Techniki zabezpieczania kodu - kontrola wywołań

Temat IX

JML – warunki wejścia i wyjścia metody

- requires, ensures
- Można opisywać za pomocą wyrażeń warunki wejścia i wyjścia metod

```
/*@ requires amount >= 0;
    ensures balance == \old(balance-amount) &&
            \result == balance;
  @*/
  public int debit(int amount) {
    ...
  }
```

JML – warunki wejścia i wyjścia c.d.

- Specyfikacje w JML-u mogą być dowolnie silne/słabe

```
/*@ requires amount >= 0;
    ensures true;
  @*/
  public int debit(int amount) {
    ...
  }
```

Domyślny warunek wyjścia – `true` można opuścić.

JML – warunki wejścia i wyjścia c.d.

- Można specyfikować wiele par requires-ensures:

```
/*@   requires amount >= 0;
      ensures true;
    also
      requires amount < 0 && webInterface;
      ensures true;

  @*/
  public int debit(int amount) {
    ...
  }
```

- Semantyka:
  - metodę można wywołać bezkarnie, gdy dowolne z requires jest spełnione,
  - jeśli dane requires jest spełnione na wejściu, to spełnione jest też odpowiednie ensures

JML – niezmienniki

- Niezmienniki klasowe, w odróżnieniu od niezmienników pętli
- Muszą być utrzymywane przez wszystkie metody, np.

```
public class Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
  /*@ invariant 0 <= balance &&
                balance <= MAX_BAL;
    @*/
  ...
}
```

- Niezmienniki są niejawnie dołączane do wszystkich warunków wstępnych i warunków końcowych
- Niezmienniki muszą być zachowywane także po wyrzuceniu wyjątku!

JML – niezmienniki c.d.

- Niezmienniki dokumentują decyzje projektowe, np.:

```
public class Directory {
   private File[] files;
   /*@ invariant files != null &&
         (\forall int i; 0 <= i && i < files.length;
                  files[i] != null &&
                  files[i].getParent() == this)
     @*/
```

- Jawne zapisywanie niezmienników pomaga w zrozumieniu kodu.

JML – niezmienniki c.d.

- Rodzajem niezmiennika są niezmienniki temporalne `constraint`
- Opisują ewolucję stanu, np.:

```
public class Gadgets {
    int counter = 0;
   /*@ constraint counter >= \old(counter);
     @*/
```

Ostrzeżenia dla niezmienników klasowych

- Niezmienniki i klauzule `constraint` generują dodatkowe warunki końcowe dla każdej z metod
- Jeśli te warunki nie zachodzą, generowane są odpowiednie ostrzeżenia:

```
public class Invariant {
   public int i,j;
   //@ invariant i > 0;
   //@ constraint j > \old(j);

   public void m() {
      i = -1;  // spowoduje błąd Invariant
      j = j-1; // spowoduje błąd Constraint
   }
}
```

Klauzula assert

- Klauzula assert opisuje własność, która powinna zachodzić w pewnym punkcie programu, np.:

```
if (i <= 0 || j <0) {
    ...
} else if (j < 5) {
    //@ assert i>0 && 0<j && j<5;
    ...
} else {
    //@ assert i>0 && j>5;
    ...
}
```

- W Javie pojawiła się też możliwość specyfikacji @Assert (od Javy 1.4)

Klauzula assert c.d.

- Większe możliwości assert w JML-u

```
for (n = 0; n < a.length; n++) {
   if (a[n]==null) break;
   /*@ assert (\forall int i; 0 <= i && i < n;
                                  a[i] != null);
     @*/
}
```

Ostrzeżenia `Assert`

- Ostrzeżenie `Assert` pojawia się, gdy adnotacja przy `assert` nie może być spełniona
- W ESC/Java2 pojawia się także przy adnotacji

  ```
  //@ unreachable;
  ```

  spoza JML, równoważnej

  ```
  //@ assert false;
  ```

Ostrzeżenia `Assert`

- Przykład:

```
public class AssertWarning {
   //@ requires i >= 0;
   public void m(int i) {
      //@ assert i >= 0; // OK
      --i;
      //@ assert i >= 0; // ŹLE
   }

   public void n(int i) {
     switch (i) {
       case 0,1,2: break;
       default: //@ unreachable; // ŹLE
     } }
}
```

Efekty uboczne – assignable

- Własności ramek ograniczają możliwe efekty uboczne metod:

```
/*@
   requires amount >= 0;
   assignable balance;
   ensures balance == \old(balance)-amount;
@*/
   public int debit(int amount) {
     ...
   }
```

- To oznacza, że metoda `debit` może przypisywać tylko do pola `balance`.
- To nie wynika z warunku wyjścia
- Domyślna klauzula: `assignable \everything`

