

Martin Glinz

# Requirements Engineering I

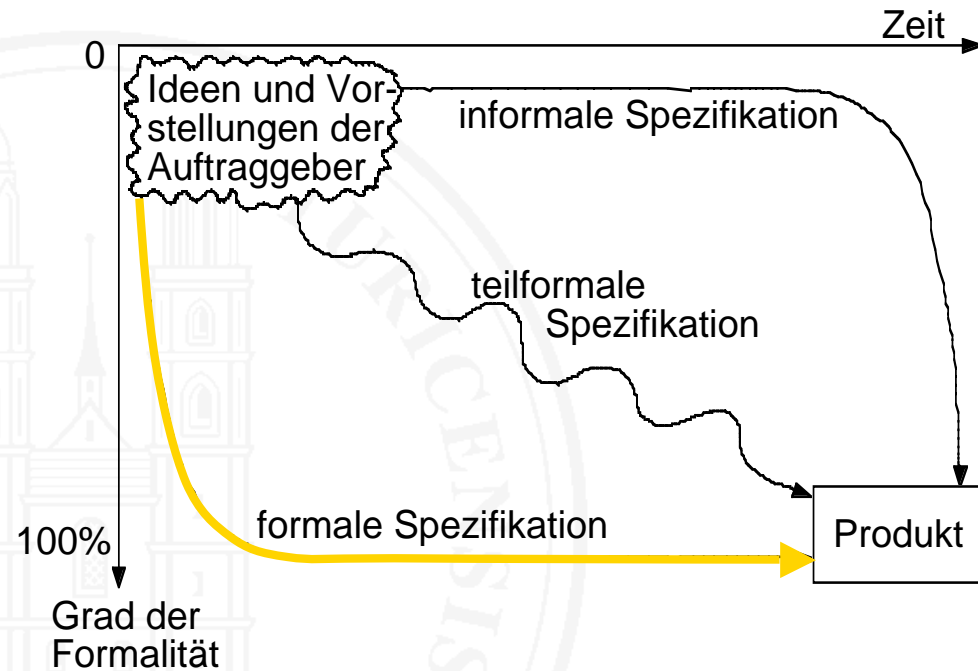
Kapitel 10

## Formale Spezifikation



Universität Zürich  
Institut für Informatik

# 10.1 Grundlagen



## Die Vision

- Das Problem **analysieren**
- Anforderungen **formal spezifizieren**
- Implementierung durch **korrektheitserhaltende Transformationen**
- **Pflege** der **Anforderungsspezifikation**, nicht des Codes

# Mittel und Formen

---

## Was bedeutet „formal“?

- **Formaler Kalkül**, d.h. Verwendung einer Spezifikationsprache mit
  - formal definierter **Syntax**
  - und
  - formal definierter **Semantik**
- Primär zur Spezifikation funktionaler Anforderungen

## Welche Arten?

- Rein deskriptiv, zum Beispiel **Algebraische Spezifikation**
- Rein konstruktiv, zum Beispiel **Petrinetze**
- Modellbasierte Mischformen, zum Beispiel **OCL**, **VDM** und **Z**

## 10.2 Algebraische Spezifikation

---

- Deskriptive, formale Methode
- Primär für die formale Spezifikation komplexer Datentypen
- Syntax durch Signaturen (Definitions- und Wertebereiche) der Operationen
- Semantik durch Axiome (Ausdrücke, die immer wahr sein müssen)
- Axiome beschreiben im Wesentlichen Invarianten unter der Anwendung von Funktionen
- Eines der ältesten formalen Spezifikationsverfahren (seit ca. 1977)
- Schwer lesbar
- Unter- und Überspezifikation schwer erkennbar
- Hat den Sprung aus der Forschung in die Praxis nie geschafft

# Algebraische Spezifikation: Beispiel

---

## Spezifikation eines Kellers (Stack)

Sei `bool` der Datentyp mit dem Wertebereich `{false, true}` und der Booleschen Algebra als Operationen. Sei ferner `elem` der Datentyp für die Datenelemente, die im spezifizierten Keller zu speichern sind.

