), q_2)))$

Formule bien formée expression bien formée à valeur booléenne
(ex. : $quantity(pump(fill(B, q_1), q_2)) > 0$)

Observation égalité de type $O = E$ où O est une expression d'observation de type T et E une expression à valeur dans T
(ex. :
$quantity(pump(fill(B, q_1), q_2)) = quantity(B) + q_1 - q_2)$)

Observations : objectifs

Objectifs

Spécifier, en tant qu'**axiomes**, les **observations minimales** permettant de caractériser de façon cohérente (et optionnellement complète) la sémantique du service spécifié.

Important les observations sont des expressions **axiomatiques**, on ne doit pas pouvoir les déduire les unes des autres.

Approche méthodologique

1. Minimiser les observateurs $\Rightarrow$ **invariants**
2. Observer, avec les observateurs non minimisés et les constantes (cf. plus loin), les différentes possibilités de construction
3. Croiser chaque observateur non minimisé avec chaque opérateur

Phase 1 : minimisation des observateurs et invariants utiles

Objectif prioritaire Minimiser les observateurs
$\Rightarrow$ **Syntaxe** : obs(...) $\overset{\text{min}}{=}$ expr

Objectif secondaire Relier les observations entre elle lorsque cela fait sens.
$\Rightarrow$ **invariants utiles**

Exemple retour aux cuves

Service : Tank
etc.
Observations :
[invariants] *// catégorie des observations*
empty(B) $\overset{\text{min}}{=}$ (quantity(B) $= 0$) *// minimisation*
quantity(B) ≥ 0 *// invariant utile*

Remarque pas d'observation spécifique pour "empty" dans la suite

Phase 2 : observer les constructeurs

Objectif décrire les observations minimales sur les constructeurs

Service : Tank
etc.
Observations :
[invariants]
empty(B) $\overset{\text{min}}{=}$ (quantity(B) $= 0$)
quantity(B) ≥ 0
[init]
quantity(init(B)) $= 0$

Remarque chaque expression respecte implicitement les domaines de définitions (sinon c'est une erreur de spécification).

Phase 3 : observer les opérateurs

Objectif décrire les observations minimales sur les opérateurs

Service : Tank
etc.
Observations :
[invariants]
empty(B) $\overset{\text{min}}{=}$ (quantity(B) $= 0$)
quantity(B) ≥ 0
[init]
quantity(init(B)) $= 0$
[fill]
quantity(fill(B,q)) $=$ quantity(B) $+ q$
[pump]
quantity(pump(B,q)) $=$ quantity(B) $- q$

Remarque approche **défensive** (préconditions plus fortes, observations/postconditions plus faibles)

Phase 3 : observer les opérateurs

Objectif décrire les observations minimales sur les opérateurs

Service : Tank
etc.
Observations :
[invariants]
empty(B) $\overset{\text{min}}{=}$ (quantity(B) $= 0$)
quantity(B) ≥ 0
[init]
quantity(init(B)) $= 0$
[fill]
quantity(fill(B,q)) $=$ quantity(B) $+ q$
[pump]
quantity(pump(B,q)) $=$ max(0,quantity(B) $- q$)

Remarque approche **permissive** (préconditions plus faibles, observations/postconditions plus fortes)

Phase 3 : complétude des observations

IMPORTANT : une observation **complète** spécifie à la fois :

- ce qui change entre l'état "avant" et l'état "après" l'opération
- ce qui ne change pas également.

Exemple :

...
skip : [Tank] → [Tank]
...
Observations
...
[skip]
quantity(skip(B)) = quantity(B)

⇒ ici il est important de savoir qu'aucun changement n'est effectué !

Cohérence et complétude

Analyse d'un service

Cohérence On peut déduire *au plus* une valeur de chaque observation

Complétude On peut déduire *au moins* une valeur de chaque observation

Objectifs

La **cohérence** est primordiale. Toute incohérence est un *bug* de spécification.
La **complétude** est importante mais est parfois difficile, voir impossible, à obtenir. En pratique, on acceptera donc si c'est justifié de « sous-spécifier ». En revanche, on essaiera dans la mesure du possible de ne pas « sur-spécifier » (élimination des redondances).

Remarque en CPS on donnera des arguments essentiellement informels concernant la cohérence et la complétude.

Cohérence et complétude : exemple

Question

Le service Tank est-il cohérent et complet ?

Cohérent : oui il n'existe pas d'état B avec par exemple quantity(B)=q_1 et quantity(B)=q_2 avec $q_1 \neq q_2$

Complétude : oui car dans tout état B quantity(B) possède une valeur.