Efekty uboczne – pure

- Metoda bez efektów ubocznych jest określana jako pure

```
public /*@ pure @*/ int getBalance(){...}

Directory /*@ pure non null @*/ getParent(){...}
```

- Metody pure mają domyślnnie `assignable \nothing`.
- W specyfikacjach można używać wyłącznie metod pure, np.:

```
/*@ invariant 0<=getBalance() &&
               getBalance()<=MAX_BALANCE;
  @*/
```

Efekty uboczne

- Ostrzeżenie `Modifies` w ESC/Java2 oznacza próbę przypisania na obiekt, który nie znajduje się w klauzuli modifies
- Uwaga: tylko niektóre naruszenia dadzą się stwierdzić na etapie kompilacji
- Uwaga druga: jest to obszar aktywnych badań naukowych

Efekty uboczne – ostrzeżenia `Modifies`

- Przykład ostrzeżenia `Modifies` z ESC/Java2:

```
public class ModifiesWarning {
    int i;

    //@ assignable i;
    void m(/*@ non_null */ ModifiesWarning o) {
        i = 1;
        o.i = 2; // ostrzeżenie Modifies
    }
}
```

Efekty uboczne – ostrzeżenia Modifies

nie wiemy, czy `o` jest równe `this`, a ponieważ można przypisywać tylko na `this.i`, to mamy ostrzeżenie:

```
-------------------------------------------------------------
ModifiesWarning.java:7: Warning: Possible violation of
                        modifies clause
    o.i = 2; // Modifies warning
    ^
Associated declaration is "ModifiesWarning.java", line 4, col 6:
    //@ assignable i;
-------------------------------------------------------------
```

Opisywanie pętli

- Klauzula `loop_invariant` tuż przed pętlą opisuje własność, która jest spełniona przed każdym wejściem do pętli oraz przy wyjściu z niej
- Gdy nie jest spełniona pojawia się ostrzeżenie `LoopInv`
- Klauzula `decreases` tuż przed petlą opisuje wartość (typu `int`), która jest nieujemna i zmniejsza się przy każdym obrocie pętli
- Gdy się nie zmniejsza, to pojawia się ostrzeżenie `DecreasesBound`
- Uwaga: pętle są sprawdzane przez rozwijanie, a nie przez punkt stały

Opisywanie pętli

- Przykład:

```
public class LoopInvWarning {
   public int max(/*@ non_null */ int[] a) {
      int m;
      /*@ loop_invariant (\forall int j; 0<=j && j<i;
                                    a[j] <= m);
        @*/
      //@ decreases a.length -  1;
      for (int i=0; i<a.length; ++i) {
          if (m < a[i])
             m = a[i];
      }
      return m;
  }
}
```

- Zmienna `i` z wnętrza pętli jest w zasięgu deklaracji

Opisywanie konstruktorów

- Normalne adnotacje requires, ensures
- Niezmienniki i historyczne warunki
  - przed wejściem nie zakładane
  - na końcu zapewniane
- Klauzule initially

Opisywanie konstruktorów c.d.

- Ostrzeżenie `Initially` pojawia się, gdy nie można stwierdzić, że klauzula Initially zachodzi
- Analiza

```
public class Initially {
    public int i; //@ initially i == 1;

    public Initially(){
    } // nie ustawia i - ostrzeżenie Initially
}
```

daje

Opisywanie konstruktorów c.d.

```
------------------------------------------------------------
Initially.java:5: Warning: Possible violation of initially
                  condition at constructor exit (Initially)
    public Initially() { } // does not set i - Initially warning
                         ^
Associated declaration is "Initially.java", line 3, col 20:
public int i; //@ initially i == 1;
                  ^
------------------------------------------------------------
```

Aliasy

- Często powoduje naruszenie niezmienników

```
public class Alias {
   /*@ non_null */ int[] a = new int[10];
   boolean noneg = true;
   /*@ invariant noneg ==> (\forall int i; 0<=i && i < a.length;
                                    a[i]>=0); */

  //@ requires 0<= i && i < a.length;
  public void insert(int i, int v) {
     a[i] = v;
     if (v < 0)
       noneg = false;
  }
}
```

daje

Aliasy c.d.

```
-------------------------------------------------------------
Alias.java:12: Warning: Possible violation of object invariant
               (Invariant)
                }
                ^
Associated declaration is "Alias.java", line 5, col 6:
   /*@ invariant (\forall int i; 0<=i && i < a.length;
-------------------------------------------------------------
```

Aliasy c.d.

