
Vorlesung „Methoden des Software Engineering“

Block C „Formale Methoden“

Spezifikation objektbasierter Systeme mit OCL

Martin Wirsing

Einheit C.2, 30.11.2006

Ziele

- Objektorientierte Systeme mit Hilfe von Invarianten und Zusicherungen spezifizieren lernen
- Verfeinerung von Spezifikationen und Korrektheit von Implementierungen nachweisen lernen

Eigenschaften von Transitionssystemen

Invarianten

Durch Invarianten wird der Zustandsraum eingeschränkt.

Zusicherungen

Durch Vor- und Nachbedingungen einer Operation m spezifiziert man die Menge der erlaubten Zustandsübergänge für die Implementierung von m .

Invarianten und Zusicherungen spielen die Rolle eines Vertrags zwischen der Implementierung und der Benutzung einer Menge von Klassen.

OCL

OCL (Object Constraint Language)

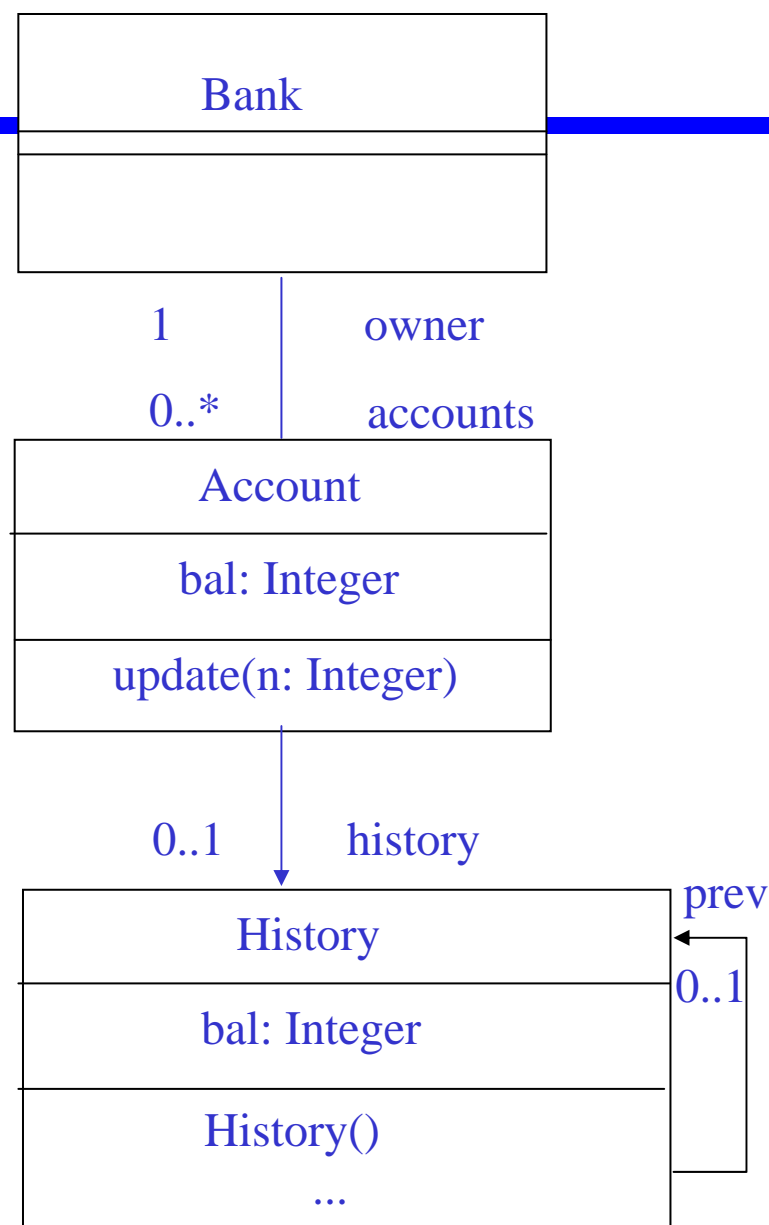
- Ursprünglich entwickelt bei IBM (1995) aufgrund von Erfahrungen mit der (formalen) OO Entwicklungsmethode Syntropy
- Sprache zur formalen Spezifikation von Constraints für UML
- Deklarative streng getypte Sprache zur Formulierung boolescher Ausdrücke
≅ Logik erster Stufe mit Quantoren über endlichen Mengen