$\Rightarrow$ preuves formelles non-triviales (surtout la complétude)

Exercice : modifier le service Tank pour introduire une incohérence, une incomplétude.

incohérence deux quantités manipulées différemment mais avec un invariant $\text{quantity}_1(B) = \text{quantity}_2(B)$.

incomplétude une opération "fault" qui ne permet plus de faire d'observation.

Activabilité et Convergence

Activabilité au moins une précondition d'opération est satisfaite dans chaque état possible (rq. : possible $\neq$ atteignable)

Convergence une opération convergente ne peut être activée indéfiniment à partir d'un état possible

Objectifs

L'**activabilité** d'un service est une condidition suffisante pour l'**absence de blocage**, c'est une garantie optionelle mais souvent souhaitable.
La **convergence** de toutes les opérations d'un service est une condition nécessaire pour la **vivacité** de ce dernier. Il s'agit d'une question toujours intéressante en pratique et souvent non-triviale. Une opération qui ne converge pas est dite **divergente**. Il existe bien sûr des cas où on ne sait pas si une opération converge ou non.

Remarque en CPS l'activabilité et la convergence seront discutée de façon semi-formelle.

Activabilité : règle générale

Soit un service S avec des opération $\text{op}_1, \ldots, \text{op}_n$ et les préconditions respectives $\text{pre}_{\text{op}_1}, \ldots, \text{pre}_{\text{op}_n}$.

Question : Le service S est-il activable ?

Formellement : S est activable *ssi* $\text{pre}_{\text{op}_1}(S) \vee \ldots \vee \text{pre}_{\text{op}_n}(S) \Longleftrightarrow \text{true}$

Remarque : on ne s'intéresse dans les préconditions qu'aux contraintes qui ne concernent pas les paramètres de l'opérations concernée (donc on suppose que l'opération est utilisée de façon correcte). On peut aussi supposer les invariants (qui sont *aussi* des préconditions mais vraies dans les états possibles).

Exemple : dans le service Tank l'activabilité est ainsi triviale car les préconditions ne dépendent que des arguments.

Activabilité : exemple

Service : Switch
Types : **boolean**
Observators :
  state : [Switch] $\rightarrow$ **boolean**
Constructors :
  init : $\rightarrow$ [Switch]
Operators :
  on : [Switch] $\rightarrow$ [Switch]
    precondition : on(S) **require** state(S) = **false**
  off : [Switch] $\rightarrow$ [Switch]
    precondition : off(S) **require** state(S) = **true**
  ...

$\Longrightarrow$ le service Switch est activable car :

[$\forall S$:Switch, state(S) = **false** $\vee$ state(S) = **true**] $\Longleftrightarrow$ **true**

Convergence : règle générale

Syntaxe d'une opération op **convergente** dans un service Serv :

Service : Serv
. . .
Operators :
. . .
op : [Serv] $\times T_1 \times \ldots \times T_n \rightarrow U$
precondition op(S,v_1,. . .,v_n) **require** $\text{pre}_{\text{op}}(S, v_1, \ldots, v_n)$
converge $\text{variant}_{\text{op}}(S)$

avec : $\text{variant}_{\text{op}}$ fonction à valeur dans un ensemble ordonné $(E, <)$ et bien fondé (en pratique, on prend souvent $(\mathbb{N}, <)$).

Critère de convergence

$\text{variant}_{\text{op}}(\text{op}(S, v_1, \ldots, v_n)) < \text{variant}_{\text{op}}(S)$

Convergence : exemple

Convergence des opérations du service Tank :

Service : Tank
. . .
Operators :
fill : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : fill(B,q) **require** $q \geq 0$
// ne converge pas

pump : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : pump(B,q) **require** $q \geq 0$
converge $\text{variant}_{\text{pump}}(B) \stackrel{\text{def}}{=} \text{quantity(B)}$

Critère de convergence

$\text{variant}_{\text{pump}}(\text{pump}(B,\text{q})) < \text{variant}_{\text{pump}}(B)$
$\Rightarrow$ Problème !

Convergence : exemple (oops !)

Convergence des opérations du service Tank :

Service : Tank
. . .
Operators :
fill : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : fill(B,q) **require** $q \geq 0$
// *ne converge pas*

pump : [Tank] $\times$ **double** $\rightarrow$ [Tank]
precondition : pump(B,q) **require** $q>0 \wedge$ quantity(B) $>= q$
converge $\text{variant}_{\text{pump}}(B) \stackrel{\text{def}}{=} \text{quantity(B)}$

Critère de convergence

$\text{variant}_{\text{pump}}(\text{fill}(B,q)) < \text{variant}_{\text{pump}}(B)$
$\text{quantity}(\text{pump}(B,q)) = \text{quantity}(B) - q < \text{quantity}(B)$ [CQFD]

Interactions

Les spécifications de service permettent de décrire l'**interface interne** des composants (fournisseurs/implémenteurs du service).
Il nous manque la description de l'**interface externe** ou contexte d'utilisation du service.