TYPE Stack

FUNCTIONS

```
new:   ()           → Stack; -- neuen (leeren) Keller anlegen
push:  (Stack, elem) → Stack; -- Element hinzufügen
pop:   Stack        → Stack; -- zuletzt hinzugefügtes Element entfernen
top:   Stack        → elem;  -- liefert zuletzt hinzugefügtes Element
empty: Stack        → bool;  -- wahr, wenn Keller kein Element enthält
full:  Stack        → bool;  -- wahr, wenn Keller voll ist
```

# Algebraische Spezifikation: Beispiel – 2

---

## AXIOMS

$\forall s \in \text{Stack}, e \in \text{elem}$

(1)  $\neg \text{full}(s) \rightarrow \text{pop}(\text{push}(s,e)) = s$

-- Pop hebt den Effekt von Push auf

(2)  $\neg \text{full}(s) \rightarrow \text{top}(\text{push}(s,e)) = e$

-- Top liefert das zuletzt gespeicherte Element

(3)  $\text{empty}(\text{new}) = \text{true}$

-- ein neuer Keller ist leer

(4)  $\neg \text{full}(s) \rightarrow \text{empty}(\text{push}(s,e)) = \text{false}$

-- nach Push ist ein Keller nicht mehr leer

(5)  $\text{full}(\text{new}) = \text{false}$

-- ein neuer Keller ist nicht voll

(6)  $\neg \text{empty}(s) \rightarrow \text{full}(\text{pop}(s)) = \text{false}$

-- nach Pop ist ein Keller niemals voll

## 10.3 Modellbasierte formale Spezifikation

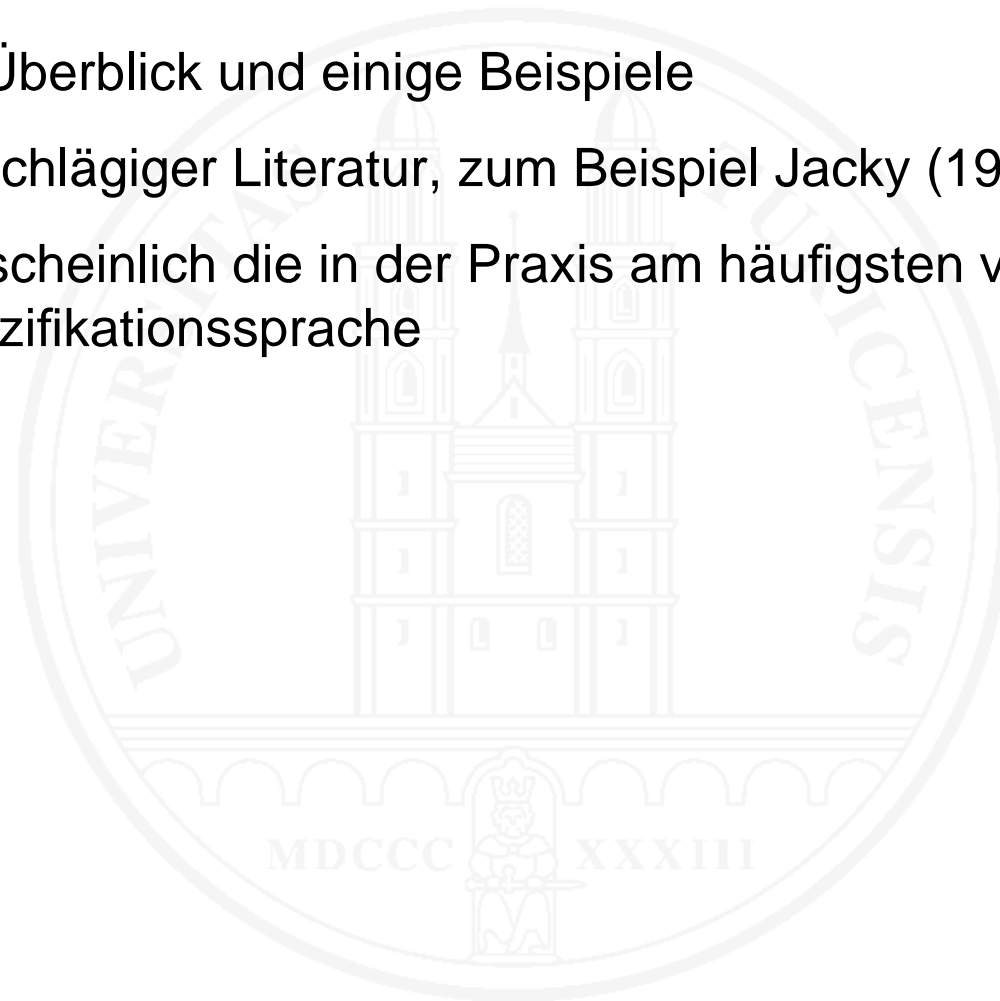
---

- Mathematisches Modell des **Systemzustands** und seiner **Veränderungen**
- Basierend auf **Mengen**, **Relationen** und **logischen Ausdrücken**
- Beschreibung von
  - Grundmengen
  - Zusammenhängen zwischen Mengen (Relationen, Funktionen)
  - Invarianten (Prädikate)
  - Zustandsveränderungen (Relationen, Funktionen)
  - Zusicherungen für Zustände
- Bekannte Vertreter:
  - **VDM** (Vienna Development Method, Björner und Jones 1978)
  - **Z** (Spivey 1992)
  - **OCL** (ab 1997; OMG 2006)

