technische universität
dortmund

Software- Engineering für langlebige Systeme

Thomas Ruhroth | Dortmund
SoSe 2014

Softwarewartung auf Modell-Ebene

VL4

- Einführung
- Semantik
- Trace und Transitionssystem-Semantik
- Semantiken für UML

- Ziele:
  - Verstehen und beherrschen der Grundlagen für die Definition von Evolution und Co-Evolution auf Modellen

Wiederholung – Modell (Herbert Stachowiak)

Ein Modell ist ein beschränktes Abbild der Wirklichkeit:

- Abbildung
  - Repräsentation eines natürlichen oder eines künstlichen Originals
- Vereinfachung
  - Nicht relevante Eigenschaften sind nicht dargestellt
- Pragmatismus
  - Modelle gelten nur
    - Für bestimmte Subjekte
    - Innerhalb eines bestimmten Zeitintervalls
    - Für einen bestimmten Zweck

Langlebige Modelle haben **dieselben** Probleme wie langlebiger Code

- Die Modelle müssen angepasst werden
- Die Modelle können falsch
  - Verstanden werden
  - Angepasst werden
  - Erodieren
- Modelle können schlecht strukturiert sein

Langlebige Modelle haben **andere** Probleme wie langlebiger Code

- Modelle können nicht ausgeführt werden
- Modelle stellen nicht alle Sachverhalte dar
- Modelle haben nicht immer eine feste Semantik
- Modelle kann man nicht testen
- Modelle kann man schlecht vergleichen

Evolution – Vergleich Modelle vs. Code

Modell

- Änderung von Text/Graphiken
- Strukturorientiert
- Vergleichbarkeit „schwierig“
- Diff/Versionierung schwierig
- Detaillierungsgrad schwankt

Code

- Änderungen von Text
- Struktur-/Algorithmen-Orientiert
- Vergleichbarkeit „einfach“ (Aufwand nur durch Menge der Fälle)
- Diff/Versionierung brauchbar gelöst
- Fester Detaillierungsgrad

