Specyfikacje formalne

część II

Piotr Szczepański

Na podstawie:
Formal Specification, Andreas Roth, Peter H. Schmitt

Plan

- Krótkie przypomnienie
  - Co to jest formalna specyfikacja?
  - Co to jest OCL?
- JML
  - Składnia języka. Przykłady.
  - Wyrażenia
  - Kontrakty operacji w JML
  - Niezmienniki
  - Pola i metody modelowe
- Porównanie JML z OCL

Formalna specyfikacja

- Matematyczny opis oprogramowania, lub sprzętu.
- Specyfikacja stawianych wymagań.
- Formalne weryfikacje.
- Rafinacja programu, czyli transformacja specyfikacji w niskopoziomowy kod.

OCL

- Część standardu UML.
- Rozszerzenie diagramów.
- Język deklaratywny
- Specyfikuje
  - niezmienniki klas
  - warunki początkowe i końcowe operacji
  - wartości początkowe i ograniczenia atrybutów

Java Modeling Language JML

- Język specyfikacji dla Javy.
- Rozwijany przez społeczność, nie ma standaryzacji OMG.
- Używany w fazie implementacji.
- Specjalne komentarze w kodzie.
- Budowa oparta na niezmiennikach i kontraktach operacji.

Składnia języka
kontrakty operacji

- Warunki wstępne
  - Karta włożona do bankomatu
  - Pin poprawny
- Warunki końcowe
  - Użytkownik uwierzytelniony

Składnia języka
kontrakty operacji

```
... /*@
public normal_behavior
    requires włożonaKarta != null
    requires !użytkownikUwierzytelniony
    requires pin == włożonaKarta.poprawnyPIN
    assignable użytkownikUwierzytelniony
    ensures użytkownikUwierzytelniony
also  ...
    @*/
    public void wprowadźPIN (int pin) { ...
```

Kontrakt z bliska

- **requires**
  - określa warunki wstępne kontraktu
  - tworzy koniunkcję wszystkich warunków
- **assignable**
  - opisuje jakie elementy mogą się zmienić
  - lista rzeczy porozdzielanych przecinkami
- **ensures**
  - określa warunki końcowe kontraktu

Widoczność

```
private /*@ spec_public @*/ int length;
private int len;
public /*@ spec_protected @*/ int width;
public int wid;
/*@ public invariant length > 1 @*/
/*@ public invariant len > 1 @*/
/*@ public invariant width > 1 @*/
/*@ public invariant wid > 1 @*/
```

Widoczność

```
private /*@ spec_public @*/ int length;
private int len;
public /*@ spec_protected @*/ int width;
public int wid;
/*@ public invariant length > 1 @*/
/*@ public invariant len > 1 @*/ BŁĘDNE!
/*@ public invariant width > 1 @*/ BŁĘDNE!
/*@ public invariant wid > 1 @*/
```

Wyjątki

public exceptional_behavior

requires włożonaKarta == **null**

signals_only ATMException

signals(ATMException) !użytkownikUwierzytelniony

Wyjątki

- **signals_only**
  - lista dopuszczonych typów wyjątków
- **signals(E1)**
  - warunek końcowy w przypadku danego wyjątku

Czyste metody

```
public/*@ pure @*/ boolean czyKartaWłożona() {
    return włożonaKarta != null ;
}
```

- metody
  - bez efektów ubocznych
  - zawsze kończące swoje przebiegi

Niezmienniki
statyczne

```
public class KartaBankowa {
    /*@ public static invariant
    (\forall KartaBankowa p1, p2;
    \created(p1) && \created(p2);
    p1!=p2 ==> p1.numerKarty!=p2.numerKarty)
    @*/
    private  /*@ spec_public @*/ int numerKarty;
    ...
}
```

Niezmienniki obiektów

```
public class KartaBankowa {
    /*@ public instance invariant
    (\forall KartaBankowa p;
    this != p  ==> this.numerKarty != p.numerKarty)
    @*/
    private /*@ spec_public @*/ int numerKarty;
    ...
}
```

Wyrażenia JML

- Wszystkie wyrażenia Javy prócz:
  - operatorów powodujących efekt uboczny np: e++
  - operatorów przypisania
  - „nieczystych” metod
- Operatory:
  - implikacji (==>)
  - równoważności (<==>)
- Kwantyfikatory
  - \forall
  - \exist
  - \num_of, \sum, \product, \min, \max

Logika a wyrażenia

- e0!=e1 ~~> !([e0] = [e1])
- e0?e1:e2 ~~> if [e0] then [e1] else [e2]
- e0 ==> e1 ~~> [e0] → [e2]
- (\forall T e;e0;e1) ~~> ∀T e (([e] != null & [e0])→[e1])
- (\exists T e;e0;e1) ~~> ∃T e ([e] != null & [e0] & [e1])