Anwendungen von OCL

Spezifikation von

- Invarianten und Vor/Nachbedingungen
- Bedingungen in Statecharts
- Constraints des UML Metamodells

Beispiel: Bank



context Bank

inv: accounts \rightarrow forAll(a | a.owner = self)

context Account

inv: bal ≥ 0

context Account: : update(n)

pre: bal + n ≥ 0

post: bal = bal@pre + n and
 history.oclIsNew() and
 history.bal = bal@pre and
 history.prev=history@pre

Typen und Operationen von OCL

OCL-Syntax (vereinfacht)

Integer, Real, Boolean, String

C, Set(C), Sequence(C), ...

x.bal

=

not, and, implies,...

set \rightarrow forAll(x | P)

set \rightarrow select(x | P)

C.allInstances()

Zusätzlich, in Nachbedingungen

x.bal@pre

x.oclIsNew()

Grunddatentypen

Klassen C und Kollektionstypen

Instanzvariable

Gleichheit

Junktoren

beschränkte Quantifizierung

$\cong \forall x \in \text{set}. P$

$\cong \{x \in \text{set} \mid P\}$

„alle existierenden Objekte der Klasse C“

„vorheriger“ Wert von x.bal

x bezeichnet neues Objekt

Semantik von OCL: OCL-Algebra

Wir definieren eine **Algebra** \mathcal{O} zur Interpretation der Datentypen und Operationen von OCL

Interpretation der Sorten

$$\begin{aligned} \mathcal{O}_{\text{Integer}} &= \mathbb{Z} & \mathcal{O}_{\text{Boolean}} &= \mathbb{B} = \{\text{true}, \text{false}\} \\ \mathcal{O}_{\text{Real}} &= \mathbb{R} & \mathcal{O}_{\text{String}} &= A^* \end{aligned}$$

wobei A ein Alphabet

$$\mathcal{O}_C = \text{Old}_C \cup \{\text{null}\}$$

wobei Old_C die (abzählbare) Menge der Objektidentifikatoren der Klasse C ist

$$\mathcal{O}_{\text{Set}(T)} = F(\mathcal{O}_T)$$

„endliche Mengen mit Elementen aus \mathcal{O}_T “

$$\mathcal{O}_{\text{Sequence}(T)} = \mathcal{O}_T^*$$

„endliche Folgen mit Elementen aus \mathcal{O}_T “

Die Operationen werden definiert auf Typen mit \perp -Element:

$$(\mathcal{O}_T)_\perp = \mathcal{O}_T \cup \{\perp\}$$

für jeden Typ T

Semantik von OCL: OCL-Algebra

Interpretation der Booleschen Operationen

$$\text{not}^\theta: B_\perp \rightarrow B_\perp$$

$$\text{not}^\theta(b) = \begin{cases} \text{false} & \text{falls } b=\text{true} \\ \text{true} & \text{falls } b=\text{false} \\ \perp & \text{falls } b=\perp \end{cases}$$

$$\text{and}^\theta: B_\perp \times B_\perp \rightarrow B_\perp$$

$$b1 \text{ and}^\theta b2 = \begin{cases} \text{true} & \text{falls } b1=\text{true} \text{ und } b2=\text{true} \\ \text{false} & \text{falls } b1=\text{false} \text{ oder } b2=\text{false} \\ \perp & \text{sonst} \end{cases}$$

