Formale objektorientierte Software-Entwicklung

Prof. Dr. Rolf Hennicker

07.07.2005

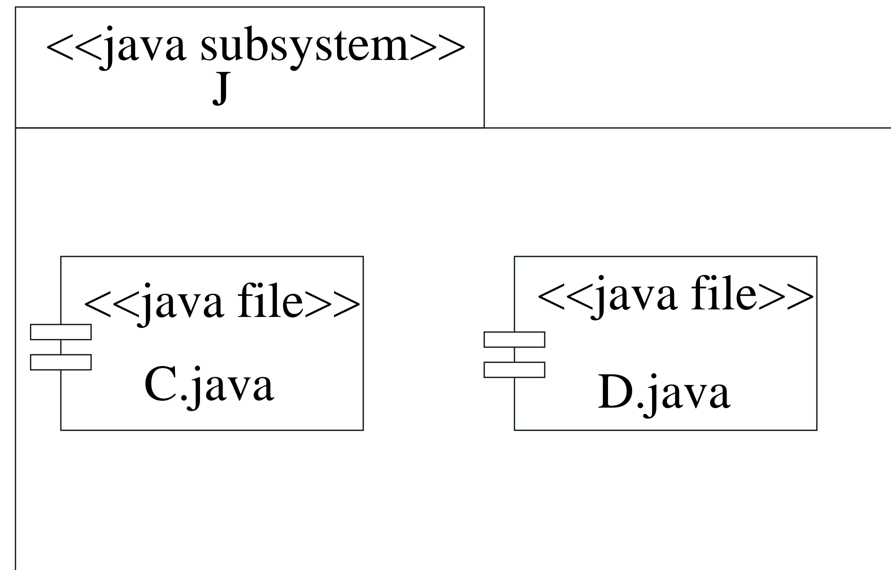
MiS
MIS

Kapitel 5:
Java Realisierungen von Komponentenspezifikationen

Ziele

- Wissen, was ein Java Subsystem ist.
- Wissen, wann ein Java Subsystem konform zu einer Klassensignatur Σ_Δ ist.
- Das durch ein Java Subsystem induzierte Σ_Δ-Transitionssystem konstruieren können.
- Verstehen, wann ein Java Subsystem eine korrekte Realisierung einer Komponentenspezifikation ist.
- Die (vereinfachten) Beweisverpflichtungen kennen, die (unter bestimmten Annahmen) die Korrektheit einer Java Realisierung garantieren.
- Das Theorem über die Korrektheit einer Java Realisierung kennen und verstehen.

5.1 Java Subsysteme



Ein Java Subsystem besteht aus einer Menge von Dateien `C.java`, so dass jede Datei `C.java` (o.B.d.A. genau) eine Deklaration einer Java Klasse `C` enthält.

Definition (Konformität):

Sei Σ_Δ eine Klassensignatur. Ein Java Subsystem J ist konform zu Σ_Δ falls gilt:

1. Für jeden Klassennamen $C \in Class_\Delta$ existiert eine entsprechende Datei `C.java` in J u.u.
2. Für jedes Attribut $(_.a : C \to T) \in A_\Delta$ existiert eine entsprechende Instanzvariable (mit demselben Namen) in `C.java` und umgekehrt.
3. Für jede Operation $(op : C \times T_1 \times \ldots \times T_n \to T) \in M_\Delta \cup Q_\Delta$ existiert eine entsprechende Methode in `C.java` und umgekehrt;
   für jeden Konstruktor $(C : T_1 \times \ldots \times T_n \to C) \in Con_\Delta$ existiert ein entsprechender Konstruktor in `C.java` und umgekehrt.
4. Für alle Klassen $C, B \in Class_\Delta$ ist B genau dann eine direkte Oberklasse von C (d.h. $C < B$ und $\nexists D \in Class_\Delta : C < D < B$) wenn die Java Klasse C (in `C.java`) vermöge "extends" direkt von der Java Klasse B (in `B.java`) erbt.
5. Die Sichtbarkeiten von Attributen, Rollen und Operationen in $A_\Delta \cup Opns_\Delta$ werden erhalten, d.h.
   "$-$" wird abgebildet auf "private", "#" auf "protected"
   "$\sim$" wird abgebildet auf Java-Default-Sichtbarkeit,
   "$+$" wird abgebildet auf "public" innerhalb einer öffentlichen Java Klasse.
6. Für jede Queryoperation $(q : C \times T_1 \times \ldots \times T_n \to T) \in Q_\Delta$ besitzt die entsprechende Java Methode keine Seiteneffekte.

Bemerkung:

1. Basistypen von OCL werden der Java Syntax entsprechend umbenannt, z.B. $Integer \mapsto int$.

2. OCL Kollektionstypen werden durch Java Kollektionstypen ersetzt.

   Insbesondere gilt:
   Für jeden Rollennamen $(_.a : C \rightarrow Set(T)) \in A_\Delta$ existiert ein entsprechendes Referenzattribut (z.B. `Set a;`) in `C.java`.