## 10.3.1 Die Spezifikationsprache Z

---

- Nur grober Überblick und einige Beispiele
- Mehr in einschlägiger Literatur, zum Beispiel Jacky (1997)
- Heute wahrscheinlich die in der Praxis am häufigsten verwendete formale Spezifikationsprache





# Die Grundelemente von Z

---

- Z basiert auf Mengen
- Eine Spezifikation besteht aus Mengen, Typen, Axiomen und Schemata
- Typen sind Grundmengen:  $[Name] \quad [Datum] \quad IN$
- Mengen haben einen Typ:  $Person: P \quad Name \quad Zähler: IN$
- Axiome definieren globale Variablen und deren (invariante) Eigenschaften:

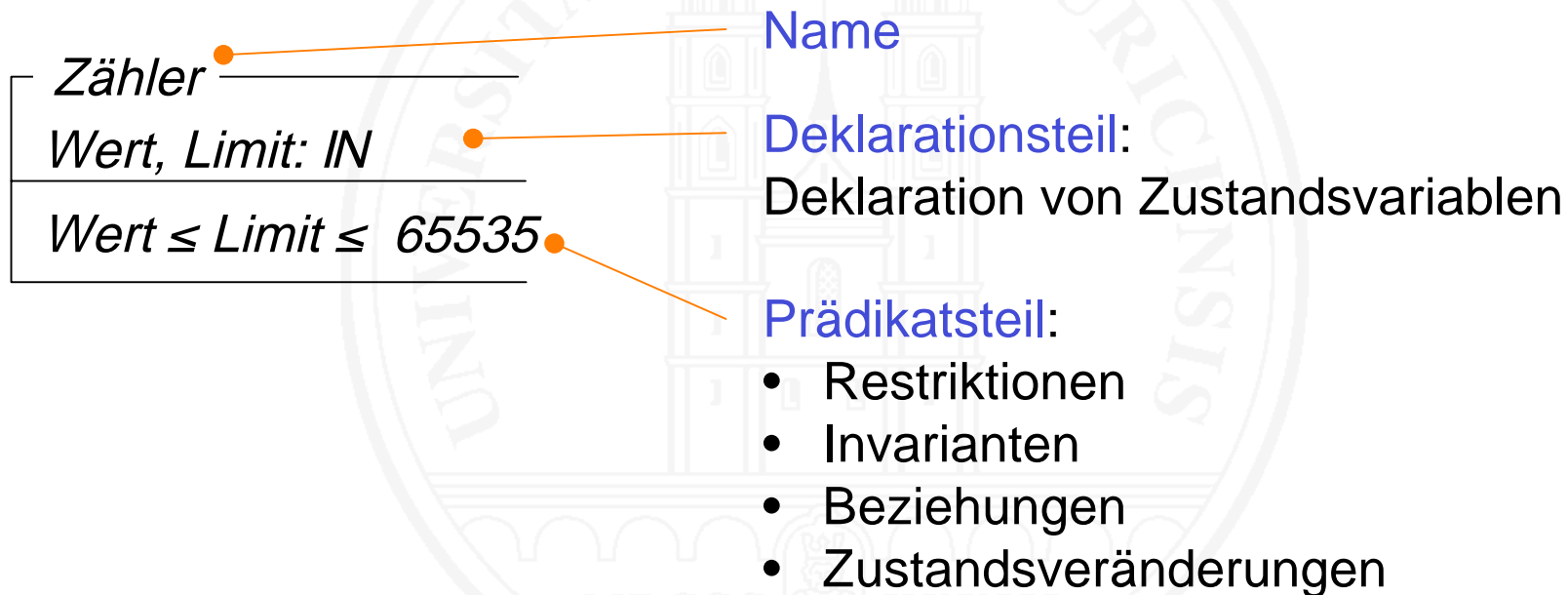
$string: seq \text{ CHAR}$	●	Deklaration
$\#string \leq 64$	●	Invariante

- $IN$  Menge der natürlichen Zahlen
- $P M$  Menge aller Teilmengen von  $M$
- $seq$  Sequenz von Elementen
- $\#M$  Anzahl Elemente der Menge  $M$