Semantik von OCL: OCL-Algebra

Eine Operation $op: T_1 \times \dots \times T_n \rightarrow T$ heißt **strikt**, wenn gilt:
 $op^{\theta}(x_1, \dots, x_n) = \perp$ falls $x_1 = \perp$ oder ... oder $x_n = \perp$

Interpretation der Operationen

Arithmetische Operationen und Mengenoperationen werden standardmäßig als strikt interpretiert und haben die übliche Bedeutung, z.B.

$$x =^{\theta} y = \begin{cases} \text{true} & \text{falls } x = y \text{ und } x \neq \perp \text{ und } y \neq \perp \\ \text{false} & \text{falls } x \neq y \text{ und } x \neq \perp \text{ und } y \neq \perp \\ \perp & \text{sonst} \end{cases}$$

$$s \rightarrow \text{includes}^{\theta}(x) = \begin{cases} \text{true} & \text{falls } x \in s \text{ und } x \neq \perp \text{ und } s \neq \perp \\ \text{false} & \text{falls } x \notin s \text{ und } x \neq \perp \text{ und } s \neq \perp \\ \perp & \text{sonst} \end{cases}$$

$$s \rightarrow \text{including}^{\theta}(x) = \begin{cases} s \cup \{x\} & \text{falls } x \neq \perp \text{ und } s \neq \perp \\ \perp & \text{sonst} \end{cases}$$

Semantik von OCL: Systemzustand

Systemzustand

Der Systemzustand σ eines Klassendiagramms Δ ist gegeben durch

- die Menge C_σ der existierenden Objekte (für jede Klasse C aus Δ)
- ein Objektdiagramm definiert durch die Belegung σ_{Val} der Instanzvariablen, so dass σ_{Val} typkorrekt ist, d.h.

für jedes $o \in \text{Old}_C$ und Attribut $_a: C \rightarrow T: \sigma_{\text{Val}}(o.a) \in T^\emptyset$

Semantik von OCL: Interpretation der Ausdrücke

Interpretation der OCL-Ausdrücke

- OCL-Ausdrücke werden über der Algebra \mathcal{O} interpretiert und machen Aussagen über
einen Systemzustand σ und dessen Vorgängerzustand $\sigma@pre$
- Wir definieren die Semantik $[[e]]$ eines OCL-Ausdrucks e induktiv über die syntaktische Struktur des Ausdrucks.
- Die Interpretationsfunktion $[[_]]$ hängt ab von
Zustand σ , Vorzustand $\sigma@pre$ und einer Belegung β der Variablen von e .

Semantik von OCL: Interpretation der Ausdrücke

$[[x]]_{\sigma@pre, \sigma, \beta} = \beta(x)$ für jede Variable x

$[[e.a]]_{\sigma@pre, \sigma, \beta} = \sigma([[e]]_{\sigma@pre, \sigma, \beta}.a)$ für jedes Attribut a

$[[e.a@pre]]_{\sigma@pre, \sigma, \beta} = \sigma@pre([[e]]_{\sigma@pre, \sigma, \beta}.a)$ für jedes Attribut a

$[[op(e_1, \dots, e_n)]]_{\sigma@pre, \sigma, \beta} = op^{\theta}([[e_1]]_{\sigma@pre, \sigma, \beta}, \dots, [[e_n]]_{\sigma@pre, \sigma, \beta})$
für jede OCL-Operation, z.B.