3. Die Argument- und Ergebnistypen von Operationen $op \in Opns_\Delta$ werden von den entsprechenden Java Methoden und Konstruktoren respektiert, z.B.

$$[createPoint : System \times Real \times Real \rightarrow Point] \mapsto [Point\ createPoint(double\ x,\ double\ y)]$$

$$[move : Point \times Real \times Real \rightarrow Void] \mapsto [void\ move(double\ mx,\ double\ my)]$$

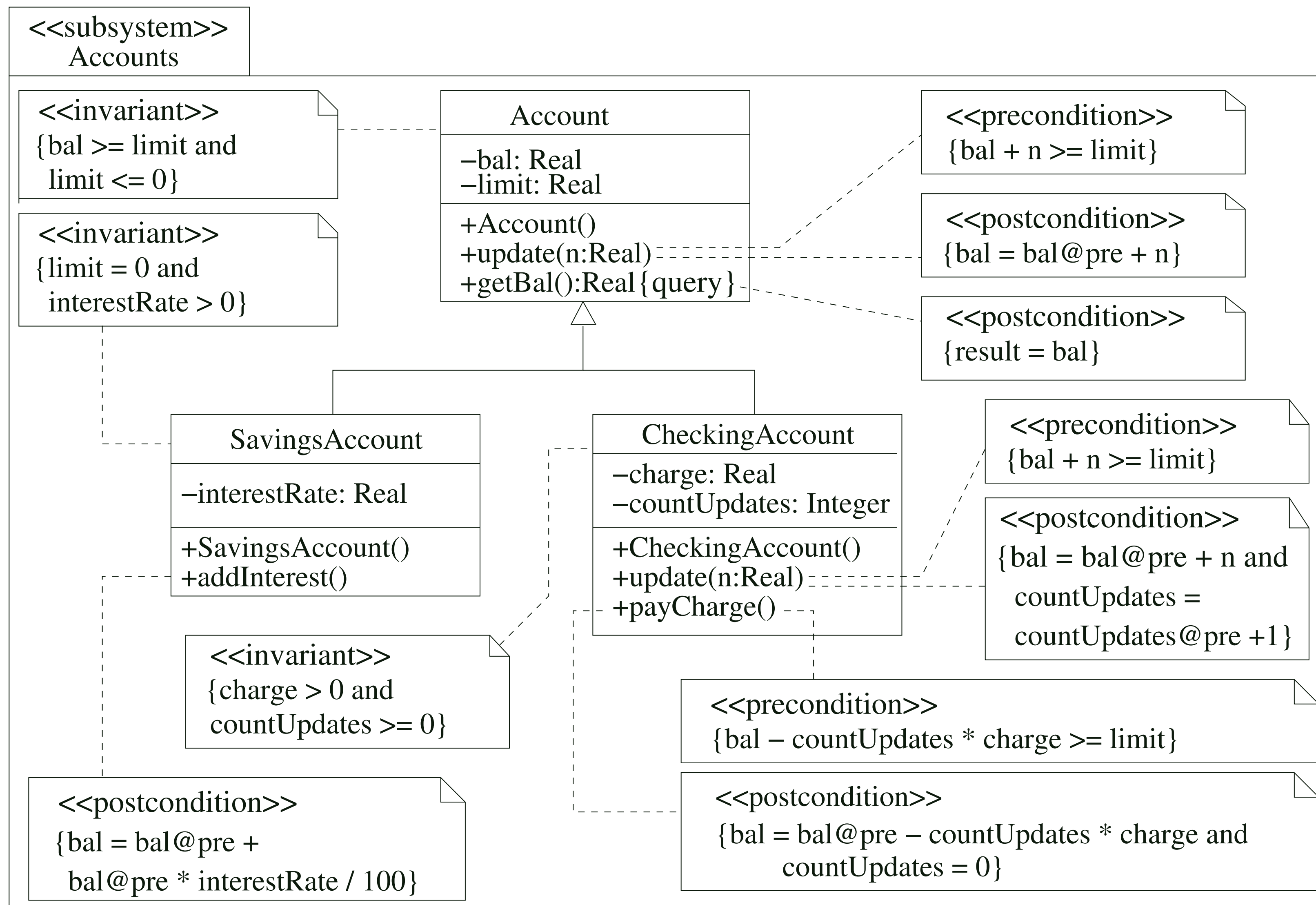
$$[Point : Real \times Real \rightarrow Point] \mapsto [Point(double\ x,\ double\ y)]$$

Beispiel (Java Subsystem für Counters):

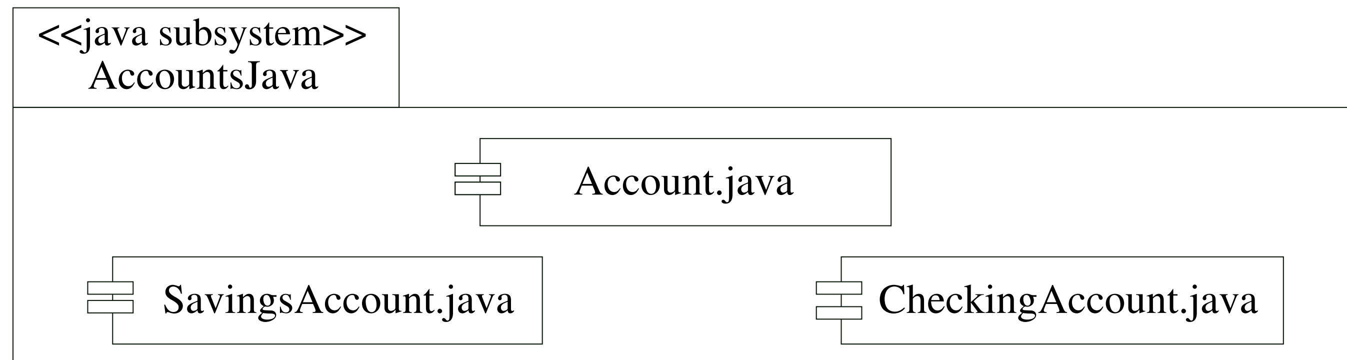
```
package CountersJava;
public class Counter {
  protected int count;

  public Counter() {
    count = 0;
  }

  public void inc() {
    count++;
  }

  public void dec() {
    count--;
  }
}
```

```
package CountersJava;
public class MCounter extends Counter {
  private int last;

    public MCounter() {
    count = 0;
    last = 0;
  }
  public void inc() {
    last = count;
    count++;
  }
  public void dec() {
    last = count;
    count--;
  }
}
```

Beispiel (Komponentenspezifikation für Accounts):