Kwantyfikatory numeryczne

(**sum int** i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4

(**product int** i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4

(**max int** i; 0 <= i && i < 5; i) == 4

(**min int** i; 0 <= i && i < 5; i-1) == -1

Kwantyfikator uogólniony

```
public class CentralnySerwer{
    /*@ public instance invariant
    this.liczbaWażnychKart ==
                (\num_of KartaBankowa p;  !p.nieAktywna)
    ...
}
```

Kontrakty operacji w JML

- **ensures** E;
- **signals** (ET1) S1; ...
- **signals**(ETn) Sn;
- **signals_only** OT1, ... , OTm;
- (e=null → [E]) & (e ∈ [ET1] = TRUE → [S1])

  ...

  & (e ∈ [ETn] = TRUE → [Sn])

  & (e ∈ [OE1] = TRUE

  ...

  | e ∈ [OEm] = TRUE)

Kontrakty operacji w JML

- **assignable**
  - **\nothing , \everything**
- **diverges**
  - Warunek jaki musi być spełniony przed wywołaniem operacji, jeśli się ona nie zakończy.
  - **diverges** false
  - **diverges** true

Specjalne kontrakty
desugaring

- **normal_behavior**
  - **requires** R;
  - **assignable** A;
  - **ensures** E;
  - **diverges** D;
- **behavior**
  - **requires** R;
  - **assignable** A;
  - **ensures** E;
  - **diverges** D;
  - **signals** (Exception) **false**;

Specjalne kontrakty
desugaring

- **exceptional_behavior**
  - **requires** R;
  - **assignable** A;
  - **diverges** D;
  - **signals** (ET) S;
  - **signals_only**(OT);
- **behavior**
  - **requires** R;
  - **assignable** A;
  - **ensures false**;
  - **diverges** D;
  - **signals**(E) S;
  - **signals_only**(OT);

Czyste metody
desugaring

- Jak zastąpić adnotację /*@ **pure** @*/ przy danej metodzie odpowiem kontraktem?

Czyste metody
desugaring

- Jak zastąpić adnotację /*@ **pure** @*/ przy danej metodzie odpowiem kontraktem?

- Dodać kontrakt operacji zawierający
  - **assignable \nothing**
  - **diverges false**

Lekkie kontrakty

- Kontrakty nie rozpoczynające się od słowa kluczowego **behavior.**
- Różnica z „ciężkimi kontraktami” jedynie w domyślnych wartościach przyjmowanych przez niepodane klauzule.
  - **requires** **\not_specified** **true**
  - **assignable** **\not_specified** **\everything**
  - **ensures** **\not_specified** **true**
  - **diverges** **false** **false**
  - **signals** **\not_specified** (Exception) **true**

Opisywanie warunków końcowych

...

requires włożonaKarta != **null**;

requires pin != włożonaKarta.poprawnyPIN;

requires błędnychWprowadzeńPIN >= 2;

...

ensures włożonaKarta == **null**;

ensures **old**(włożonaKarta).nieAktywna;

...

Dziedziczenie dla kontraktów operacji

- Wszystkie kontrakty dla metod automatycznie przysługują metodom przedefiniowującym.

```
... /*@
also  jakiś_kontrakt
@*/
@Override
public void enterPIN (int pin) { ...
```

Niezmienniki w JML
niezmienniki obiektów

- Odnoszą się do zmiennej **this**
- Muszą być spełnione w każdym widzialnym stanie obiektu
  - wywołanie, ukończenie metody
  - zakończenie konstruktora klasy
  - kiedy żadna z metod, czy konstruktor nie jest w trakcie działania
- Metody z adnotacją /*@ helper @*/ zwolnione z zachowania niezmienników.

Niezmienniki w JML
niezmienniki statyczne

- Mogą się odnosić do pól obiektu, jedynie jeśli jest on pod kwantyfikatorem.

```
public class CentralnySerwer{
    public static int maxLiczbaKont;
    /*@ public instance invariant maxLiczbaKont >= 0 @*/
    /*@ public static invariant maxLiczbaKont >= 0 @*/
    ...
}
```

Pola i metody modelowe

- Rozszerzamy nasz przykład bankowy o interfejs

```
public interface IBonusowaKarta{
    public void dodajPunkty(int nowePunkty);
}
```

- Jak dodać specyfikacje dla operacji, jeśli interfejs nie posiada żadnych pól?

