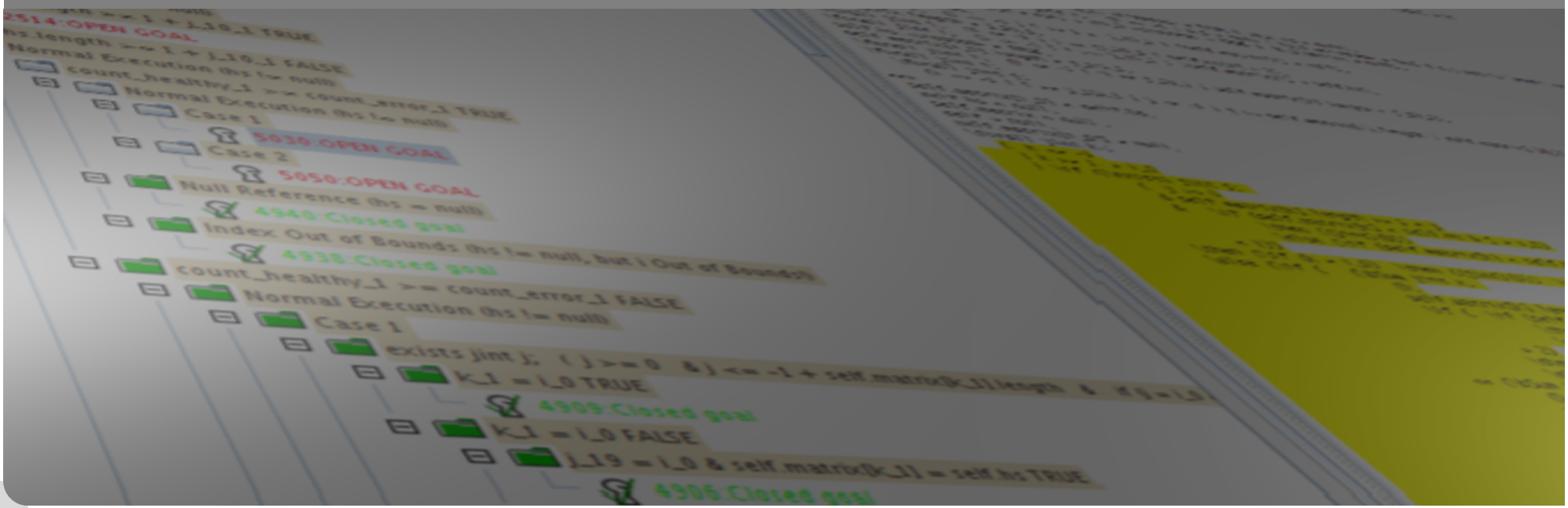
Formale Systeme

Prof. Dr. Peter H. Schmitt

KIT – Institut für Theoretische Informatik



JML

Java Modeling Language

Kurzcharakterisierung

Historie

- Initiator Gary Leavens
- erste Publikation 1999
- seither kontinuierlicher Aufbau einer weltweiten *community*

Grundlagen

- Softwaretechik: *design by contract* , Softwareverträge
- Logik: Hoare Kalkül

Aktuelle Informationen

Webseite `http://www.cs.ucf.edu/~leavens/JML/`

```
public class PostInc{
  public PostInc rec; public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @    rec.x>=0 && rec.y>=0;
    @*/

  /*@ public normal_behavior
    @ requires true;
    @ ensures rec.x == \old(rec.y) &&
    @           rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

JML Annotationen sind spezielle Kommentare im Quelltext
Vorbedingung Nachbedingung Normale Terminierung:
Terminierung und keine Ausnahme wird ausgelöst (*no*

JML Ausdrücke

Das syntaktische Material, aus dem JML Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

In der Klasse `PostInc` deklarierte Felder Operationen und Literal aus den Java Datentypen `int` und `boolean`. Anwendungsoperator.

Voll ausgeschrieben sieht die Invariante so aus:

```
this.x >= 0 && this.y >= 0 && rec.x >= 0 &&
                rec.y >= 0
```

Alternative zum `old` Operator