Beispiel (Java Subsystem AccountsJava):




```
package AccountsJava;
public class Account
{
  private double bal;
  private double limit;
  public Account(){
    bal = 0;limit = 0;}
  public void update(double n){
     bal = bal + n;}
  public double getBal(){
    return bal;}
}
```

```
package AccountsJava;
public class SavingsAccount extends Account
{
  private double interestRate;

  public SavingsAccount(){
    super(); interestRate=2.5;}
  public void addInterest(){
    update(interestRate/100);
  }
}
```

```
package AccountsJava;
public class CheckingAccount extends Account
{
  private double charge;
  private int countUpdates;

  public CheckingAccount(){
    super();
    charge = 8.5;
    countUpdates = 0;
  }
  public void update(double n){
    super.update(n);
    countUpdates = countUpdates + 1;
  }

  public void payCharge(){
    super.update(-countUpdates*charge);
  }
}
```

Definition (Durch ein Java Subsystem induziertes Σ_Δ-Transitionssystem):

Sei Σ_Δ eine Klassensignatur und J ein Java Subsystem, das konform zu Σ_Δ ist.
J induziert das Σ_Δ-Transitionssystem

$$\mathcal{T}_\Delta^J = (State_\Delta, \sigma_{init}, Label_\Delta, \Omega_\Delta, R_\Delta^J)$$

wobei R_Δ^J die kleinste Relation ist, die folgende Bedingungen erfüllt:

1. Für alle $(op : C \times T_1 \times \ldots \times T_n \longrightarrow Void) \in M_\Delta$ mit zugehöriger Java Methode $void\ op(T_1\ x_1, \ldots, T_n\ x_n)\ body$ der Klasse C ist

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \in State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq null$
und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$ und den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren,

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \bot \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \in State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq null$
und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$ und den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren,

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \bot \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \notin State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$ oder $o = null$.

2. Für alle $(op : C \times T_1 \times \ldots \times T_n \longrightarrow T) \in M_\Delta \cup Q_\Delta$, $T \neq \mathit{Void}$ mit zugehöriger Java Methode $T\ op(T_1\ x_1, \ldots, T_n\ x_n)\ \mathit{body}$ der Klasse C ist

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n):r} \sigma \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \in \mathit{State}_\Delta$ X $(\llbracket C \rrbracket \times \llbracket T_1 \rrbracket \times \ldots \times \llbracket T_n \rrbracket)$, $o \neq \mathit{null}$ und body angewandt auf σ^- und die lokale Umgebung mit dem Wert o für this und den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren mit Ergebniswert r,

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \bot \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \in \mathit{State}_\Delta$ X $(\llbracket C \rrbracket \times \llbracket T_1 \rrbracket \times \ldots \times \llbracket T_n \rrbracket)$, $o \neq \mathit{null}$ und body angewandt auf σ^- und die lokale Umgebung mit dem Wert o für this und den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren,

$$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \bot \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, o, v_1, \ldots, v_n) \notin \mathit{State}_\Delta$ X $(\llbracket C \rrbracket \times \llbracket T_1 \rrbracket \times \ldots \times \llbracket T_n \rrbracket)$ oder $o = \mathit{null}$.

3. Für alle $(C : T_1 \times \ldots \times T_n \longrightarrow C) \in Con_\Delta$ mit zugehörigem Java Konstruktor $C(T_1\ x_1, \ldots, T_n\ x_n)\ body$ ist

$$\sigma^- \xrightarrow{C(v_1,\ldots,v_n):o} \sigma \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, v_1, \ldots, v_n) \in State_\Delta$ X $([\![T_1]\!] \times \ldots \times [\![T_n]\!])$ und $body$ angewandt auf σ^- und die lokale Umgebung mit den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren mit dem neu erzeugten Objekt o,

$$\sigma^- \xrightarrow{C(v_1,\ldots,v_n)} \perp \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, v_1, \ldots, v_n) \in State_\Delta$ X $([\![T_1]\!] \times \ldots \times [\![T_n]\!])$ und $body$ angewandt auf σ^- und die lokale Umgebung mit den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren,

$$\sigma^- \xrightarrow{C(v_1,\ldots,v_n)} \perp \in R^J_\Delta \qquad \text{falls}$$

$(\sigma^-, v_1, \ldots, v_n) \notin State_\Delta$ X $([\![T_1]\!] \times \ldots \times [\![T_n]\!])$.

Beispiel ($\mathcal{T}^{\mathrm{CountersJava}}_{\mathrm{Counters}}$)

Das Java Subsystem $\mathrm{CountersJava}$ induziert das Σ_Δ-Transitionssystem $\mathcal{T}^{\mathrm{CountersJava}}_{\mathrm{Counters}} = (State_\Delta, \sigma_{init}, Label_\Delta, \Omega_\Delta, R^{\mathrm{CountersJava}}_\Delta)$ mit

$$
\begin{aligned}
R^{\mathrm{CountersJava}}_\Delta = R_\Delta & \cup \{ \sigma^- \xrightarrow{null.inc_{\mathrm{Counter}}()} \bot \} \\
& \cup \{ \sigma^- \xrightarrow{null.dec_{\mathrm{Counter}}()} \bot \} \\
& \cup \{ \sigma^- \xrightarrow{null.inc_{\mathrm{MCounter}}()} \bot \} \\
& \cup \{ \sigma^- \xrightarrow{null.dec_{\mathrm{MCounter}}()} \bot \}
\end{aligned}
$$