Pola i metody modelowe

```
public interface IBonusowaKarta {
    /*@ public instance model int punktyBonusowe; @*/
    /*@ ensures punktyBonusowe ==
                    \old(punktyBonusowe)+nowePunkty;
    assignable punktyBonusowe; @*/
    public void dodajPunkty (int nowePunkty);
}
```

Pola i metody modelowe
implementacja interfejsu

```
public class KartaBankowa implements
                                    IbonusowaKarta {
    /*@ public instance model int punktyBonusowe; @*/
    /*@ also
    assignable punktyKarty;
    @*/
    Public void dodajPunkty (int nowePunkty){
        punktyKarty+=nowePunkty;
    }
}
```

Pola i metody modelowe
ustalenie relacji

```
/*@
    private represents punktyBonusowe <- punktyKarty;
@*/


/*@
    private represents punktyBonusowe
                \such_that punktyBonusowe == punktyKarty;
@*/
```

Wspieranie weryfikacji

- Chcemy dowieść, że program spełnia podaną specyfikację.
- Dodatkowo możemy wprowadzać niezmienniki do pętli.

```
m = a[0]; i = 1;
while ( i < a.length) {
    /*@  ensures \forall integer x; 0>=x && x<i; a[x] >= m;
           assignable m,i  @*/
    if (a[i] < m) them m = a[i];
    i++;
}
```

Przewaga OCL and JML

- OCL wyższego poziomu, może być stosowane przed powstaniem kodu. Automatyczne generowanie ograniczeń ze wzorów cięższe na poziomie kodowania.
- OCL nie jest ukierunkowany na konkretny język programowania. Wygodniejszy w modelowaniu systemów.
- OCL jest standardem OMG, w przeciwieństwie do JML.

Przewaga JML nad OCL

- JML jest bliższy językowi Java co wpłynęło na jego znacznie większą popularność w porównaniu do OCL.
- JML oferuje wiele ciekawych koncepcji na poziomie implementacji np.:
  - opisywanie wyjątków
  - opisywanie modyfikowanych pól
  - niezmienniki pętli

Odnoszenie się do poprzedniego stanu

- @pre
  - Można stosować do indywidualnych symboli.
  - a@pre.b@pre.c@pre
  - a.b@pre.c
- \old
  - Jedynie do wyrażeń.
  - \old(a.b.c)
  - \old(a).b.c
  - a.b.\old(c) NIEPOPRAWNE

Odnoszenie się do poprzedniego stanu

- Chcemy wyrazić, że metoda zmienia tablice i w końcowym stanie a[0] = a[idx]
  - \old(a[idx]) BŁĘDNE!
  - a.get@pre(idx)

Odnoszenie się do poprzedniego stanu

- Chcemy wyrazić, że metoda zmienia tablice i w końcowym stanie a[0] = a[idx]
  - (\forall int x; x==idx; \old(a[x]) == a[0]);
  - a.get@pre(idx)

Modyfikowane lokacje

- JML
  - Jedynie lokacje wymienione za assignable mogą być zmieniane podczas działania metody.
- KeY
  - Podczas wywołania mogą się zmieniać.
- OCL
  - Nie ma jasnego określenia.
  - W rozszerzeniach do OCL zakłada się, że mogą się zmieniać jedynie lokacje wymienione w warunku końcowym

Zasięg kwantyfikatorów

- JML
  - Kwantyfikatory przebiegają po wszystkich elementach, nie tylko stworzonych, czy zaalokowanych.
  - Kwantyfikatory w niezmiennikach instancji przebiegają tylko po stworzonych.
- OCL
  - Jedynie po stworzonych obiektach.

Liczby całkowite

- JML
  - Używa typów Javy
  - 1073741821 * 1073741821 = -2147483639
  - 1073741822 * 1073741822 = 4
- KeY
  - Pozwala wybierać pomiędzy liczbami całkowitymi w rozumieniu matematycznym czy JAVA.
- OCL
  - Nie pozwala kwantyfikować po wszystkich liczbach.
  - Dopuszcza jedynie skończone zbiory.

Podsumowanie
zastosowania i przydatne narzędzia

- Kompilacja z asercjami, sprawdzanie podczas działania programu – *jmlc*.
- Tworzenie testów jednostkowych – *jmlunit*.
- Sprawdzanie statyczne – *ESC/JAVA2*.
- Weryfikacja programów – *LOOP,JACK,KeY*.
- Narzędzia wspomagające – *jmlspec,Canapa*.

Dziękuję za uwagę

Odpowiedzi na niektóre pytania
równość

- Jak interpretowany jest znak równości w JMLu?
  - Dokładnie tak samo jak w Javie, jest to równość referencji.
  - Jeśli chcielibyśmy porównywać stringi, używamy takiej jak w Javie notacji np:

    public String name;

    /*@ **public invariant** !name.equals(””);@*/

Odpowiedzi na niektóre pytania
diverges

- Kiedy właściwie ma być spełniony warunek początkowy przy słowie diverges?
  - Metoda zapętli się.
  - Metoda nie powróci do miejsca swojego wywołania.
    - Wyjątki nie podchodzą pod warunek diverges.
  - Np. dla metody System.exit warunek przy diverges zawsze będzie musiał być spełniony.