```
@ requires oldrecy == rec.y;
@ ensures rec.x == oldrecy && rec.y == oldrecy+1;
```

wobei `oldrecy` im nachfolgenden Code nicht auftritt.

Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `self.rec == null` gilt?

Es wird eine *NullPointerException* ausgelöst.

Somit würde die Methode ihren Vertrag nicht erfüllen, denn es wird normale Terminierung verlangt.

Kein Problem.
JML nimmt als Voreinstellung, als *default*, an, daß alle vorkommenden Attribute und Parameter mit einem Objekttyp vom Nullobjekt verschieden sind.

Überschreiben der Voreinstellung

```
public class PostInc{
 public  /*@ nullable @*/ PostInc rec;
 public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @   (rec != null ==> rec.x>=0 && rec.y>=0);
    @*/
  /*@ public normal_behavior
    @ requires rec != null;
    @ ensures rec.x == \old(rec.y) &&
    @         rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

Syntax zum Überschreiben der Voreinstellung Verstärkte Vorbedingung jetzt erforderlich Änderung der Invarianten erforderlich

Übungsaufgabe

Was ist die richtige Nachbedingung?

```
public class PostIncxx{
 public PostInc rec;
 public int x;
  /*@  public invariant x>=0 && rec.x>=0;
    @*/

  /*@ public normal_behavior
    @ requires true;
    @ ensures ???;
    @*/
 public void postinc() {rec.x = rec.x++;}}
```

```
class SITA{ public int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @   r <= a1.length && r <= a2.length;
   @  assignable \nothing;
   @  ensures ( l <= \result && \result < r &&
   @  a1[\result] ==  a2[\result])
   @   || \result == r ;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j] );
   @*/
public int  commonEntry(int l, int r) {...}}
```

Zwei Deklarationen von Nachbedingungen werde wie ihre Konjunktion behandelt. JML Schlüsselwort für den Rückgabewert einer Methode, falls vorhanden. Die Methode soll einen Index i im Intervall [l, r) finden, für

Quantoren in JML

Syntax

```
(\forall C x;B;R)        (\exists C x;B;R)
```

`B` *Bereichseinschränkung*
`R` *Rumpf*

Bedeutung in prädikatenlogischer Notation

$$\forall x^C(B \rightarrow R) \qquad \exists x^C(B \wedge R)$$

Beispiel

`(\forall C x; B; R)` und
`(\forall C x; true; (B ==> R))`
sind äquivalent.

Vergleich der Notationen

JML*	Prädikatenlogik
`==`	$\doteq$
`&&`	$\wedge$
`||`	$\vee$
`!`	$\neg$
`==>`	$\rightarrow$
`<==>`	$\leftrightarrow$
`(\forall C x;e1;e2)`	$\forall x((x \neq \mathit{null} \wedge [e1]) \rightarrow [e2])$
`(\exists C x;e1;e2)`	$\exists x(x \neq \mathit{null} \wedge [e1] \wedge [e2])$

Dabei steht $[ei]$ für die prädikatenlogische Notation des JML Ausdrucks `ei`.

Fortsetzung

```
class SITA{ public int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @   r <= a1.length && r <= a2.length;
   @  assignable \nothing;
   @  ensures ( l <= \result && \result < r &&
   @  a1[\result] ==  a2[\result])
   @   || \result == r ;
   @  ensures (\forall int j; l <= j && j < \result;
   @   a1[j] !=  a2[j] );
   @*/
public int  commonEntry(int l, int r) {...}}
```

Vorbedingung: Einschränkungen an die Parameter. Die *assignable* Klausel gibt an, welche Werte die nachfolgende Methode höchstens ändern darf. Die Methode `commonEntry`

Schleifenspezifikationen

```
public int  commonEntry(int l, int r)
  {int k = l;
/*@ loop_invariant
  @   l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
 while(k < r) {
   if(a1[k] == a2[k]){break;}
     k++;}
 return k;}
```

Schlüsselwort für Schleifeninvariante (*loop invariant*)
Schleifeninvariante *assignable* Klausel auch für Schleifen
Variante, sichert Terminierung Details folgen jetzt

Aufgabe von Schleifeninvarianten

Die Angabe einer Schleifeninvarianten *SI* verlangt, daß

1. (Anfangsfall)
   SI vor Eintritt in die Schleife erfüllt ist,
2. (Iterationsschritt)
   für jeden Programmzustand s_0, in dem *SI* und die Schleifenbedingung gilt, im Zustand s_1, der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit s_0 erreicht wieder, wieder *SI* erfüllt ist.
3. (Anwendungsfall)
   nach Beendigung der Schleife reicht die Schleifeninvarianten *SI* zusammen mit den Bedingungen für die Schleifenterminierung aus für die Verifikation der Nachbedingung.