# Die Grundelemente von Z – 2

---

- **Schemata** gliedern eine Z-Spezifikation und bilden einen eigenen Namensraum:



# Relationen, Funktionen und Operationen

- **Relationen** und **Funktionen** sind Mengen geordneter Tupel:

*Bestellung:  $\mathcal{P}$  (Teil x Lieferant x Datum)*

Eine Teilmenge aller geordneten Tripel  $(t, l, d)$  mit  $t \in \text{Teil}$ ,  $l \in \text{Lieferant}$  und  $d \in \text{Datum}$

*Geburtstag:  $\text{Person} \rightarrow \text{Datum}$*

eine Funktion, die jeder Person genau ein Datum als Geburtstag zuordnet

- Zustandsveränderung durch **Operationen**:

*Inkrementieren* ———

$\Delta$  Zähler

$\text{Wert} < \text{Limit}$

$\text{Wert}' = \text{Wert} + 1$

$\text{Limit}' = \text{Limit}$

$\Delta S$  Die in S definierten Mengen werden verändert

$M'$  Zustand der Menge M nach Ausführung der Operation

Mathematische Gleichheit; keine Zuweisung!

# Beispiel: Spezifikation eines Bibliothekssystems

---

Die Bibliothek hat einen Bestand an Büchern und eine Menge von Personen als Benutzer.

Bücher aus dem Bestand können an Benutzer ausgeliehen sein.

*Bibliothek*

*Bestand:  $\mathcal{P}$  Buch*

*Benutzer:  $\mathcal{P}$  Person*

*ausgeliehen: Buch  $\rightarrow$  Person*

***dom*** *ausgeliehen*  $\subseteq$  Bestand

***ran*** *ausgeliehen*  $\subseteq$  Benutzer

$\rightarrow$  partielle Funktion  
***dom*** Definitionsbereich...  
***ran*** Wertebereich...  
...einer Relation

# Spezifikation eines Bibliothekssystems – 2

---

Nicht ausgeliehene Bücher aus dem Bestand können von Benutzern ausgeliehen werden.

*Ausleihen*

$\Delta$  *Bibliothek*

*auszuleihendesBuch?: Buch*

*Ausleiher?: Person*

*auszuleihendesBuch? ∈ Bestand \ dom ausgeliehen*

*Ausleiher? ∈ Benutzer*

*ausgeliehen' = ausgeliehen ∪ {(auszuleihendesBuch?, Ausleiher?)}*

*Bestand' = Bestand*

*Benutzer' = Benutzer*

$x?$	$x$ ist Eingabevariable
$a \in X$	$a$ ist Element der Menge $X$
$\setminus$	Mengendifferenzoperator
$\cup$	Vereinigungsoperator

# Spezifikation eines Bibliothekssystems – 3

---

Es kann erfragt werden, ob ein bestimmtes Buch ausgeliehen ist.

*AnfrageObAusleihbar*

$\exists$  *Bibliothek*

*angefragtesBuch?*: *Buch*

*istAusleihbar!*: {*ja*, *nein*}

*angefragtesBuch?*  $\in$  *Bestand*

*istAusleihbar!* = **if** *angefragtesBuch?*  $\notin$  **dom** *ausgeliehen*  
**then ja else nein**

*x!* x ist Ausgabevariable

# Aufgabe 10.1

Zu erstellen ist ein System zur Erteilung und Verwaltung von Autorisierungen für Dokumente.

Gegeben seien folgende Mengen:

*Autorisierung*

*Bestand:  $\mathcal{P}$  Dokument*

*Mitarbeiter:  $\mathcal{P}$  Person*

*autorisiert:  $\mathcal{P}$  (Dokument  $\times$  Person)*

*gesperrt:  $\mathcal{P}$  (Dokument  $\times$  Datum)*

Spezifizieren Sie eine Operation, welche einer Person die Berechtigung für den Zugang zu einem nicht gesperrten Dokument gibt, in einem Z-Schema.