$[[e_1 = e_2]]_{\sigma@pre, \sigma, \beta} = [[e_1]]_{\sigma@pre, \sigma, \beta} =^{\theta} [[e_2]]_{\sigma@pre, \sigma, \beta}$

$[[C.allInstances()]]_{\sigma@pre, \sigma, \beta} = C_{\sigma}$ für alle Klassen C aus Δ

$[[e.ocllsNew()]]_{\sigma@pre, \sigma, \beta} = \begin{cases} \text{true} & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} \in C_{\sigma} \text{ und } [[e]]_{\sigma@pre, \sigma, \beta} \notin C_{\sigma@pre} \\ \perp & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} = \perp \\ \text{false} & \text{sonst} \end{cases}$

Vor-/Nachbedingungen

Vor-/Nachbedingungen schränken das mögliche Verhalten von Operationen ein:

- Die Vorbedingung muss erfüllt sein, wenn die Operation aufgerufen wird.
- Die Nachbedingung muss nach Beendigung der Ausführung der Operation erfüllt sein.

Zusicherungsspezifikation

Sei $SP_m = \text{context } C :: m(x_1: T_1, \dots, x_n: T_n)$

pre: PRE post: POST

- SP_m heißt **Zusicherungsspezifikation** (Operation specification, Vor-/Nachbedingungsspez.) und definiert eine Transitionsrelation.

- Die Semantik von SP_m ist durch die folgende Transitionsrelation gegeben:

$$\sigma@pre \xrightarrow{o.m(v_1, \dots, v_n)} \sigma \in [[SP_m]] \text{ gdw}$$

$$[[PRE]]_{\sigma@pre, \sigma@pre, \beta} = \text{true} \Rightarrow [[POST]]_{\sigma@pre, \sigma, \beta} = \text{true}$$

für alle β mit $\beta(\text{self}) = o$, $\beta(x_1) = v_1$, ..., $\beta(x_n) = v_n$

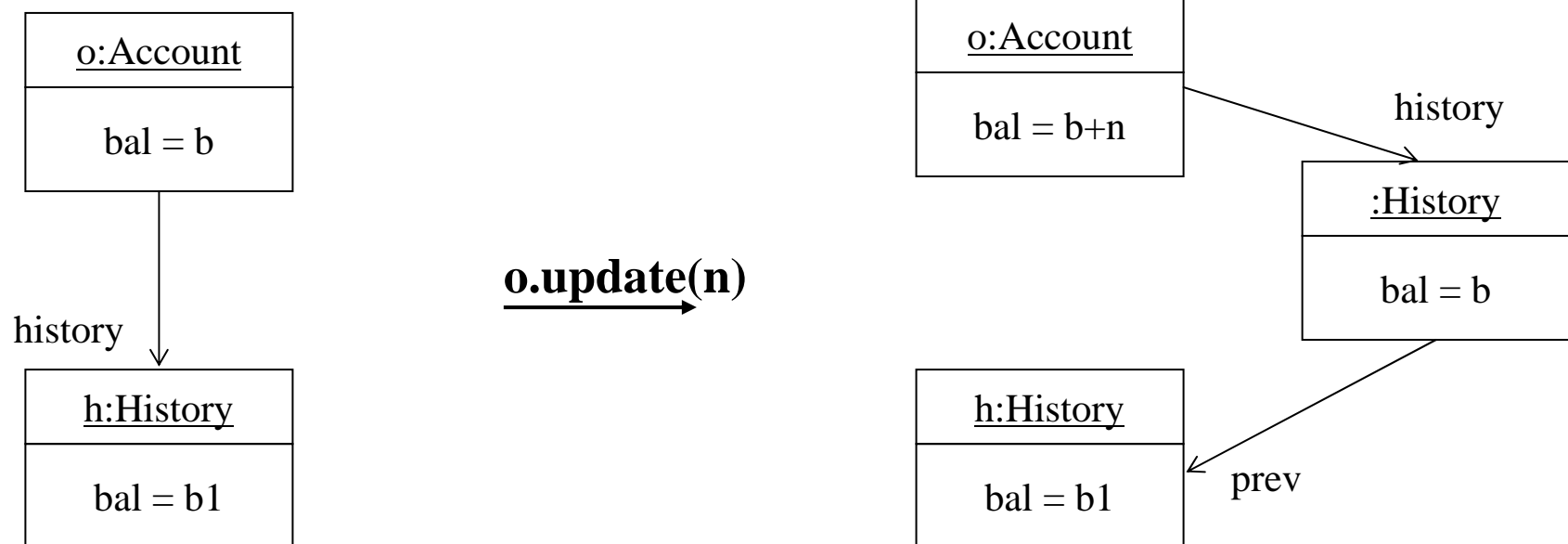
Transitionssystem für Bank (objektorientiert)