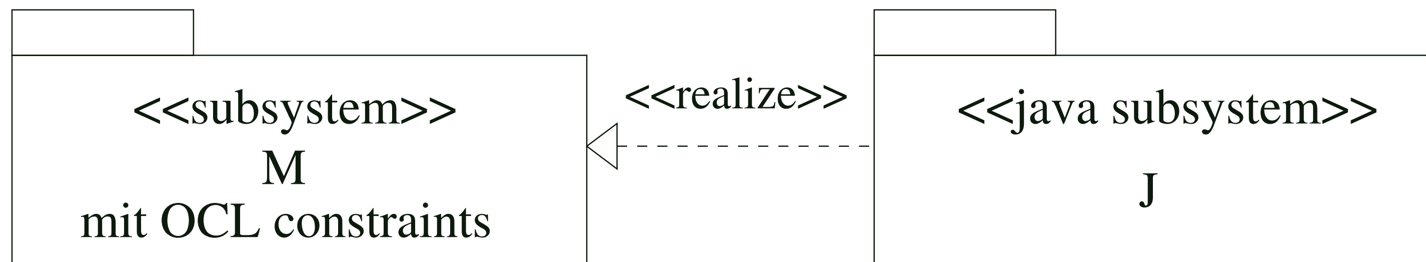
wobei R_Δ die in Abschnitt 4.4 definierte Transitionsrelation von $\mathcal{T}_{\mathrm{Counters}}$ ist.

5.2 Korrekte Java Realisierungen

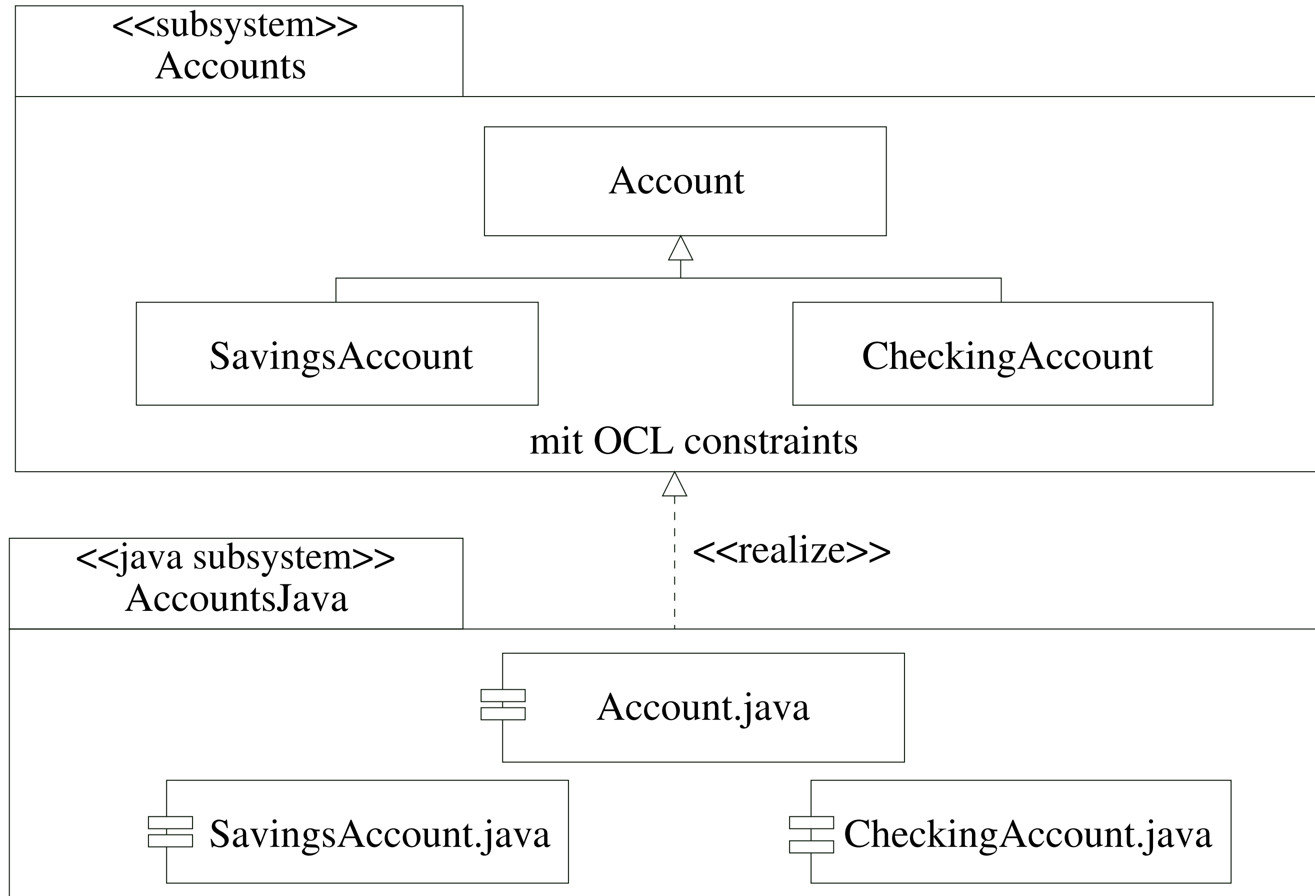
Definition (Korrekte Java Realisierung):
Sei $CompSpec = (\langle M, \Delta \rangle, OpSpecs, Invs^e)$ eine Komponentenspezifikation und sei J ein Java Subsystem. J ist eine *korrekte Java Realisierung* von $CompSpec$ falls die folgenden zwei Bedingungen erfüllt sind:

1. J ist konform zu Σ_Δ
2. Das durch J induzierte Σ_Δ-Transitionssystem $\mathcal{T}_\Delta^J$ ist eine korrekte Realisierung von $CompSpec$.

