MGL7160 Méthodes formelles et semi-formelles

Révision

Roger Villemaire

Département d'informatique
UQAM

15 avril 2014

Unified Modeling Language

- Nous allons nous concentrer sur les diagrammes de classes UML qui permettent de :
  - décrire des classes avec leurs attributs (variables membres, champs) et opérations (méthodes).

Associations

- La représentation des associations met l'emphase sur les interactions (usages) des classes.
  - Une extrémité est déterminée par :
    - sa *multiplicité* (le nombre d'éléments associés),
    - son statut d'*unicité* (pas du doublons),
    - le fait d'être *ordonné* ou pas.
  - En pratique une extrémité d'une association est implémentée par un attribut de la classe *opposée*.
  - Plus précisément cet attribut est en général une collection d'éléments (en fonction de la multiplicité). Le type de collection est déterminé par l'unicité et le fait d'être ordonné ou non.

Object Constraint Language

- Une norme de l'OMG (Object Management Group),
- En fait une partie de UML (Unified Modeling Language),
- Permet de décrire des *contraintes* sur les attributs de classes de diagrammes UML.

Quelques opérations

- Boolean : and, or, xor, not, implies, if-then-else-endif
  - true implies false = false,
  - if false then A else B = B
- Integer : *, +, -, /, div(), abs()
  - 5/2 = 2.5, 5.div(2) = 2
  - (-5).abs() = 5
- Real : *, +, -, /, floor()
  - 1.1 * 5.0 = 5.5
  - (5.2).floor() = 5.0
- String : concat(), size(), substring()
  - 'abc'.concat('def') = 'abcdef'
  - 'abc'.size() = 3
  - 'abcdef'.substring(1,3) = 'abc'

Les collections

Collection	Ordre	Unicité
Set	non	oui
Bag	non	non
Sequence	oui	non
OrderedSet	oui	oui

- Set{0,1,2}
- Bag{0,0,1,1,1,2}
- Sequence{2,0,1,1,0,2}
- OrderedSet{2,0,1,3}

Opérations sur les collections

- Ces opérations sont préfixées par ->
  - Set{0,1,2} -> size() = 3
  - Bag{0,0,1,1,1,2} -> isEmpty() = false
  - Sequence{2,0,1,1,0,2} -> notEmpty() = true
  - Bag{0,0,1,1,1,2} <> Bag{0,1,2} = true
  - Sequence{2,0,1,1,0,2} = Sequence{} = false

Inclusion et exclusion

- Set{0,1,2} -> includes(1) = true
- Bag{0,0,1,1,1,2} -> excludes(2) = false
- Sequence{2,0,1,1,0,2} -> includesAll(Set{2,0}) = true
- OrderedSet{2,0,1,3} -> excludesAll(Sequence{5,0}) = false

Quelques Itérateurs

- Bag{0,0,1,1,1,2} -> select(c : c > 0) = Bag{1,1,1,2}
- Sequence{2,0,1,1,0,2} -> reject(c : c > 1) = Sequence{0,1,1,0}
- Bag{1,2,0,1,2,3} -> collect(c : c.mod(2)) = Bag{1,0,0,1,0,1}
- Sequence{1,2,0,1,2,3} -> collect(c : c.mod(2)) = Sequence{1,0,0,1,0,1}
- Set{2,0,1,3} -> collect(c : c.mod(2)) = Bag{0,0,1,1}
- OrderedSet{2,0,1,3} -> collect(c : c.mod(2)) = Sequence{0,0,1,1}

Invariants pré- post-conditions

- En pratique les expressions OCL servent principalement à spécifier des invariants, des pré- et des post-conditions.
  - invariant : propriété des instances d'une classe qui devrait *en principe* être toujours vérifiée.
  - pré-condition : propriété qui devrait toujours être vérifiée *avant* l'appel d'une opération.
  - post-condition : propriété qui devrait toujours être vérifiée *après* l'appel d'une opération.