- Można ESC/Java2 namówić, aby wyświetliło więcej informacji
- Kontekst kontrprzykładu (opcja -counterexample) wygląda tak (dużo więcej tam jest):

```
brokenObj%0 != this
(brokenObj%0).(a@pre:2.24) == tmp0!a:10.4
this.(a@pre:2.24) == tmp0!a:10.4
```

  z czego można odczytać, że `this` oraz pewien inny obiekt `brokenObj` współdzielą ten sam obiekt `a`

Aliasy c.d.

- Można temu zapobiec deklarując właściciela
- Idea: `a` powinno być własnością tylko jednego obiektu
- Pole `owner` jest polem rodzaju `ghost` w java.lang.Object
- Pola `ghost` widoczne i zmienialne tylko w specyfikacjach

Aliasy c.d.

- Kod z poprawionymi specyfikacjami:

```
public class Alias {
   /*@ non_null */ int[] a = new int[10];
   boolean noneg = true;
  /*@ invariant noneg ==> (\forall int i; 0<=i && i < a.length;
                                    a[i]>=0); */
  //@ invariant a.owner == this; ÂL’ ÂŚ

  //@ requires 0<= i && i < a.length;
  public void insert(int i, int v) {
     a[i] = v;
     if (v < 0)
       noneg = false;
  }

  public Alias() {
     //@ set a.owner = this;
  }
}
```

Aliasy c.d.

- Inny przykład.
- Tym razem popsuty jest post-warunek

```
public class Alias2 {
  /*@ non_null */ Inner n = new Inner();
  /*@ non_null */ Inner nn = new Inner();
  //@ invariant n.owner == this;
  //@ invariant nn.owner == this;

  //@ ensures n.i == \old(n.i + 1);
  public void add() {
    n.i++;
    nn.i++;
  }
  Alias2();
}
class Inner {
  public int i;
  //@ ensures i == 0;
  Inner(); }
}
```

Aliasy c.d.

- Kontekst kontrprzykładu pokazuje:

  ```
  this.(nn:3.24) == tmp0!n:10.4
  tmp2!nn:11.4 == tmp0!n:10.4
  ```

- Sugerują one, że `n` oraz `nn` odwołują się do tego samego obiektu
- Jeśli dodamy niezmiennik:

  ```
  //@ invariant n != nn;
  ```

  wykluczający aliasowanie, wszystko działa bez problemu.

Aliasy c.d.

- Aliasy stanowią duże utrudnienie przy weryfikacji
- Aliasy stanowią duże utrudnienie przy rozumieniu programów
- Obsługa aliasów to aktywna dziedzina badań związana z obsługą ramek metod
- Chodzi o to, aby wiedzieć, co jest modyfikowane, a co nie
- Pola `owner` tworzą rodzaj enkapsulacji pozwalający ESC/Java2 na dokładniejsze określenie, co może być zmodyfikowane w wyniku danej operacji
- Do JML-a dodany został system typów uniwersowych, który zapewnia panowanie nad aliasami

Jak pisać specyfikacje? – generalizacja

- Zapisuj niezmienniki obiektów
- Upewnij się, że niezmienniki dotyczą one aktualnego obiektu
- Zdania o wszystkich obiektach klasy mogą być prawdziwe, ale trudne do udowodnienia (zwłaszcza dla automatów)
- Jeśli na przykład zdanie P zachodzi dla obiektów typu T, to nie pisz

  ```
  //@ invariant (\forall T t; P(t));
  ```

  ale

  ```
  //@ invariant P(this);
  ```

  To ostatnie jest łatwiejsze do udowodnienia, bo bardziej konkretne są punkty zachodzenia.

Jak pisać specyfikacje? – niespójne założenia

- Jeśli napiszesz niespójne specyfikacje, to możesz udowodnić wszysko:

```
public class Inconsistent {
   public void m() {
      int a,b,c,d;
      //@ assume a == b;
      //@ assume b == c;
      //@ assume a != c;
      //@ assert a == d; // Przechodzi, ale niespójne
      //@ assert false;  // Przechodzi, ale niespójne
   }
}
```

Jak pisać specyfikacje? – niespójne założenia

- Inny przykład

```
public class Inconsistent2 {
  public int a,b,c,d;
  //@ invariant a == b;
  //@ invariant b == c;
  //@ invariant a != c;
  public void m() {
    //@ assert a == d; // Przechodzi, ale niespójne
    //@ assert false;  // Przechodzi, ale niespójne
  }
}
```

Może któregoś dnia będzie to sprawdzane...

