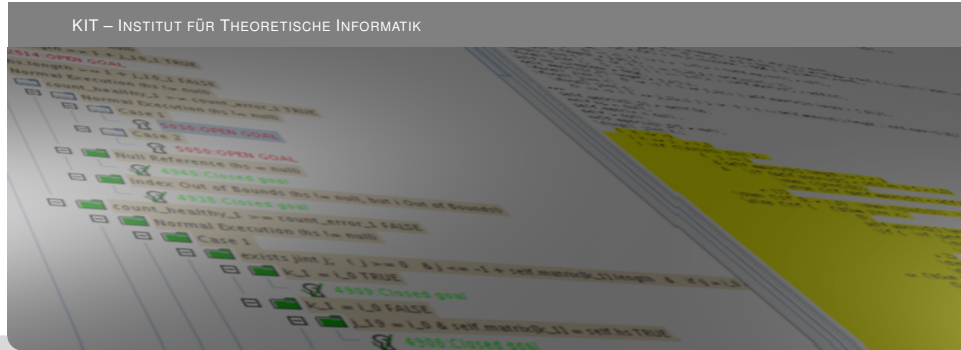


Formale Systeme

Prof. Dr. Peter H. Schmitt

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



JML

Java Modeling Language

Historie

Historie

- ▶ Initiator Gary Leavens

Historie

- ▶ Initiator Gary Leavens
- ▶ erste Publikation 1999

Historie

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

Historie

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

Grundlagen

Historie

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

Grundlagen

- ▶ Softwaretechnik: *design by contract* , Softwareverträge

Historie

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

Grundlagen

- ▶ Softwaretechnik: *design by contract* , Softwareverträge
- ▶ Logik: Hoare Kalkül

Historie

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

Grundlagen

- ▶ Softwaretechnik: *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++;}}
```

```
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

```

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++;}
}

```

Vorbedingung

```

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++;}
}

```

Nachbedingung

```

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++;}}

```

Normale Terminierung: Terminierung und keine Ausnahme wird ausgelöst (*no exception thrown*)

```

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++;}}

```

Invariante


```

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++;}
}

```

Zugriff im Nachzustand auf den Wert eines JML Ausdruck im Vorzustand

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
```

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

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
```

Operationen und Literal aus den Java Datentypen `int` und `boolean`.

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
```

Anwendungsoperator.

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
```

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?

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.

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.

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++;}}
```

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

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++;}}
```

Verstärkte Vorbedingung jetzt erforderlich

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++;}}
```

Änderung der Invarianten erforderlich

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) {...}}

```

```

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.

```

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) {...}}

```

JML Schlüsselwort für den Rückgabewert einer Methode, falls vorhanden.

```

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) {...}}

```

Die Methode `commonEntry` soll einen Index i im Intervall $[l, r)$ finden, für den die beiden Felder `a1,a2` denselben Wert haben oder i ist gleich der rechten Suchgrenze.

```

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) {...}}

```

Außerdem, sollen für alle Indizes kleiner als i die Felder a_1, a_2 verschiedene Werte haben.

Syntax

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

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

B *Bereichseinschränkung*

R *Rumpf*

Syntax

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

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

B *Bereichseinschränkung*

R *Rumpf*

Bedeutung in prädikatenlogischer Notation

$\forall x^C (B \rightarrow R)$

$\exists x^C (B \wedge R)$

Syntax

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

B *Bereichseinschränkung*

R *Rumpf*

Bedeutung in prädikatenlogischer Notation

$\forall x^C (B \rightarrow R)$

$\exists x^C (B \wedge R)$

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
$(\backslash \text{forall } C \ x; e1; e2)$	$\forall x((x \neq \text{null} \wedge [e1]) \rightarrow [e2])$
$(\backslash \text{exists } C \ x; e1; e2)$	$\exists x(x \neq \text{null} \wedge [e1] \wedge [e2])$

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

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) {...}}
```

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.

```
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) {...}}
```

Die *assignable* Klausel gibt an, welche Werte die nachfolgende Methode höchstens ändern darf.

```
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) {...}}
```

Die Methode `commonEntry` soll nichts ändern.
Sie ist eine *reine Methode* (*pure method*)

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;}
```

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*)

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;}
```

Schleifeninvariante

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;}
```

assignable Klausel auch für Schleifen

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;}
```

Variante, sichert Terminierung

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;}
```

Details folgen jetzt.

Aufgabe von Schleifeninvarianten

Die Angabe einer Schleifeninvarianten SI verlangt, daß

1. (Anfangsfall)
 SI vor Eintritt in die Schleife erfüllt ist,

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.

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.

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.

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?


```
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++;}
```

```
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]);
```

```
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

```
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]);
```

Steht in der Vorbedingung

```
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]);
```

Leerer Bereich des Quantors

```
/*@ 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++;}
```

```
/*@ 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
```

```
/*@ 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])
```



```
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;
```

```
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

```
int k = 1;
/*@ 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

```
int k = 1;
/*@ 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.

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])
```

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 $a1[k] == a2[k]$

Schleifenvariante

```
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])
```

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++}
    }
```

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.

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++;
    }
```

Die Variante ist stets ≥ 0

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++}
```

Die Variante wird in jedem Schleifendurchlauf echt kleiner.

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

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
```

```
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) { ....}}
```