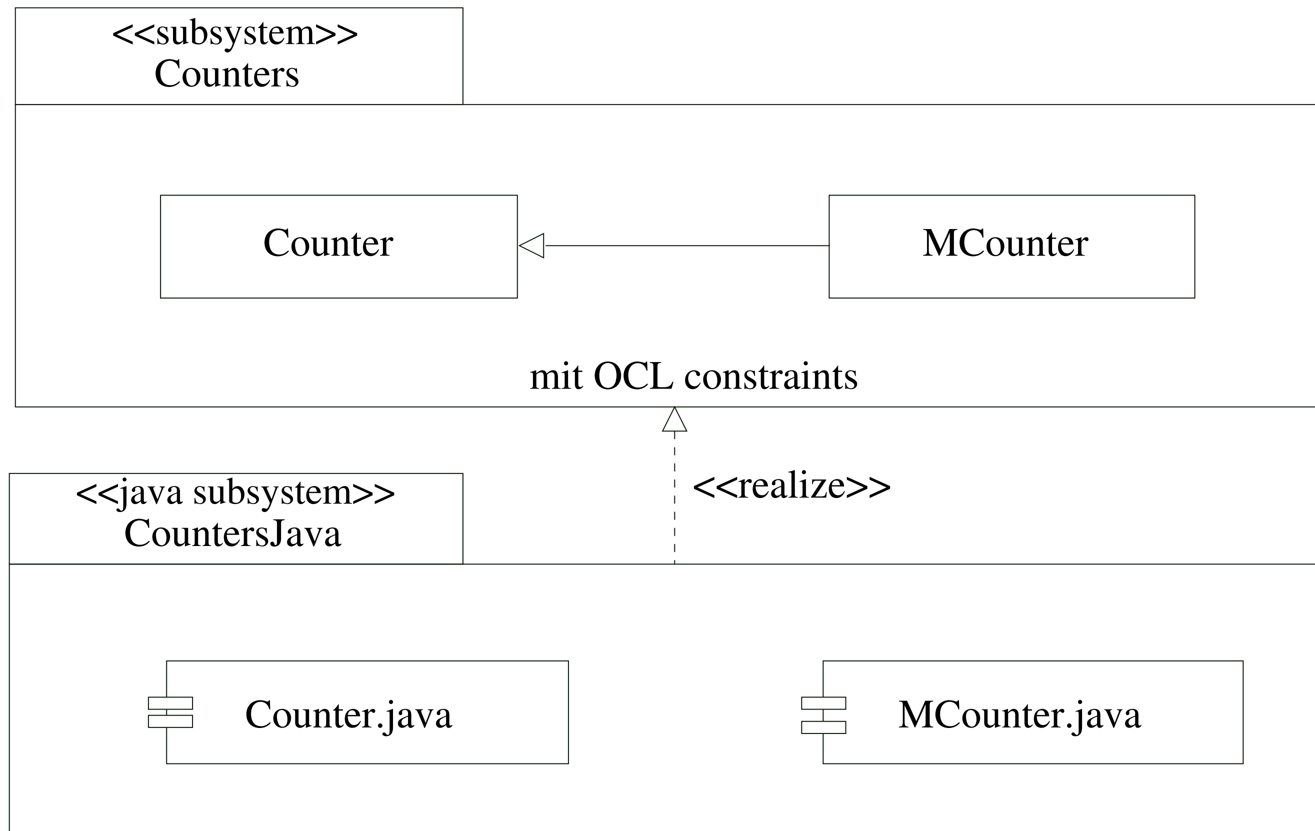
Notation:



Beispiel (AccountsJava realisiert Accounts):



Beispiel (CountersJava realisiert Counters):



Im Folgenden sind wir an vereinfachten Beweisverpflichtungen für korrekte Java Realisierungen interessiert.

Idee:

In der Nachbedingung $Q^{Visibility(op)}$ wollen wir statt $CLASSINV_{\Delta}$ nur die Klasseninvarianten für $self$ überprüfen.

Generelle Voraussetzungen für Komponentenspezifikationen

- Alle Klasseninvarianten erfüllen das Lokalitätsprinzip.
- Es gilt die Verhaltensverträglichkeit bzgl. Subtypen.
- Alle Komponenteninvarianten gelten im Anfangszustand σ_{init}, d.h. $[\![COMPINV_M]\!]_{\beta,\sigma_{init},\sigma_{init}} = true$.

Generelle Voraussetzungen für Java Programme

- Es gibt keine privaten Konstruktoren in Java Klassen.
- Alle Attribute (d.h. Instanzvariablen) von Java Klassen sind "private" oder "protected".
- Auf geschützte (protected) Attribute wird höchstens in der deklarierenden Klasse oder in deren Subklassen zugegriffen.
- Innerhalb einer Klasse können Zugriffe auf private oder geschützte Attribute nur für $this$ erfolgen.

  Nicht erlaubt ist z.B.

```
class C {
  private int a;

  void op(C x) {
    x.a = 0;
  }
}
```

Definition (Beweisverpflichtung $PObs^{CompSpec}$):
Sei $CompSpec$ eine Komponentenspezifikation mit formaler Repräsentation $FRep(CompSpec) = (\langle M, \Sigma_\Delta \rangle, OpSpecs, Invs)$.
Sei $C \in Class_\Delta$ und Q ein OCL-Ausdruck vom Typ Boolean.

$$FULLINV_C = \bigwedge_{A \geq C} INV_A \qquad \text{("volle" Klasseninvariante von C)}$$

$$\begin{aligned} Simpl_C(Q^+) &= Q \ and \ COMPINV_M \ and \ FULLINV_C \ and \\ &\quad \bigwedge_{D<C} (self.oclIsTypeOf(D) \ implies \ FULLINV_D) \end{aligned}$$

$$\begin{aligned} Simpl_C(Q^\sim) &= Q \ and \ FULLINV_C \ and \\ &\quad \bigwedge_{D<C} (self.oclIsTypeOf(D) \ implies \ FULLINV_D) \end{aligned}$$

$$Simpl_C(Q^-) = Q$$

Für jedes $op \in Opns_\Delta$ ist die zugehörige Beweisverpflichtung $PObs_{op}^{CompSpec}$ definiert wie folgt:

1. Sei $(op : C \times T_1 \times \ldots \times T_n \rightarrow Void) \in M_\Delta$ eine Methode ohne Rückgabewert.

$$\begin{aligned} PObs_{op}^{CompSpec} \quad = \quad \mathbf{context} \quad & C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ \mathbf{pre}: \quad & P^{Visibility(op)} \\ \mathbf{post}: \quad & Simpl_C(Q^{Visibility(op)}) \end{aligned}$$

wobei $(\mathbf{context}\ C :: op(x_1 : T_1, \ldots, x_n : T_n)\ \mathbf{pre} : P\ \mathbf{post} : Q) \in OpSpecs$.