Jak pisać specyfikacje? – referencje

- Gdy referencja do wewnętrznego obiektu jest eksportowana, pojawiają się problemy:

```
public class Exposed {
  /*@ non_null */ private int[] a = new int[10];
  //@ invariant a.length > 0 && a[0] >= 0;

  //@ ensures \result != null;
  //@ ensures \result.length > 0;
  //@ pure
  public int[] getArray() { return a; }
}

class X {
  void m(/*@ non_null */ Exposed e) {
    e.getArray()[0] = -1; // unchecked invariant violation
  }
}
```

ESC/Java2 nie sprawdza, czy każdy zaalokowany obiekt spełnia swoje niezmienniki

Jak pisać specyfikacje? – referencje

- Gdy referencja do wewnętrznego obiektu jest eksportowana, pojawiają się problemy:

```
public class Exposed {
  /*@ non_null */ private int[] a = new int[10];
  //@ invariant a.length > 0 && a[0] >= 0;

  //@ ensures \result != null;
  //@ ensures \result.length > 0;
  //@ pure
  public int[] getArray() { return a; }
}

class X {
  void m(/*@ non_null */ Exposed e) {
    e.getArray()[0] = -1; // unchecked invariant violation
  }
}
```

- Podobne ukryte problemy mogą się pojawiać, gdy pola publiczne są bezpośrednio modyfikowane

Jak pisać specyfikacje? – `\old`

- `\old` jest wykorzystywane do zaznaczenia w post-warunku wyliczenia w stanie wejściowym metody
- Spróbujmy wyspecyfikować:

```
public static native void
   arraycopy(Object[] src, int srcPos,
             Object[] dest, int destPos, int length)
```

Zacznijmy od:

```
ensures (\forall int i; 0<=i && i<length;
                 dest[destPos+i] == src[srcPos+i]);
```

Jak pisać specyfikacje? – `\old`

Źle! Oprócz wyjątków i niepoprawnych argumentów należy pamiętać o aliasach – `dest` i `src` mogą być tą samą tablicą:

```
ensures (\forall int i; 0<=i && i<length;
                 dest[destPos+i] == \old(src[srcPos+i]);
```

I nie należy zapominać o pozostałych elementach:

```
ensures (\forall int i; (0<=i && i<destPos) ||
                        (destPos+length <= i &&
                         i < destPos.length);
                        dest[i] == \old(dest[i]);
```

Jak pisać specyfikacje? – `\old`

- Czy w post-warunku

```
ensures (\forall int i; 0<=i && i<length;
                dest[destPos+i] == \old(src[srcPos+i]);
```

  nie powinniśmy napisać `\old(length)` zamiast `length`?

Jak pisać specyfikacje? – `\old`

- Czy w post-warunku

```
ensures (\forall int i; 0<=i && i<length;
                dest[destPos+i] == \old(src[srcPos+i]);
```

  nie powinniśmy napisać `\old(length)` zamiast `length`?

- Właściwie: tak, ale ponieważ łatwo o tym zapomnieć, dla dowolnego argumentu `x` wyrażenie `x` w post-warunku oznacza naprawdę `\old(x)`.

- Oznacza to, że nie można w post-warunku odwoływać się do nowej wartości długości, ale ta wartość jest i tak nie do zaobserwowania przez klientów.

Jak pisać specyfikacje? – podsumowanie

- Zacznij od procedur bazowych i bibliotecznych
- Dla każdego pola: czy jest z nim związany niezmiennik?
- Dla każdego pola referencyjnego: czy nie powinno być `non_null`?
- Dla każdego pola referencyjnego: czy nie powinno mieć właściciela?
- Dla każdej metody: czy nie powinna być `pure`?

Jak pisać specyfikacje? – podsumowanie c.d.

- Dla każdej metody: czy argumenty i wynik nie powinny być `non_null`?
- Dla każdej klasy: jaki niezmiennik wyraża spójność wewnętrznych danych?
- Dodaj pre- i post-warunki, aby ograniczyć dane wejściowe i wyniki metod
- Dodaj możliwe nieraportowane wyjątki do klauzuli `throws`
- Zaczynaj od prostych specyfikacji, przechodź do bardziej skomplikowanych w miarę potrzeb

Jak pisać specyfikacje? – podsumowanie

- Rozdzielaj koniunkcje, aby uzyskać lepsze wskazania na temat tego, co jest nie tak. Używaj

  ```
  requires A;
  requires B;
  ```

  zamiast

  ```
  requires A && B;
  ```

- Używaj `assert`, aby zorientować się, co jest nie tak
- Używaj `assume`, aby pomóc mechanizmowi dowodzenia
- Używaj `assume` TYLKO, gdy wiesz, co robisz.

Pliki ze specyfikacjami

- Specyfikacje można dodawać bezpośrednio do plików .java
- Specyfikacje można dodawać alternatywnie do specjalnych plików ze specyfikacjami:
  - nie ma ciał metod
  - nie ma inicjalizatorów pól
  - sufiksy: .spec, .refines-java, .jml, .refines-java
- Muszą znajdować się na ścieżce klas lub specyfikacji