context Account :: update(n)

pre: $bal + n \geq 0$

post: $bal = bal@pre + n$ and $history.ocIsNew()$ and
 $history.bal = bal@pre$ and $history.prev = history@pre$

definiert folgendes Transitionssystem für alle b, n mit $b+n \geq 0$:



Vor-/Nachbedingungen als Vertrag

Eine Vor-/Nachbedingungsspezifikation kann als Vertrag dienen zwischen

- einem Kunden, der die Operation benutzt und
- einem Programmierer, der die Operation realisiert.

Verantwortung des Kunden

Der Kunde ruft die Operation nur auf, wenn die Vorbedingung erfüllt ist

Verantwortung des Programmierers

Der Programmierer garantiert, dass

nach Ausführung der Operation die Nachbedingung gilt,
falls die Operation in einem Zustand aufgerufen wurde,
der die Vorbedingung erfüllt.

Vor-/Nachbedingungen als Vertrag

Bemerkungen:

- Der Vertrag sagt nichts aus über Situationen, in denen die Vorbedingung nicht erfüllt ist.
- Bei einer query-Operation muss der Programmierer auch garantieren, dass der Systemzustand unverändert bleibt.
- Bei einem Konstruktor muss der Programmierer auch garantieren, dass ein neues Objekt erzeugt wird.

Verfeinerung

- Eine Zusicherungsspezifikation SPK heißt (**Operations–**) **Verfeinerung** einer Zusicherungsspezifikation SPA, falls
 - SPK alle Eingaben akzeptiert, die auch SPA akzeptiert und
 - SPK determinierter ist als SPA in Bezug auf Eingaben von SPA

- **Formal**

Sei $SPA = \text{context } C: m(x:T) \text{ pre: } PRE_A \text{ post: } POST_A$

und $SPK = \text{context } C: m(x:T) \text{ pre: } PRE_K \text{ post: } POST_K$

Dann ist **SPK (Operations–) Verfeinerung von SPA**, falls gilt:

1. $PRE_A \Rightarrow PRE_K$ und
2. $PRE_A @ \text{pre} \wedge POST_K \Rightarrow POST_A$

Verfeinerung: Beispiel Account

- Spezifikation ohne Anforderung an die Kontoentwicklung

```

SPAupdate = context Account::update(n)
           pre: bal + n ≥ 0
           post: bal = bal@pre + n

```

- Spezifikation mit Anforderungen an die Kontoentwicklung

```

SPKupdate = context Account::update(n)
           pre: bal + n ≥ 0
           post: bal = bal@pre + n          and
                history.oclIsNew()        and
                history.bal = bal@pre
                history.prev=history@pre

```

- SPKC_{update} ist Operationsverfeinerung von SPA_{update}
 denn $PRE_{SPA} \Rightarrow PRE_{SPK}$ wg. $PRE_{SPA} = PRE_{SPK}$
 und $PRE_A@pre \wedge POST_K \Rightarrow POST_A$ wg. $POST_K \Rightarrow POST_A$

Verfeinerung: Bemerkungen

- Zur Verfeinerung eines Klassendiagramms müssen natürlich **alle Methoden und Konstruktoren verfeinert** werden.
- Bei der **Operationsverfeinerung** gibt es **keine Änderung der Signatur**; insbesondere ändert sich nichts an den Elementen des Systemzustands.
- Bei einem **Wechsel der Datenstruktur** haben der abstrakte und der konkrete Datentyp unterschiedliche Signaturen. Es wird eine (Daten-)Verfeinerungsrelation zwischen den Werten des abstrakten und des konkreten Datentyps definiert; die Bedingungen der Operationsverfeinerung werden modulo Datenverfeinerung und Signaturwechsel formuliert.
(Für Genaueres siehe Grundlagen der Systementwicklung)