Invariants

- Un invariant
  - est une propriété satisfaite par les instances *correctement construites* d'un classe.
  - décrit des contraintes sur les attributs de la classe qui doivent être maintenues.
  - permet de préciser le sens de la classe.

Opérations logiques

- Set{0,1,2,3} -> exists(e | 2∗e = 4)= true.
- Set{0,1,2,3} -> exists(e1,e2 | e1 + e2 = 4)= true.
- Set{0,1,2,3} -> forAll(e | 2∗e = 4)= false.
- Set{0,1,2,3} -> one(e | 2∗e = 4)= true.
  - Un seul élément de la collection satisfait la condition.
- Set{0,1,2,3} -> isUnique(e | 2∗e)= true.
  - Chaque élément donne une valeur différente.
- Set{0,1,2,3} -> any(e | e > 1)= 3.
  - Retourne un élément quelconque de la collection qui satisfait la condition.

Opérations ensemblistes

- Set{0,1,2} -> union(Set{3,4})= Set{0,1,2,3,4}
  - l'union d'un Set et d'un Bag donne un Bag, l'union de deux Sequences les met bout-à-bout.
- Set{0,1,2} -> intersection(Set{0,3,4})= Set{0}
  - l'intersection d'un Set et d'un Bag donne un Set, celle de deux Bag un Bag.
- Différence d'ensembles : Set{0,1,2} – Set{0,3,4})= Set{1,3}

Collections ordonnées

- first() retourne le premier élément.
- last() retourne le dernier élément.
- append(objet) ajoute un élément à la fin.
- prepend(objet) ajoute un élément au début.
- insertAt(index, objet) insère objet à la position index (décale les autres éléments).
- at(i) retourne l'élément à la position i.
- indexOf(objet) retourne la position d'objet.
- reverse() retourne la collection renversée.
- subOrderedSet et subSequence(début,fin) donnent les sous-ensemble ordonné/séquence.

Conversion de types

- asSet() retourne un ensemble (enlève les dupliqués)
- asOrderedSet() (enlève les dupliqués et invente l'ordre si nécessaire).
- asSequence() (invente l'ordre si nécessaire !).
- sortedBy retourne la collection triée selon la valeur d'une expression (comparable par rapport à <)
    - Set donne un OrderedSet, Bag une Sequence

pré- post-conditions

- Une opération est définie par son type de retour ainsi que les types de ses arguments. OCL permet d'associer à une opération UML :
  - des pré-conditions : propriétés qui devraient toujours être vérifiées *avant* l'appel de l'opération.
  - des post-conditions : propriétés qui devraient toujours être vérifiées *après* l'appel de l'opération.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
  - `result`, la valeur de retour de l'opération,
  - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
    - `a@pre`, l'ancienne valeur de `a`.
    - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
    - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
    - `a@pre.b@pre`, l'ancien b de l'ancien `a`.
    - `ref.oclIsNew()`, permet de savoir si la référence a été créée.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
  - une obligation pour le client qui veut utiliser l'opération.
  - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
  - une obligation pour le fournisseur qui doit l'assurer.
  - une garantie pour le client qui utilise l'opération.

MDE

- L'ingénierie dirigée par les modèles (Model Driven Engineering, MDE) est une approche qui vise :
  - le développement de systèmes informatiques par
    - l'usage de modèles (souvent UML) pour l'analyse,
    - la conception par transformation et l'intégration de modèles,
    - si possible de façon automatique ou à l'aide d'outils.
  - en vue principalement de réduire le coût et le temps de développement.

Approche MDA de l'OMG

- un modèle de référence indépendant de la plate-forme (platform-independent model (PIM))
  - accent sur les fonctionnalités et le comportement métiers indépendamment des technologies utilisées pour la réalisation.
- des modèles spécifiques (platform-specific models (PSM))
  - décrivent comment le PIM est implémenté dans une certaine technologie (middleware platform : Web Services,.NET,EJB etc.)
- les implémentations complètes (une par plate-forme).
- Les étapes de transformation sont réalisées à l'aide d'outils automatiques et semi-automatiques.