2. Analog für Methoden mit Rückgabewert, die keine Queries sind.

3. Sei $(q : C \times T_1 \times \ldots \times T_n \rightarrow T) \in Q_\Delta$ eine Query.

$$
\begin{aligned}
PObs_q^{CompSpec} \quad = \quad \textbf{context} \quad & C :: q(x_1 : T_1, \ldots, x_n : T_n) : T \\
\textbf{pre}: \quad & P^{Visibility(q)} \\
\textbf{post}: \quad & Q
\end{aligned}
$$

wobei $(\textbf{context } C :: q(x_1 : T_1, \ldots, x_n : T_n) : T \ \textbf{pre} : P \ \textbf{post} : Q) \in OpSpecs$.

4. Sei $(C : T_1 \times \ldots \times T_n \rightarrow C) \in Con_\Delta$ ein Konstruktor.

$$
\begin{aligned}
PObs_C^{CompSpec} \quad = \quad \textbf{context} \quad & C :: C(x_1 : T_1, \ldots, x_n : T_n) \\
\textbf{pre}: \quad & P^{Visibility(C)} \\
\textbf{post}: \quad & Q \text{ and } COMPINV_M \text{ and } FULLINV_C \\
& (\text{ohne } COMPINV_M \text{ falls } Visibility(C) = \sim)
\end{aligned}
$$

wobei $(\textbf{context } C :: C(x_1 : T_1, \ldots, x_n : T_n) \ \textbf{pre} : P \ \textbf{post} : Q) \in OpSpecs$.

Die Menge der durch $CompSpec$ induzierten Beweisverpflichtungen für Java Programme ist definiert durch

$$PObs^{CompSpec} = \{PObs_{op}^{CompSpec} \mid op \in Opns_{\Delta}\}$$

Bemerkung:

In manchen Fällen kann $P^{Visibility(op)}$ durch eine schwächere und kürzere Vorbedingung ersetzt werden (die weniger Invarianten voraussetzt).

Beispiel (Beweisverpflichtungen für $\mathrm{Counters}$**)**

Zunächst bestimmen wir die vollen Klasseninvarianten:

$$FULLINV_{Counter} = \text{count} >= 0$$

$$FULLINV_{MCounter} = \text{count} >= 0 \text{ and last} >= 0$$

Die Beweisverpflichtungen für die Operationen lauten (nach Vereinfachung der Vorbedingungen):

context Counter::Counter()
post : count = 0

context Counter::inc()
pre : count >= 0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 0))
post : count = count@pre + 1 and
count >= 0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 0))

context Counter::dec()
pre : count >0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 0))
post : count = count@pre - 1 and count >= 0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 0))

context MCounter::MCounter()
post : count = 0 and last = 0

context MCounter::inc()
pre : count >= 0 and last >= 0
post : count = count@pre + 1 and last = last@pre and count >= 0 and last >= 0

context MCounter::dec()
pre : count >0 and last >= 0
post : count = count@pre - 1 and last = last@pre and count >= 0 and last >= 0

Beispiel (Beweisverpflichtungen für Accounts)

Wir bestimmen die vollen Klasseninvarianten:

$FULLINV_{Account}$ = bal >= limit and limit <= 0

$FULLINV_{SavingsAccount}$ = bal >= limit and limit = 0 and interestRate >0

$FULLINV_{CheckingAccount}$ = bal >= limit and limit <= 0 and charge >0 and countUpdates >= 0

$COMPINV_{Accounts}$ = true

Die Beweisverpflichtungen für die Operationen lauten (nach Vereinfachung der Vorbedingungen):

context Account::Account()
post : bal >= limit and limit <= 0

context Account::update(n: Real)
pre : bal + n >= limit and $FULLINV_{Account}$ and
self.oclIsTypeOf(SavingsAccount) implies $FULLINV_{SavingsAccount}$ and
self.oclIsTypeOf(CheckingAccount) implies $FULLINV_{CheckingAccount}$
post : bal = bal@pre + n and $FULLINV_{Account}$ and
self.oclIsTypeOf(SavingsAccount) implies $FULLINV_{SavingsAccount}$ and
self.oclIsTypeOf(CheckingAccount) implies $FULLINV_{CheckingAccount}$

context Account::getBal():Real
pre : $FULLINV_{Account}$
post : result = bal

context SavingsAccount::SavingsAccount()
post : $FULLINV_{SavingsAccount}$

context SavingsAccount::addInterest()
pre : $FULLINV_{SavingsAccount}$
post : bal = bal@pre + bal@pre * interestRate/100 and $FULLINV_{SavingsAccount}$

context CheckingAccount::CheckingAccount()
post : $FULLINV_{CheckingAccount}$

context CheckingAccount::update(n: Real)
pre : bal + n >= limit and $FULLINV_{CheckingAccount}$
post : bal = bal@pre + n and countUpdates = countUpdates@pre + 1 and $FULLINV_{CheckingAccount}$

context CheckingAccount::payCharge()
pre : bal - countUpdates * charge >= limit and $FULLINV_{CheckingAccount}$
post : bal = bal@pre - countUpdates * charge and countUpdates = 0 and $FULLINV_{CheckingAccount}$

Theorem:
Sei $CompSpec = (\langle M, \Delta\rangle, OpSpecs, Invs^e)$ eine Komponentenspezifikation mit formaler Repräsentation $FRep(CompSpec) = (\langle M, \Delta\rangle, OpSpecs, Invs)$ und sei J ein Java Subsystem.