Invarianten

- Eine **Invariante** ist eine Bedingung, die die Menge der möglichen Systemzustände einschränkt.
- Eine Invariante sollte deshalb **vor und nach jedem Aufruf einer (öffentlichen) Methode** gelten.

Beispiele:

context Account

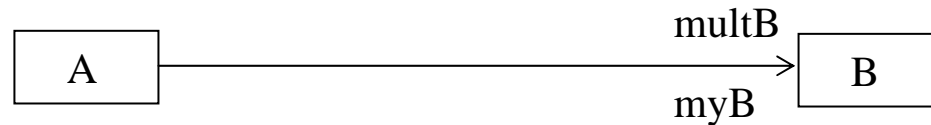
inv: bal \geq 0

context Bank

inv: accounts \rightarrow forAll(a | a.owner = self)

Implizite Invarianten durch Assoziationen

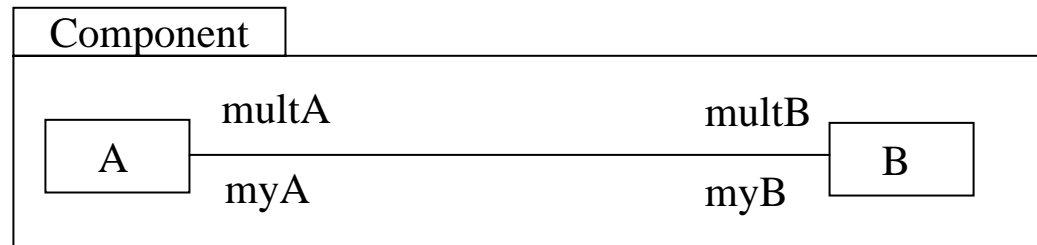
Die Assoziationen eines Klassendiagramms induzieren implizite Invarianten:



- $\text{multB} = 0..1$ ist vollständig formalisiert durch $_.\text{myB}: A \rightarrow B$
- $\text{multB} = 1$ induziert die Klasseninvariante
context A $\text{inv: myB} \langle \rangle \text{ null}$
- $\text{multB} = *$ ist vollständig formalisiert durch $_.\text{myB}: A \rightarrow \text{Set}(B)$
- $\text{multB} = 1..*$ induziert die Klasseninvariante
context A $\text{inv: myB} \rightarrow \text{exists}(b \mid b \langle \rangle \text{ null})$

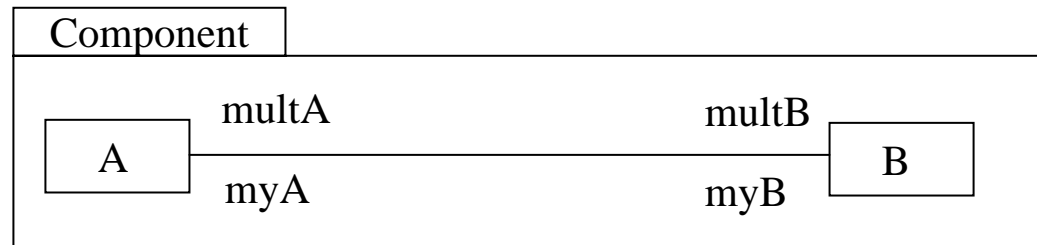
Implizite Invarianten durch Assoziationen

Bidirektionale Assoziationen:



- $\text{multA} = \text{multB} = 0..1$ induziert die Komponenteninvariante
 context Component
 inv: $A.\text{allInstances}() \rightarrow \text{forall}(a \mid a.\text{myB} \neq \text{null} \text{ implies } a.\text{myB}.\text{myA} = a)$ and
 $B.\text{allInstances}() \rightarrow \text{forall}(b \mid b.\text{myA} \neq \text{null} \text{ implies } b.\text{myA}.\text{myB} = b)$
- $\text{multA} = \text{multB} = 1$ induziert die Komponenteninvariante
 context Component
 inv: $A.\text{allInstances}() \rightarrow \text{forall}(a \mid a.\text{myB}.\text{myA} = a)$ and
 $B.\text{allInstances}() \rightarrow \text{forall}(b \mid b.\text{myA}.\text{myB} = b)$

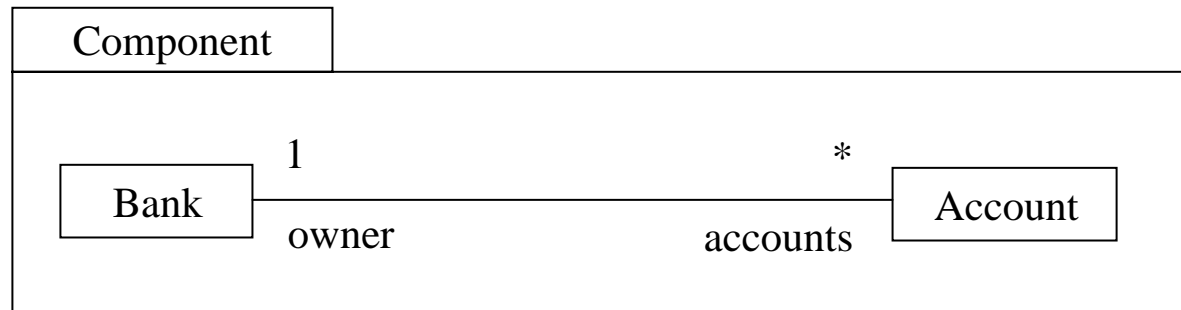
Implizite Invarianten durch Assoziationen



- $\text{multA} = \text{multB} = *$ induziert die Komponenteninvariante
 context Component
 inv: $A.\text{allInstances}() \rightarrow \text{forall}(a \mid$
 $a.\text{myB} \rightarrow \text{forall}(b \mid b \neq \text{null} \text{ implies } b.\text{myA} \rightarrow \text{includes}(a)) \text{ and}$
 $B.\text{allInstances}() \rightarrow \text{forall}(b \mid$
 $b.\text{myA} \rightarrow \text{forall}(a \mid a \neq \text{null} \text{ implies } a.\text{myB} \rightarrow \text{includes}(b))$
- Alle anderen Fälle werden analog behandelt

Implizite Invarianten durch Assoziationen

Beispiel:



- **Komponenteninvariante**

context Component

inv: Bank.allInstances() →forall(b |

b.accounts →forall(a | a<>null implies a.owner =b) and

Account.allInstances() →forall(a | a.owner.accounts→includes(a))

Zusicherungsspezifikation und Invarianten

- Sei INV die Konjunktion aller Invarianten eines Klassendiagramms Δ
 und $SP_m =$ context $C :: m(x_1: T_1, \dots, x_n: T_n)$
 pre: PRE post: $POST$
 eine Zusicherungsspezifikation von Δ .
- Die **von INV induzierte Zusicherungsspezifikation SP_m^{INV}** lautet:
 $SP_m^{INV} =$ context $C :: m(x_1: T_1, \dots, x_n: T_n)$
 pre: **PRE and INV** post: **$POST$ and INV**
- Die Spezifikation ist **konsistent**, wenn für jeden Zustand $\sigma@pre$ und für jede Belegung β der formalen Parameter gilt:
 $[[PRE \text{ and } INV]]_{\sigma@pre, \sigma@pre, \beta} = true \Rightarrow$
 es gibt einen Zustand σ mit $[[POST \text{ and } INV]]_{\sigma@pre, \sigma, \beta} = true$

Zusicherungsspezifikation und Invarianten: Beispiel

- Die BankAccount-Spezifikation

context Account

inv: $bal \geq 0$

context Account::update(n)

pre: $bal + n \geq 0$

post: $bal = bal@pre + n$

ist konsistent.