Chaîne de transformations

- Création d'un modèle UML à l'aide de Papyrus,
  - produit le fichier XML contenant le modèle UML.
- Ajout de propriétés OCL pour préciser le sens du modèle UML à l'aide de Dresden OCL.
  - lit le fichier XML contenant le modèle UML
  - produit le fichier OCL.
- Production d'un squelette de code Java à l'aide de Acceleo.
  - lit le fichier XML contenant le modèle UML,
  - produit les déclarations Java correspondantes.
- Instrumentalisation du code Java par Dresden OCL pour la réalisation des tests.
  - lit le modèle UML ainsi que les propriétés OCL,
  - produit des fichiers AspectJ pour instrumenter le code Java.

Vérification exhaustive

- Pour vérifier simultanément tous les comportements d'un logiciel, il faut :
  - s'assurer de couvrir tous les cas possibles.
- Ceci peut se faire à l'aide *d'exécutions symboliques*.
  - Mais il faut alors disposer d'une *formalisation* de l'exécution du système.

Système de déduction

- Un système de déduction est un ensemble de *règles* qui permettent de transformer un *énoncé* pour établir sa véracité.
- Un système de déduction complet possède suffisamment de règles pour pouvoir établir tous les énoncés qui sont vrais.
- Il existe plusieurs systèmes de déduction formalisant tous les concepts de l'informatique.

Démonstrateur de théorème

- Un *démonstrateur de théorème (Automatic theorem prover)* est un outil qui
  - implémente un système déductif complet,
  - permet de construire de façon interactive une preuve (justification)
- De tels outils ne sont normalement pas adéquats, car on préfère utiliser des *stratégie* de preuves, plus effectives en pratique plutôt que d'explorer toutes les possibilités d'applications des règles.
- Souvent l'impossibilité d'établir une preuve peut être exploité pour déterminer qu'une propriété est finalement fausse.

Système déductif

- Un système déductif est un ensemble de règles qui servent à justifier des *affirmations* (propriétés).
- On débute donc avec une affirmation qu'on veut justifier.
- Les règles permettent de remplacer une affirmation par d'autres qui la justifie.
- Ceci construit *une dérivation* en forme d'arbre.
- Si toutes les feuilles de la dérivation sont justifiés, la dérivation est close et l'affirmation de départ est justifiée.

Résultats possibles

- Si on n'arrive pas à justifier toutes les feuilles d'une dérivation, il se peut que :
  - il existe une dérivation close (possiblement en revenant en arrière),
  - il n'existe pas de dérivation close car l'affirmation est fausse,
  - l'affirmation est vraie, mais le système déductif n'est pas assez fort pour la justifier (incomplétude).

Démonstrateur automatique

- Un démonstrateur automatique est un logiciel qui nous aide à construire des dérivations en :
  - déterminant quelles règles s'appliquent,
  - nous aidant à choisir une règle à appliquer à l'aide d'*heuristiques*,
  - réalisant l'application de la règle choisie.
- On peut aussi ajouter nos propres heuristiques et même nos propres règles.

Le système des séquents

- Le *séquent* $a_1, \ldots, a_n \vdash c_1, \ldots, c_m$ représente l'affirmation :
  - "si **TOUS** les énoncés $a_1, \ldots, a_n$ sont vérifiés alors **UN** des énoncés de $c_1, \ldots, c_m$ l'est aussi.
- Par exemple $a_1, a_1 \rightarrow a_2 \vdash a_2$ est un séquent valide,
- alors que ce n'est pas le cas pour $a_1 \rightarrow a_2 \vdash a_2$.
- Le système KeY note $\vdash$ par $==>$.

Calcul des séquents pour le calcul propositionnel

- Le calcul des séquents du calcul propositionnel possède les propriétés importantes suivantes.
  - Une belle symétrie gauche-droite :
    - les règles ou-droite et et-gauche se correspondent,
    - comme ou-gauche et et-droite.
  - Toutes les règles réduisent le nombre total de connecteurs ($\wedge$, $\vee$, $\neg$).
    - Donc toutes les branches d'une dérivation sont finies.
  - Un séquent sur lequel aucune règle ne s'applique ne contient plus aucun connecteur.