Besoins

Certains services ont besoins d'autres services pour être spécifiés.

Exemple le tuyau (*Pipe*) reliant deux cuves.
$\Rightarrow$ Dans ce cas, on ajoute la section **Use** aux spécifications et on observe explicitement les services requis.

Service : *nom du service*
etc.
Use : *services externes*
etc.

Exemple : service « tuyau »

Service : Pipe
Use : Tank
Types : **boolean**, **double**
Observators :
- quantity : [Pipe] → **double**
- **const** capacity : [Pipe] → **double**
- **const** inTank : [Pipe] → Tank
- **const** outTank : [Pipe] → Tank
- openIn : [Pipe] → **boolean**
- openOut : [Pipe] → **boolean**

Constructors :
- init : Tank × Tank × **double** → [Pipe]
  - **precondition** init(I,O,c) **require** $c > 0$

Operators :
- switchIn : [Pipe] → [Pipe]
  - **precondition** switchIn(P) **require** ¬openOut(P)
- switchOut : [Pipe] → [Pipe]
  - **precondition** switchOut(P) **require** ¬openIn(P)
- flush : [Pipe] → [Pipe]
  - **precondition** flush(P) **require** ¬openIn(P) ∧ ¬openOut(P)

Interlude : observateurs constants

Un **observateur constant** est décoré par le mot clé **const**

- Pour un observateur C constant, il est uniquement nécessaire de décrire les observations de C sur les constructeurs.
- Pour un service s et un opérateur op qui n'est pas un constructeur, on aura implicitement : $C(op(s,\ldots)) = C(s,\ldots)$

Par exemple, si on retire le **const** de l'observateur capacity de capacité de tuyau :

[init]
- capacity(init(I,O,c)) = c
- ...

[switchIn]
- capacity(switchIn(P)) = capacity(P)
- ...

[switchOut]
- capacity(switchOut(P)) = capacity(P)
- ...

[flush]
- capacity(flush(P)) = capacity(P)

Observations

Service : Pipe
etc.
Observations :
[invariants]
$0 \leq \text{quantity}(P) \leq \text{capacity}(P)$
$\text{openIn}(P)$ **xor** $\text{openOut}(P)$
[init]
$\text{quantity}(\text{init}(I,O,c)) = 0$
$\text{capacity}(\text{init}(I,O,c)) = c$
$\text{inTank}(\text{init}(I,O,c)) = I$
$\text{outTank}(\text{init}(I,O,c)) = O$
$\text{openIn}(\text{init}(I,O,c)) =$ **false**
$\text{openOut}(\text{init}(I,O,c)) =$ **false**
[switchIn]
$\text{quantity}(\text{switchIn}(P)) = ?$
etc. ?

$\Rightarrow$ Comment exprimer les interactions entre les cuves et le tuyau ?

Observations composées

Important

observation des interactions entre services

On sépare clairement :

la cause qui est l'opérateur (ou le constructeur) du service courant que l'on souhaite spécifier

La conséquence interne ou modification (changement d'état observable) du service spécifié

La conséquence externe qui correspond à la modification d'un service externe

Par exemple :

cause ouverture de la porte d'entrée du tuyau (opérateur switchIn)

conséquence interne remplissage du tuyau

conséquence externe la cuve d'entrée se vide au moins partiellement

Observation externe

Lorsqu'un service utilise des services externes :

Service : Serv
Use : S_1, S_2, ... // *services externes*
etc.

On peut manipuler des informations des types externes S_1, S_2,...

- utiliser un observateur obs du service externe S par S::obs
  ex. : Tank::quantity(...)
- décrire une **modification externe** par utilisation d'une opération op de S, dénotée S::op(...)

ATTENTION : on ne peut pas décrire une modification externe sans utiliser une opération, afin de préserver la **séparation des préoccupations**

Exemple : opérateur externe

cause ouverture de la porte d'entrée du tuyau (opérateur switchIn)

conséquence interne remplissage du tuyau d'un volume v

conséquence externe la cuve d'entrée se vide du volume v

On pose $avail(P) \stackrel{\text{def}}{=} capacity(P) - quantity(P)$ la capacité de remplissage actuelle du tuyau.
et $dev(P) \stackrel{\text{def}}{=} min(Tank.capacity(inTank(P)), avail(P))$ le volume maximum déversable de la cuve vers le tuyau.

...
[switchIn]
quantity(switchIn(P)) = quantity(P) + dev(P)
inTank(switchIn(P))) = Tank::pump(inTank(P),dev(P))
outTank(switchIn(P)) = outTank(P)

Remarque : il faudrait faire de "avail()" et "dev()" des observateurs.

Fin