## 10.3.2 OCL (Object Constraint Language)

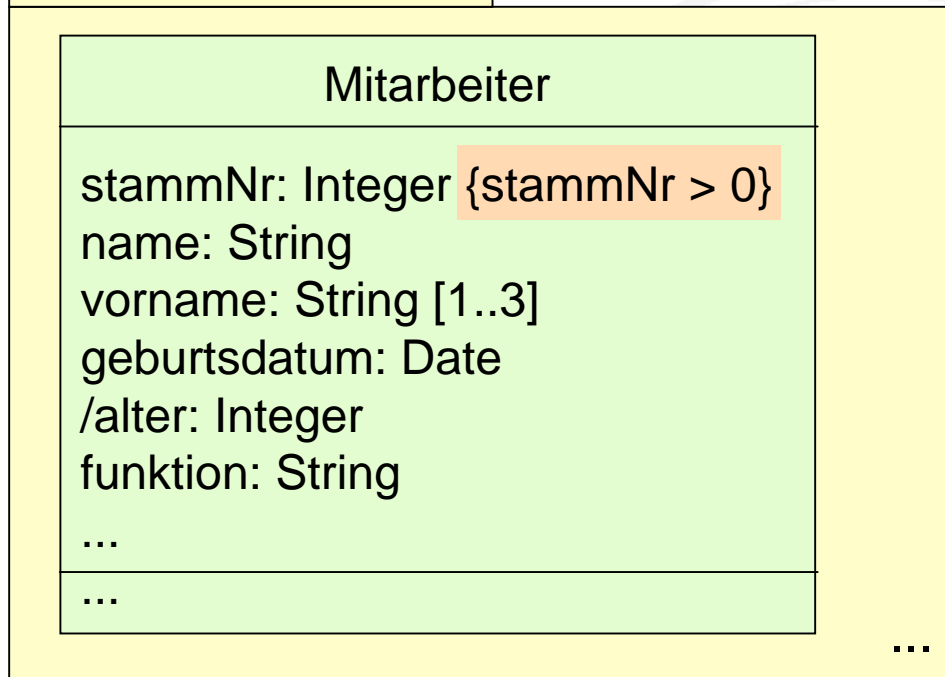
---

- Was ist OCL?
  - Eine **textuelle formale** Sprache
  - Dient zur **Präzisierung** von UML Modellen
  - Jeder OCL-Ausdruck steht im **Kontext** eines UML Modellelements
  - Ursprünglich von IBM als Sprache zur formalen Formulierung von Integritätsbedingungen entwickelt
  - 1997 in UML 1.1 integriert
  - Liegt aktuell in **Version 2.0** vor
- Verwendung von OCL
  - Spezifikation von **Invarianten** (i.e. zusätzliche **Restriktionen**) auf UML Modellen
  - Spezifikation der **Semantik von Operationen** in UML Modellen
  - Auch verwendbar als **Anfragesprache** auf UML Modellen



# OCL-Ausdrücke: Invarianten

Mitarbeiterverwaltung

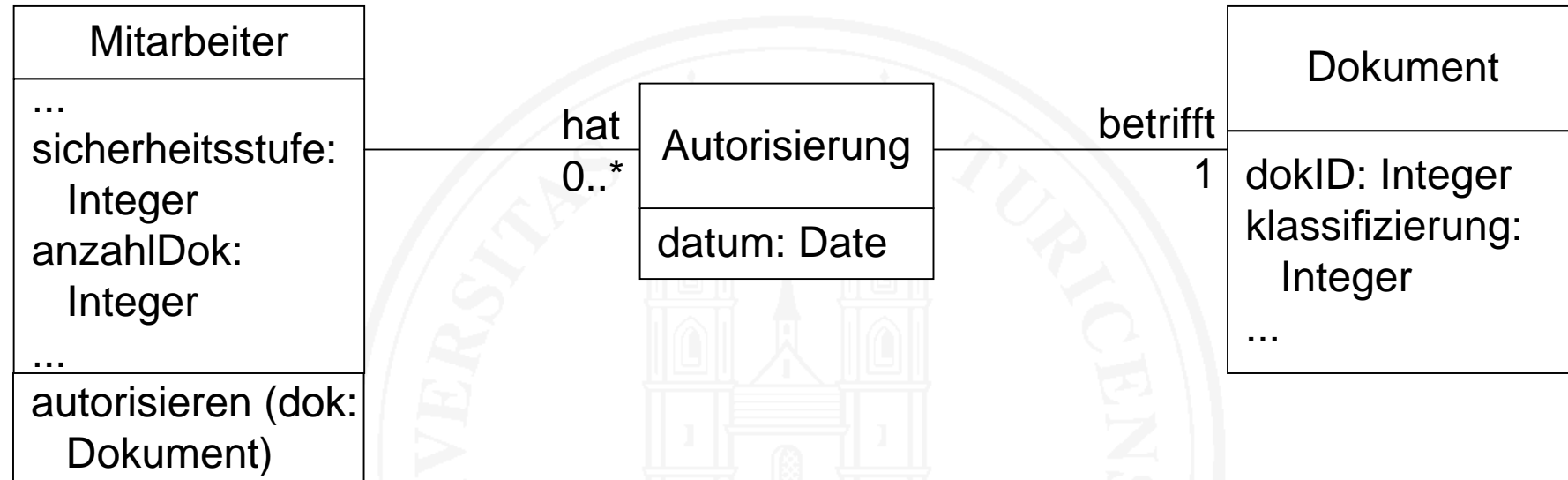


- OCL-Ausdruck ist Bestandteil eines UML Modellelements
- Kontext für OCL-Ausdruck ist implizit gegeben