Algorithme de complétude pour le calcul propositionnel

- Étant donné un séquent du calcul propositionnel, l'algorithme suivant va soit le justifier, soit en retourner un contre-exemple.
  - Il suffit de produire une dérivation en développant les branches tant que des règles sont applicables.
  - Si toutes les branches sont closes, le séquent est établi et valide.
  - Sinon, en donnant la valeur vraie à toutes les variables de gauche et fausse à toutes celle de droite d'un séquent non-clos de la dérivation, on obtient un contre-exemple au séquent initial.

mise-à-jour KeY

- ```
  \programVariables { int x,y; }
  \problem {
      {x:=-(2*y+x) || y:=(2*y+x) }(x + y = 0)
  }
  ```
- Après le remplacement on obtient
  `-(2*y+x) + (2*y+x) = 0`,
- Ce qui est bien une égalité arithmétique, comme peut le montrer KeY.
  ```

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Notion de contrat

- Un contrat est une entente entre un *fournisseur* et un *client* par lequel
  - Le fournisseur s'engage à remplir ses *obligations* dans la mesure où ses *garanties* sont respectées.
  - Le client est prêt à remplir ses *obligations* pour pouvoir profiter des *garanties* prévues pour lui.

Pré et postconditions

- En génie logiciel la notion de contrat peut s'appliquer au niveau des fonctions (ou méthodes).
  - La réalisation d'une fonction doit remplir des obligations, exprimées par les *postconditions*, mais seulement dans la mesure où les garanties, exprimées par les *préconditions* sont remplies.
  - Tout usage de la fonction doit s'assurer que certaines obligations, les *préconditions*, sont respectées pour pouvoir profiter des garanties que sont les *postconditions*.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Historique

- Le *JAVA Modeling Language* (JML) est un langage de spécification spécifique à Java.
  - Contrairement à OCL, il ne s'agit pas d'une norme proposée par une organisation,
  - mais d'un projet de la communauté dirigé par Gary T. Leavens de l'Iowa State University www.jmlspecs.org.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Spécification

- L'approche JML est de considérer, pour un programme, une spécification formée :
  - d'invariants, qui contraignent les valeurs des attributs,
  - de contrats, qui contraignent le comportement des opérations.
- Un contrat est formé de préconditions, de postconditions et d'un ensemble de références qui peuvent être modifiées.

Obligations

- L'outil KeY génère automatiquement à partir d'une spécification JML des formules JavaDL.
- KeY utilise des gabarits qui produisent des formules (proof obligations), correspondant à l'approche de vérification des états observés.
- Il peut donc y avoir des façons différentes de justifier des choses analogues.
- C'est pourquoi il faut être attentif à la méthodologie de vérification utilisée ainsi qu'à ses avantages et limitations.

Le rôle des invariants

- Les invariants d'instance d'une classe sont des contraintes sur les attributs qui doivent être satisfaites sur des instances "correctes" de la classe.
- Les invariants font donc partie intégrante de la définition de la classe, même si le langage de programmation n'offre pas de support pour eux.
- Néanmoins, lors de la conception du système, les invariants sont, souvent implicitement, considérés comme étant satisfaits.
- En particulier, lors de la réalisation des opérations, on présuppose normalement que les invariants sont satisfaits sur le receveur (`self`).
  - L'opération sera correcte que sur des arguments corrects !

Le rôle des invariants en KeY

- Conformément à cette vision des choses, le système KeY ajoute les invariants aux préconditions lors de la vérification d'un contrat.
- Le choix des invariants est configurable au moment du choix du contrat (onglet "Assumed Invariants").
- Comme on l'a déjà vu, KeY ajoute déjà aux contrats les invariants implicites suivants.
  - `self` est créé et non-nul,
  - les arguments de l'opération sont créés et non-nuls.