Sind diese drei Forderungen in unserem Beispiel erfüllt?

Nachprüfung: Anfangsfall

```
  int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @    (\forall int i; l<=i && i<k; a1[i] != a2[i]);
 while(k<r) {if(a1[k] == a2[k]){break;} k++;}
```

Die Schleifeninvariante wird zu

```
 l <= l && l <= r &&
(\forall int i; l<=i && i<l; a1[i] != a2[i]);
```

Triviale Identität Steht in der Vorbedingung Leerer Bereich des Quantors

Nachprüfung: Iterationsschritt

```
/*@ loop_invariant  l <= k && k <= r &&
  @    (\forall int i; l<=i && i<k; a1[i] != a2[i]);
 while(k<r) {if(a1[k] == a2[k]){break;} k++;}
```

Vor dem Schleifendurchlauf

```
l<=k && k<=r &&(\forall int i;l<=i && i<k;a1[i !=a2[i])
&&  k < r
```

Nach dem Schleifendurchlauf, $k \rightsquigarrow k+1$

```
l<=k+1 && k+1<=r &&
(\forall int i;l<=i && i<k+1;a1[i !=a2[i])
```

Nachprüfung: Anwendungsfall

```
  int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
 while(k<r) {if(a1[k] == a2[k]){break;} k++;}
 return k;
```

Fallunterscheidung

1. Die Schleife terminiert weil `k=r` erreicht wurde
2. Es gilt `k<r` und die Schleife terminiert weil `a1[k] == a2[k]` gilt.

Nachprüfung: Anwendungsfall

Fall `k=r`

Schleifeninvariante

```
l <= k && k <= r && k = r &&
\forall int i; l<=i && i<k; a1[i] != a2[i]);
```

Nachbedingung

```
((l<=\result && \result<r && a1[\result]==a2[\result])
     || \result == r)
 &&
(\forall int j; l<=j && j<\result;a1[j] != a2[j])
```

Nach Beendigung der Schleife vor Ende der Methode wird noch die Anweisung `return k` ausgeführt

Nachprüfung: Anwendungsfall

Fall `k<r` und `a1[k] == a2[k]`

Schleifeninvariante

```
l <= k && k <= r && k < r && a1[k] == a2[k] &&
\forall int i; l<=i && i<k; a1[i] != a2[i]);
```

Nachbedingung

```
((l<=\result && \result<r && a1[\result]==a2[\result])
     || \result == r)
 &&
(\forall int j; l<=j && j<\result;a1[j] != a2[j])
```

Nach Beendigung der Schleife vor Ende der Methode wird noch die Anweisung `return k` ausgeführt

Terminierung

```
public int  commonEntry(int l, int r)
  int k = l;
/*@
  @  decreases a1.length - k;
  @*/
 while(k < r) {
   if(a1[k] == a2[k]){break;}
     k++}
```

Der Ausdruck nach dem `decreases` Schlüsselwort heißt die *Variante* der Schleife. Die Variante ist stets ≥ 0 Die Variante wird in jedem Schleifendurchlauf echt kleiner.

Verallgemeinerte Quantoren in JML

Generalized Quantifiers

```
(\sum       T x; R ; t)
(\product   T x; R ; t)
(\max       T x; R ; t)
(\min       T x; R ; t)
```

Beispiele

```
(\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
```

Methodenvertrag

```
class SumAndMax {
  int sum; int max;
/*@ public normal_behaviour
  @  requires (\forall int i;0<=i && i<a.length;0<=a[i]);
  @  assignable sum, max;
  @  ensures (\forall int i;0<=i && i<a.length;a[i]<=max);
  @  ensures (a.length > 0
  @    ==> (\exists int i;0<=i && i<a.length;max == a[i]));
  @  ensures sum == (\sum int i;0<=i && i <a.length; a[i]);
  @  ensures sum <= a.length * max;
  @*/
 void sumAndMax(int[] a) { ....}}
```