**context** Mitarbeiterverwaltung::Mitarbeiter **inv:**  
self.funktion = "Fahrer" **implies** self.alter ≥ 18

- OCL Ausdruck wird separat aufgeschrieben
- Kontext muss explizit spezifiziert werden

# OCL Ausdrücke: Semantik von Operationen



**context** Mitarbeiter::autorisieren (dok: Dokument)  
**pre:** self.sicherheitsstufe  $\geq$  dok.klassifizierung  
**post:** anzahlDok = anzahlDok@pre + 1  
**and**  
self.hat->**exists** (a: Autorisierung | a.betrifft = dok)

# Navigation, Aussagen über Mengen in OCL

---

- Personen mit Sicherheitsstufe 0 können für kein Dokument autorisiert werden:

**context** Mitarbeiter **inv**: self.sicherheitsstufe = 0 **implies**  
self.hat->isEmpty()

Navigation vom aktuellen Objekt zu einer Menge assoziierter Objekte

Anwendung einer Funktion auf eine Resultatmenge

# Navigation, Aussagen über Mengen in OCL – 2

---

Weitere Beispiele:

- Die Anzahl der Dokumente für jeden Mitarbeiter muss mit der Menge der bestehenden Autorisierungen übereinstimmen:

**context** Mitarbeiter **inv**: self.hat->size() = self.anzahlDok

- Die für eine Person autorisierten Dokumente sind alle voneinander verschieden

**context** Mitarbeiter **inv**: self.hat->forAll (a1, a2: Autorisierung |  
a1 <> a2 **implies** a1.betrifft.dokID <> a2.betrifft.dokID)

- Es gibt maximal 1000 Dokumente:

**context** Dokument **inv**: self.allInstances()->size() ≤ 1000

# Elemente von OCL (Auszug)

---

- **Art und Kontext:** `context`, `inv`, `pre`, `post`
- **Prädikatenlogische Ausdrücke:** `and`, `or`, `not`, `implies`, `exists`, `forAll`
- **Fallunterscheidung:** `if then else`
- **Operationen auf Resultatmengen:** `size()`, `isEmpty()`, `notEmpty()`, `sum()`
- **Funktionen zur Modellreflexion**, zum Beispiel liefert die Funktion `self.oclIsTypeOf (Mitarbeiter)` im Kontext von `Mitarbeiter` immer den Wert WAHR
- Aussagen über **alle Instanzen** einer Klasse: `allInstances()`
- **Navigation:** übliche **Punktnotation** `self.hat.datum = ...`
- **Anwendung auf Mengen:** **Pfeilnotation** `self.hat->size()`
- **Zustandsveränderung:** **@pre-Notation** `anzahlDok = anzahlDok@pre + 1`