Korrektheit der Implementierung

- Eine **Implementierung m** heißt **korrekt** bzgl. einer konsistenten Spezifikation SP^{INV}_m , wenn die Implementierung eine Operationsverfeinerung von SP^{INV}_m ist.

- In anderen Worten:

$$(\text{PRE and INV}) \Rightarrow \text{PRE}(\underline{m}) \quad \text{und}$$

$$(\text{PRE}@pre and INV@pre) \wedge \text{POST}(\underline{m}) \Rightarrow \text{POST and INV}$$

wobei \underline{m} die Spezifikation des Transitionssystems von m bezeichnet.

Korrektheit der Implementierung: Beispiel

- Die folgende BankAccount-Implementierung

```
class Account
{   int bal;   History history;   Bank owner;
    ...
    void update(int n)
    {   History h1 = new History();
        h1.bal = bal;   h1.prev = history;
        history = h1;
        bal = bal+n;
    }
}
```

besitzt die Spezifikation update =

pre: true

post: $bal = bal@pre + n$ and $history.oclIsNew()$ and
 $history.bal = bal@pre$ and $history.prev=history@pre$

Korrektheit der Implementierung: Beispiel

Die Implementierung ist korrekt bzgl. der Spezifikation

context Account	context Account::update(n)
inv: bal ≥ 0	pre: bal + n ≥ 0
	post: bal = bal@pre + n

Denn es gilt:

$$\begin{aligned}
 & (\text{PRE and INV}) \Rightarrow \text{PRE}(\underline{\text{update}}) (= \text{true}) \quad \text{und} \\
 & (\text{PRE@pre and INV@pre}) \wedge \text{POST}(\underline{\text{update}}) \\
 & \equiv \text{bal@pre} + n \geq 0 \text{ and } \text{bal@pre} \geq 0 \text{ and } \text{POST}(\underline{\text{update}}) \\
 & \Rightarrow \text{bal} = \text{bal@pre} + n \text{ and } \text{bal@pre} + n \geq 0 \\
 & \Rightarrow \text{POST and INV}
 \end{aligned}$$

Zusicherungsspezifikation und Invarianten

Bemerkung

- Die Forderung, dass jede Methode alle Invarianten erhält, ist im Allgemeinen zu stark.
- Man muss dies nur für Methoden fordern, die außerhalb einer Komponente sichtbar sind (public visibility).
- Methoden, auf die nur innerhalb einer Komponente von anderen Klassen zugegriffen werden kann, müssen nur die Klasseninvarianten erhalten (default visibility in Java).
- Private Methoden einer Klasse müssen überhaupt keine Invarianten erhalten.

(Für eine genaue Untersuchung siehe Hennicker: FOOSE).

Zusammenfassung

OCL ist eine Spezifikationssprache zur Formulierung von Invarianten und Vor- und Nachbedingungen

Invarianten

Durch Invarianten wird der Zustandsraum eingeschränkt.

Zusicherungen

Durch Vor- und Nachbedingungen einer Operation m spezifiziert man

- a) die Bedingung unter der die Operation m aufgerufen werden darf,
- b) die Menge der erlaubten Zustandsübergänge für die Implementierung.

Verfeinerung

Durch (Operations-)Verfeinerung zeigt man die Korrektheit von Implementierungen.

Literatur

- **Wirsing: Grundlagen der Systementwicklung, Kap. 8**
- **Hennicker: Formale objektorientierte Software-Entwicklung,
Kap. 2-5**