

Formale Systeme

Prof. Dr. Bernhard Beckert, WS 2016/2017

Java Modeling Language (JML)

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK

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
- ▶ Standardisierungsvorhaben seit 2016

Historie

- ▶ Initiator Gary Leavens
- ▶ Erste Publikation 1999
- ▶ Seither kontinuierlicher Aufbau einer weltweiten Community
- ▶ Standardisierungsvorhaben seit 2016

Grundidee

Design by contract

Historie

- ▶ Initiator Gary Leavens
- ▶ Erste Publikation 1999
- ▶ Seither kontinuierlicher Aufbau einer weltweiten Community
- ▶ Standardisierungsvorhaben seit 2016

Grundidee

Design by contract

JML Einführung, weitere Informationen

<http://www.key-project.org/thebook2/>, Chapter 7

```

public class PostInc {
    PostInc rec;
    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 {
    PostInc rec;
    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 {
    PostInc rec;
    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 {
    PostInc rec;
    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 {
    PostInc rec;
    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 {
    PostInc rec;
    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 {
    PostInc rec;
    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
```

Wie in Java steht `x >= 0` für `this.x >= 0`

Alternative zum `old` Operator

```
PostInc oldrec;
```

```
/*@ ...
```

```
  @ requires oldrec.y == rec.y;
```

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

wobei `oldrec.y` im nachfolgenden Code nicht auftritt.

Zeiger auf Null (null pointer)

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

Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `this.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 `this.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 `this.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++;}}
```

Lösung

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

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

```

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

```

```

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

```

Zwei Deklarationen von Nachbedingungen werde wie ihre Konjunktion behandelt.

```

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

```

JML-Schlüsselwort für den Rückgabewert einer Methode, falls vorhanden.

```

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

$(\backslash\text{forall } C \ x; \ B; \ R)$

$(\backslash\text{exists } C \ x; \ B; \ R)$

B *Bereichseinschränkung*

R *Rumpf*

Syntax

$(\backslash\text{forall } C \ x; \ B; \ R)$ $(\backslash\text{exists } C \ x; \ B; \ R)$

B *Bereichseinschränkung*

R *Rumpf*

Bedeutung in prädikatenlogischer Notation

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

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

Syntax

$(\backslash\text{forall } C \ x; \ B; \ R)$ $(\backslash\text{exists } C \ x; \ B; \ R)$

B *Bereichseinschränkung*

R *Rumpf*

Bedeutung in prädikatenlogischer Notation

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

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

Beispiel

$(\backslash\text{forall } C \ x; \ B; \ R)$ und
 $(\backslash\text{forall } C \ x; \ \text{true}; \ (B \implies R))$
sind äquivalent.

JML	Prädikatenlogik
<code>==</code>	$\doteq$
<code>& &</code>	$\wedge$
<code> </code>	$\vee$
<code>!</code>	$\neg$
<code>==></code>	$\rightarrow$
<code><==></code>	$\leftrightarrow$
<code>(\forall x C x; e1; e2)</code>	$\forall x (\neg x \doteq null \wedge [e1]) \rightarrow [e2]$
<code>(\exists x C x; e1; e2)</code>	$\exists x (\neg x \doteq null \wedge [e1] \wedge [e2])$

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

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

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

Vorbedingung: Einschränkungen an die Parameter.

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

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

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

Details folgen jetzt.

Die Angabe einer Schleifeninvarianten SI verlangt, dass

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

Die Angabe einer Schleifeninvarianten SI verlangt, dass

1. (Anfangsfall)
 SI vor Eintritt in die Schleife gilt,
2. (Iterationsschritt)
für jeden Programzustand s_0 , in dem SI und die Schleifenbedingung gilt, im Zustand s_1 , der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit s_0 erreicht wird, wieder SI gilt.

Die Angabe einer Schleifeninvarianten SI verlangt, dass

1. (Anfangsfall)
 SI vor Eintritt in die Schleife gilt,
2. (Iterationsschritt)
für jeden Programmmzustand s_0 , in dem SI und die Schleifenbedingung gilt, im Zustand s_1 , der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit s_0 erreicht wird, wieder SI gilt.
3. (Anwendungsfall)
nach Beendigung der Schleife reicht die Schleifeninvariante SI zusammen mit den Bedingungen für die Schleifenterminierung aus, um die Nachbedingung zu beweisen.

Die Angabe einer Schleifeninvarianten SI verlangt, dass

1. (Anfangsfall)
 SI vor Eintritt in die Schleife gilt,
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 wird, wieder SI gilt.
3. (Anwendungsfall)
nach Beendigung der Schleife reicht die Schleifeninvariante SI zusammen mit den Bedingungen für die Schleifenterminierung aus, um die Nachbedingung zu beweisen.

Sind diese drei Forderungen in unserem Beispiel erfüllt?

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

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

Die Schleifeninvariante wird zu

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

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

Die Schleifeninvariante wird zu

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

Triviale Identität

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

Die Schleifeninvariante wird zu

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

Folgt aus der Vorbedingung

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

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

Invariante vor dem Schleifendurchlauf

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

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

Invariante vor dem Schleifendurchlauf

$l \leq k \ \&\& \ k \leq r \ \&\& \ (\forall \text{int } i; l \leq i \ \&\& \ i < k; a1[i] \neq a2[i])$

Schleifenbedingung

$k < r \ \&\& \ a1[k] \neq a2[k]$

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

Invariante vor dem Schleifendurchlauf

$l \leq k \ \&\& \ k \leq r \ \&\& \ (\forall \text{int } i; l \leq i \ \&\& \ i < k; a1[i] \neq a2[i])$

Schleifenbedingung

$k < r \ \&\& \ a1[k] \neq a2[k]$

Inv. nach dem Schleifendurchlauf: $k \rightsquigarrow k + 1$

$l \leq k+1 \ \&\& \ k+1 \leq r \ \&\& \$
 $(\forall \text{int } i; l \leq i \ \&\& \ i < k+1; a1[i] \neq a2[i])$

```
int k = 1;
/*@ loop_invariant  l <= k && k <= r &&
   @    (\forall int i; l<=i && i<k; a1[i] != a2[i]);
   @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

```
int k = 1;
/*@ loop_invariant  l <= k && k <= r &&
   @    (\forall int i; l<=i && i<k; a1[i] != a2[i]);
   @*/
while(k < r && a1[k] != a2[k]) 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 && a1[k] != a2[k]) k++;
return k;
```

Fallunterscheidung

1. Die Schleife terminiert, weil $k < r$ nicht mehr gilt

```
int k = 1;
/*@ loop_invariant  l <= k && k <= r &&
   @    (\forall int i; l<=i && i<k; a1[i] != a2[i]);
   @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

Fallunterscheidung

1. Die Schleife terminiert, weil $k < r$ nicht mehr gilt
2. Es gilt $k < r$, und die Schleife terminiert, weil $a1[k] == a2[k]$ gilt.

Schleifeninvariante

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

Negierte Schleifenbedingung (Fall 1)

```
k >= r
```

Nachbedingung (nach Schleife und `return k;`)

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

Schleifeninvariante

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

Negierte Schleifenbedingung (Fall 2)

```
k < r && a1[k] == a2[k]
```

Nachbedingung (nach Schleife und `return k;`)

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

Terminierung

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

Terminierung

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

Der Ausdruck nach dem Schlüsselwort `decreases` heißt die *Variante* der Schleife.

Terminierung

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

Die Variante ist stets ≥ 0

Terminierung

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

```
(\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) { ....}}
```