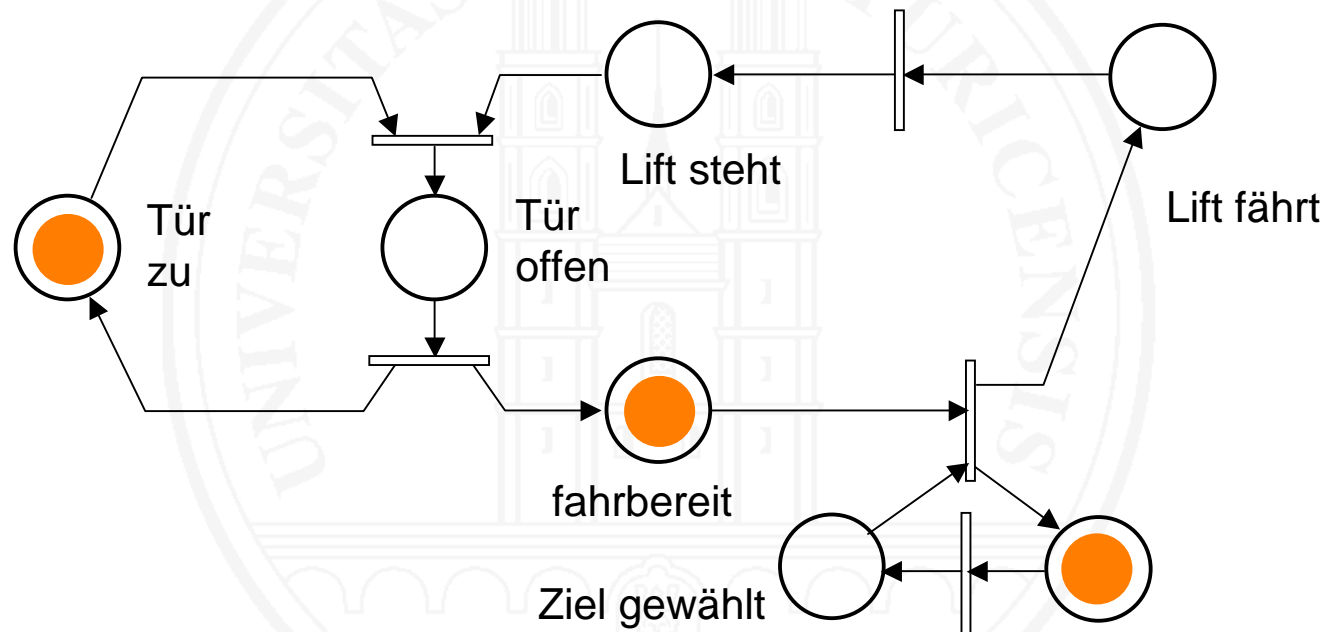
## 10.4 Nachweis geforderter Eigenschaften

---

- Werden Anforderungen durch Modelle beschrieben, so möchte man wissen, ob das Modell gewisse geforderte Eigenschaften aufweist.
- Formale Spezifikationen erlauben
  - die Gültigkeit von Eigenschaften zu **beweisen**  
**Verfahren:** mathematisch-logisches Schließen, unterstützt durch **Theorembeweiser-Software**
  - die Gültigkeit von Invarianten **automatisiert zu testen** und ggf. Gegenbeispiele zu finden  
**Verfahren:** Systematisches, automatisiertes Explorieren des gesamten Zustandsraums der Spezifikation und Prüfen der gewünschten Eigenschaft in jedem Zustand (**Model Checking**)
- Eigenschaften, deren Gültigkeit man beweisen oder testen möchte, sind zum Beispiel **sicherheitskritische Invarianten**

# Beispiel: Beweis einer Sicherheitseigenschaft

Eine stark vereinfachte Liftsteuerung sei mit einem einfachen Petrinetz modelliert (vgl. Vorlesung Informatik IIa: Modellierung):



Zu beweisen sei die Sicherheitseigenschaft, dass der Lift nur mit geschlossenen Türen fahren kann

# Beispiel: Beweis einer Sicherheitseigenschaft – 2

---

Beweis:

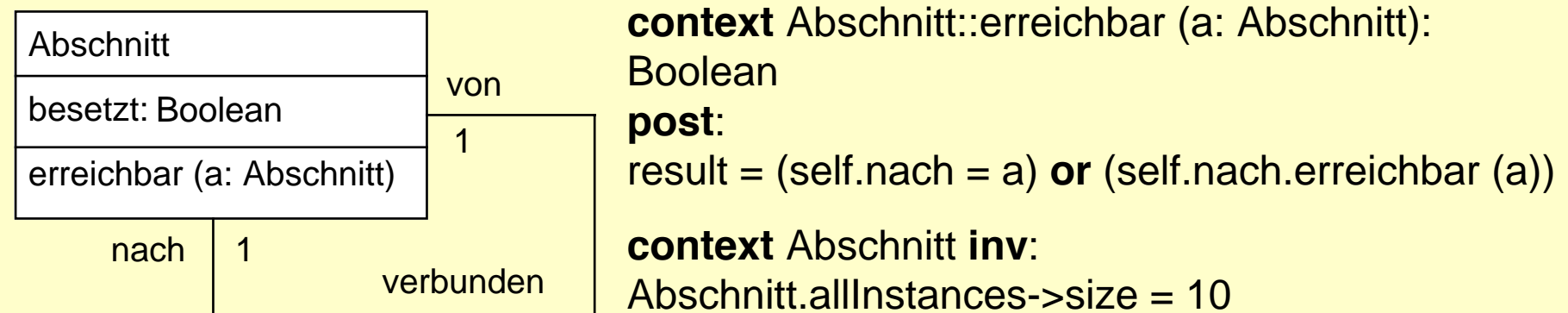
Zu beweisen ist, dass die Transition von fahrbereit nach Lift fährt nur feuern kann, wenn Tür zu markiert ist