J ist eine korrekte Java Realisierung von $CompSpec$ wenn die folgenden Bedingungen erfüllt sind:

1. *Verantwortlichkeit des Implementierers:*

   (a) J ist konform zu $\Sigma_\triangle$

   (b) Das durch J induzierte $\Sigma_\triangle$-Transitionssystem erfüllt die Beweisverpflichtungen $PObs^{CompSpec}$, die durch $CompSpec$ induziert werden, d.h.

$$\mathcal{T}^J_\Delta \models PObs^{CompSpec}_{op} \text{ für alle } op \in Opns_\triangle$$

2. *Verantwortlichkeit des Benutzers:*

Wenn ein Methodenaufruf $o.op(v_1, \ldots, v_n)$ oder ein Konstruktoraufruf $new\ C(v_1, \ldots, v_n)$ erfolgt, dann gilt:

(a) Die Vorbedingung der Operationsspezifikation der Methode op oder des Konstruktors C is erfüllt (bzgl. des zur Programmierzeit bekannten Klassentyps von o).

(b) Falls op komponenten-privat ist, dann gilt die volle Klasseninvariante für das aufrufende Objekt.

(c) Falls op komponenten-öffentlich ist, gilt (b) und zusätzlich gelten alle Komponenteninvarianten.

Beweis des Theorems:

Es ist zu zeigen $\mathcal{T}^J_\Delta \in [\![CompSpec]\!]$.

Nach der generellen Voraussetzung über Komponentenspezifikationen ist $[\![COMPINV_M]\!]_{\beta,\sigma_{init},\sigma_{init}} = true$. Also bleibt nach dem Satz über die Charakterisierung von Modellen zu zeigen, dass

für alle $(op : C \times T_1 \times \ldots \times T_n \longrightarrow Void) \in Opns_\Delta$ und

für alle $(\texttt{context}\ C :: op(x_1 : T_1, \ldots, x_n : T_n)\ \texttt{pre} : P\ \texttt{post} : Q) \in OpSpecs$ gilt:

$$\begin{array}{lll} \mathcal{T}^J_\Delta & \models & \texttt{context} \quad C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ & & \quad \texttt{pre} \; : \quad P^{Visibility(op)} \\ & & \quad \texttt{post} : \quad Q^{Visibility(op)} \end{array}$$

und analog für die anderen Arten von Operationen.

Wir betrachten den oben angegebenen Fall einer Methode ohne Rückgabewert.

Außerdem sei o.B.d.A. $Visibility(op) = \sim$.

Wegen Voraussetzung (1) des Theorems gilt:

$$\begin{array}{lll} \mathcal{T}^J_\Delta & \models & \texttt{context} \quad C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ & & \texttt{pre} \; : \quad P \;\texttt{and}\; CLASSINV_\Delta \\ & & \texttt{post} : \quad Q \;\texttt{and}\; FULLINV_C \\ & & \qquad \texttt{and} \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \;\texttt{implies}\; FULLINV_D) \end{array}$$

Es ist zu zeigen, dass daraus folgt:

$$\begin{array}{lll} \mathcal{T}^J_\Delta & \models & \texttt{context} \quad C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ & & \texttt{pre} \; : \quad P \;\texttt{and}\; CLASSINV_\Delta \\ & & \texttt{post} : \quad Q \;\texttt{and}\; CLASSINV_\Delta \end{array}$$

Sei nun $(\sigma^-, o, v_1, \ldots, v_n) \in State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq null$ und gelte $[\![P \;\texttt{and}\; CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma^-} = true$ (mit $\beta(self) = o, \beta(x_i) = v_i$). Folglich gilt:

(a) Es existiert $\sigma \in State_\Delta$, so dass $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta$

(b) $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \bot \notin R^J_\Delta$

(c) Für alle $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta$ ist (mit β wie oben)

$$[\![Q \;\texttt{and}\; FULLINV_C \;\texttt{and} \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \;\texttt{implies}\; FULLINV_D)]\!]_{\beta,\sigma^-,\sigma} = true$$

Es genügt also zu zeigen, dass dann für alle $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta$ gilt:

$$[\![\, Q \text{ and } CLASSINV_\Delta \,]\!]_{\beta,\sigma^-,\sigma} = true \quad \text{(mit } \beta \text{ wie oben)}$$

Sei nun $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta$ mit

$$[\![\, Q \text{ and } FULLINV_C \text{ and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D)]\!]_{\beta,\sigma^-,\sigma} = true$$

$$\text{also } [\![\, FULLINV_C \text{ and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D)]\!]_{\beta,\sigma^-,\sigma} = true$$

Folglich gilt für alle $D \in Class_\Delta$ mit $o \in D_\sigma$ und für alle $(\texttt{context } D \texttt{ inv} : Inv) \in Invs : [\![Inv]\!]_{\beta,\sigma^-,\sigma} = true$ (mit β wie oben).

Es bleibt zu zeigen, dass dann auch alle Klasseninvarianten für alle Objekte $obj \neq o$ erhalten bleiben durch die aktuelle Ausführung des Rumpfes der Methode $void\ op(T_1\ x_1, \ldots, T_n\ x_n)$ der Klasse C (bzw. dass für neue Objekte obj die Klasseninvarianten etabliert sind in σ).

Das heißt, wir wollen zeigen, dass für alle $D \in Class_\Delta$,
für alle $obj \in D_\sigma$ mit $obj \neq o$ und für alle $(\texttt{context}\ D\ \texttt{inv} : Inv) \in Invs$
gilt: $[\![Inv]\!]_{\beta[self \mapsto obj],\sigma^-,\sigma} = true$

Dies ist eine Konsequenz der Annahme $[\![P\ \texttt{and}\ CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma^-} = true$
und des unten bewiesenen Lemmas.

Insgesamt folgt damit (vgl. Lemma in Abschnitt 4.2)
$[\![CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma} = true$ und damit, wegen (c), auch
$[\![Q\ \texttt{and}\ CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma} = true$.

Lemma (Erhaltung von Invarianten)

Es seien die generellen Voraussetzungen für Komponentenspezifikationen und Java Programme sowie die Voraussetzungen (1) und (2) des Theorems erfüllt. Dann gilt für alle $C \in Class_\Delta$, $\sigma^- \in State_\Delta$, und $o \in C_{\sigma^-}$ das Folgende:

Falls

für alle $D \in Class_\Delta$, für (zumindest) alle $obj \in D_{\sigma^-}$ mit $obj \neq o$ und
für alle $(\texttt{context}\ D\ \texttt{inv} : Inv) \in Invs$ gilt $[\![Inv]\!]_{\beta[self \mapsto obj],\sigma^-,\sigma^-} = true$

und falls

ein Aufruf $o.op(v_1, \ldots, v_n)$ oder $new\ C(v_1, \ldots, v_n)$ im Zustand σ^- erfolgt
und ohne Laufzeitfehler mit Zustand σ terminiert

dann gilt

für alle $D \in Class_\Delta$, für (zumindest) alle $obj \in D_{\sigma^-}$ mit $obj \neq o$ und
für alle $(\texttt{context}\ D\ \texttt{inv} : Inv) \in Invs$ gilt: $[\![Inv]\!]_{\beta[self \mapsto obj],\sigma^-,\sigma} = true$

Beweis des Lemmas:

Induktion über die Tiefe n der geschachtelten Methoden- und Konstruktoraufrufe:

Fall 0: $n = 0$.

Wegen der generellen Voraussetzungen für Java Programme kann sich höchstens der Zustand von o geändert haben. Folglich bleiben alle Invarianten für alle Objekte $obj \neq o$ erhalten.

Fall 1: $n > 0$.

Sei $o'.op'(v_1', \ldots, v_{n'}')$ oder $new\ C'(v_1', \ldots, v_{n'}')$ ein geschachtelter Aufruf, der im Rumpf von op oder $C(...)$ ausgeführt wird. Wir nehmen o.B.d.A. an, dass kein weiterer Aufruf im Rumpf von op bzw. $C(...)$ erfolgt. Nach den generellen Voraussetzungen für Java Programme haben alle Objekte $obj \neq o$ ihren Zustand nicht geändert (d.h. sie erfüllen ihre Invarianten), solange die Ausführung des Rumpfes den geschachtelten Aufruf noch nicht erreicht hat.

Im Folgenden betrachten wir nun den geschachtelten Aufruf $o'.op'(v_1', \ldots, v_{n'}')$ (der Konstruktoraufruf wird ähnlich behandelt).

Fall 1.1: $o' = o$.
Nach Induktionsvoraussetzung erfüllen alle Objekte $obj \neq o' = o$ ihre Invarianten nach Ausführung des geschachtelten Methodenaufrufs und folglich, nach den generellen Voraussetzungen für Java Programme und wegen der Lokalität von Klasseninvarianten, auch nach Ausführung des übergeordneten Aufrufs.

Fall 1.2: $o' \neq o$.
Nach den Bedingungen (2b) und (2c) erfüllen alle existierenden Objekte (auch o) ihre Invarianten bevor der geschachtelte Aufruf ausgeführt wird. Außerdem ist nach Bedingung (2a) und wegen der Verhaltensverträglichkeit bzgl. Subtypen die Vorbedingung der Operationsspezifikation von op' or $C'(...)$ erfüllt bevor der geschachtelte Aufruf ausgeführt wird. Somit ist die Vorbedingung der korrespondierenden Beweisverpflichtung erfüllt. Nach Bedingung (1b) erfüllt o' seine Invarianten nach Ausführung des geschachtelten Aufrufs. Nach Induktionsvoraussetzung erfüllen auch alle anderen Objekte ihre Invarianten nach Ausführung des geschachtelten Aufrufs. Mithin erfüllen nach den generellen Voraussetzungen für Java Programme und wegen der Lokalität von Klasseninvarianten alle existierenden Objekte ihre Invarianten nach Ausführung des übergeordneten Aufrufs.

5.3 Zusammenfassung

Sei $CompSpec = (\langle M, \Delta \rangle, OpSpecs, Invs^e)$ ein Komponentenspezifikation.

- Ein Java Subsystem J ist eine korrekte Java Realisierung von $CompSpec$ falls
  - J konform zu $\Sigma_\triangle$ ist und
  - das (durch J induzierte) $\Sigma_\triangle$-Transitionssystem $\mathcal{T}_\Delta^J$ eine korrekte Realisierung von $CompSpec$ ist.
- Eine Komponentenspezifikation induziert eine Menge von Beweisverpflichtungen.
- Die Erfüllung der Beweisverpflichtungen (durch das Transitionssystem $\mathcal{T}_\Delta^J$) garantiert die Korrektheit der Java Realisierung falls
  - die generellen Voraussetzungen (z.B. keine öffentlichen Attribute, nur lokale Klasseninvarianten) und
  - die Bedingungen gemäß der Verantwortlichkeit des Implementierers und
  - die Bedingungen gemäß der Verantwortlichkeit des Benutzers

  erfüllt sind.