- Aufgrund der Definition einfacher Petrinetze gilt
  - Die Transition von fahrbereit nach Lift fährt kann nur feuern, wenn fahrbereit markiert ist (1)
  - Wenn fahrbereit markiert ist, kann Tür offen nicht markiert sein (2)
  - Wenn Tür offen nicht markiert ist, muss Tür zu markiert sein (3)
- (1), (2), (3)  $\Rightarrow$  Die Transition von fahrbereit nach Lift fährt kann nur feuern, wenn Tür zu markiert ist  $\square$



# Aufgabe 10.2 Modell einer Ringbahnlinie

Eine ringförmige S-Bahn-Strecke mit 10 aufeinander folgenden Streckenabschnitten sei mit UML und OCL wie folgt modelliert:



In einem Ring muss jeder Abschnitt von jedem anderen Abschnitt und von sich selbst aus erreichbar sein. Es muss also gelten:

**context** Abschnitt **inv** (1)  
Abschnitt.allInstances->forAll (x, y | x.erreichbar (y) )

a) Falsifizieren Sie diese Invariante, indem Sie ein Gegenbeispiel finden

## Aufgabe 10.2 Modell einer Ringbahnlinie – 2

Lediglich folgende triviale Invariante ist beweisbar:

**context** Abschnitt **inv**:

Abschnitt.allInstances->forall (x | x.erreichbar (x) )

b) Führen Sie den Beweis mit Hilfe der Definition der Operation erreichbar

Das Modell der S-Bahn-Strecke ist offensichtlich falsch. Es bildet die Eigenschaft, ringförmig zu sein nicht korrekt ab.

c) Wie müsste das Modell verändert werden, damit die Erreichbarkeits-Invariante (1) korrekt wird?

# 10.5 Bewertung formaler Spezifikation

---

## Stärken

- + Immer eindeutig (da Semantik formal definiert)
- + Widerspruchsfreiheit formal prüfbar
- + Erfüllung wichtiger Eigenschaften beweisbar / automatisiert testbar
- + Lösungsneutral
- + Formale Verifikation von Programmen möglich
- + Modelle simulierbar/animierbar, z.B. Petrinetze

# Bewertung formaler Spezifikation – 2

---

## Schwächen

- Erstellung sehr aufwendig
- Prüfung/Nachweis der Vollständigkeit wird nicht einfacher
- Nicht ohne profunde Ausbildung lesbar → Prüfung auf Adäquatheit schwierig
- Große Spezifikationen auch für Fachleute schwer zu verstehen
- Aspekte wie Benutzerschnittstellen sind praktisch nicht modellierbar
- Beschreibung von Ausnahmefällen schwierig
- Zum Teil muss perfekte Technologie angenommen werden

# Machbarkeit/Wirtschaftlichkeit formaler Spezifikation

---

- **Marginale Rolle** in der **Praxis**
  - trotz theoretischer Vorteile
  - trotz intensiver Forschung (zum Beispiel Algebraische Spezifikation seit ca. 1977)
- Einsatz heute
  - **Punktuell sinnvoll** und **möglich**
  - Vor allem für **sicherheitskritische** Komponenten
  - Einsatz in der Breite
    - **nicht möglich** (Prüfung auf Adäquatheit!)
    - **nicht sinnvoll** (unwirtschaftlich)
- Auch möglich: Teilformale Spezifikation mit gezielter Formalisierung kritischer Teile

# Literatur

---

Björner, D., C. Jones (1978). *The Vienna Development Method*. Berlin, etc.: Springer.

Jacky, J. (1997). *The Way of Z: Practical Programming With Formal Methods*. Cambridge: Cambridge University Press.

OMG (2005). *UML Superstructure Specification, v2.0*. OMG document formal/05-07-04.  
<http://www.omg.org/cgi-bin/doc?formal/05-07-04>

OMG (2006). *Object Constraint Language, v2.0*. OMG document formal/06-05-0  
<http://www.omg.org/cgi-bin/doc?formal/06-05-01>

Pepper, P. et al. (1982). Abstrakte Datentypen: Die algebraische Spezifikation von Rechenstrukturen. *Informatik-Spektrum* **5**, (1982). 107-119.

Spivey, J.M. (1992). *The Z Notation: A Reference Manual*. Second Edition. Hemel Hempstead: Prentice Hall International.

Wordsworth, J.B. (1992). *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Wokingham, etc.: Addison-Wesley.