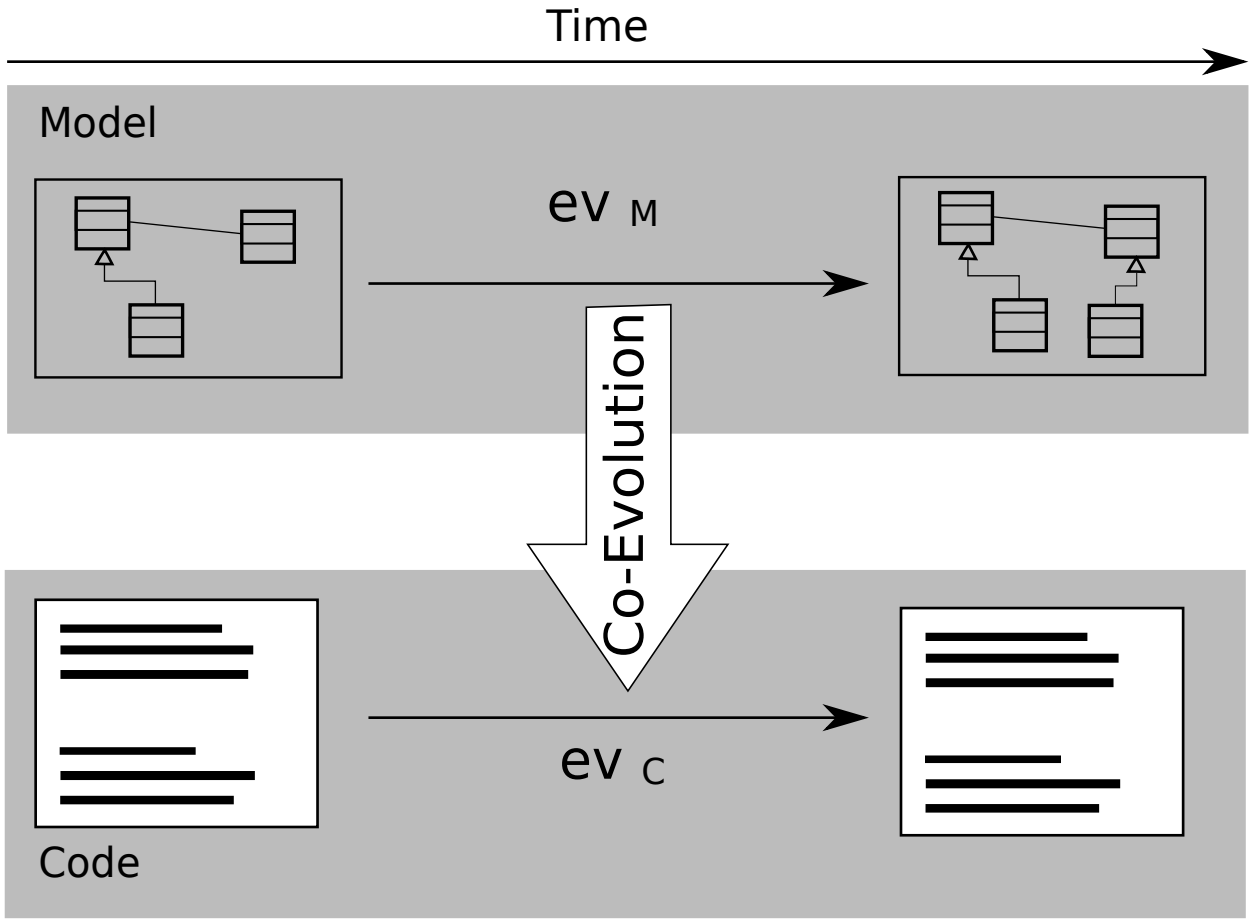
Vergleichbarkeit - Code

- Vergleich von Programmläufen
  - Gleiche Eingaben führen zu gleichen Ausgaben
  - Instrumentierung
  - Tests
- Vergleich der Funktionalität
  - Vergleich von Programmläufen
  - Vergleich von Oberflächen

Vergleich von Modellen

- Probleme
  - Modelle sind Vereinfachungen
    - Daher nur Vergleich innerhalb der Vereinfachung möglich
  - Modelle sind nicht „ausführbar“
    - Vergleiche müssen formal durchgeführt werden
  - Verschiedene Modelle können verschiedene Bereich abbilden
    - Vergleich nur auf „Überschneidungen“ möglich
  - Modelle können textuell oder graphisch oder beides (gemischt) sein

Modelle und Code



Plan

- 1. Modelle und Semantik (hier Traces und Transitionssysteme)
- 2. Vergleich von Modellen, Evolution und Refactorings
- 3. Verhältnis von Modell und Code, Co-Evolution

Modelle sind nicht „ausführbar“

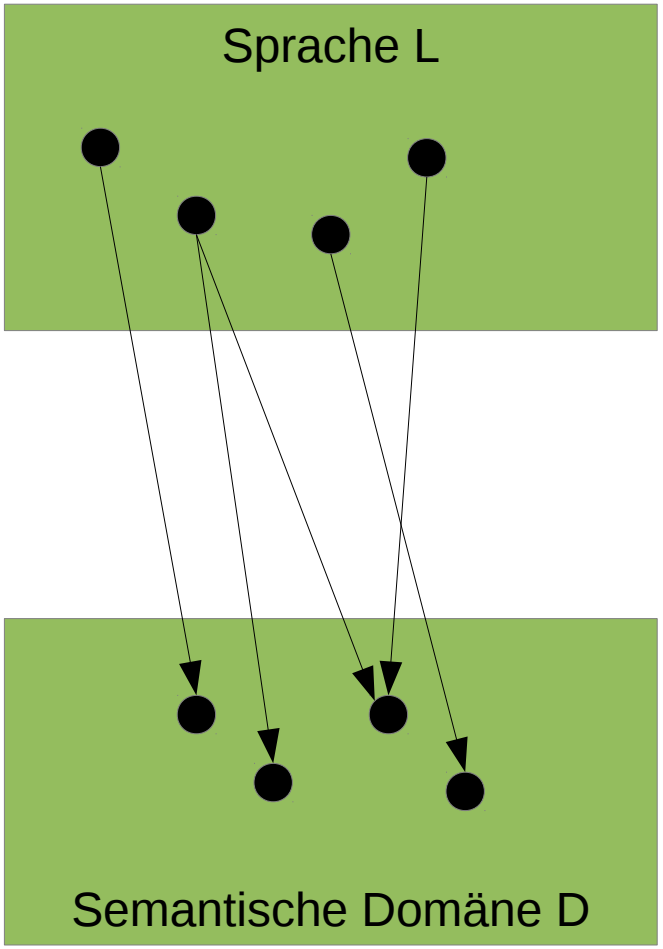
- Semantik
  - Beschreibt die Bedeutung von etwas (später genauer)
  - Kann formal/informal sein
  - Bei Modellen meist Vereinfachung der Wirklichkeit

- Vorgehen
  - Definition (und Verstehen) von Semantiken
  - Vergleichsrelationen auf Semantiken
    - Refinement
    - Gleichheit
    - Simulation

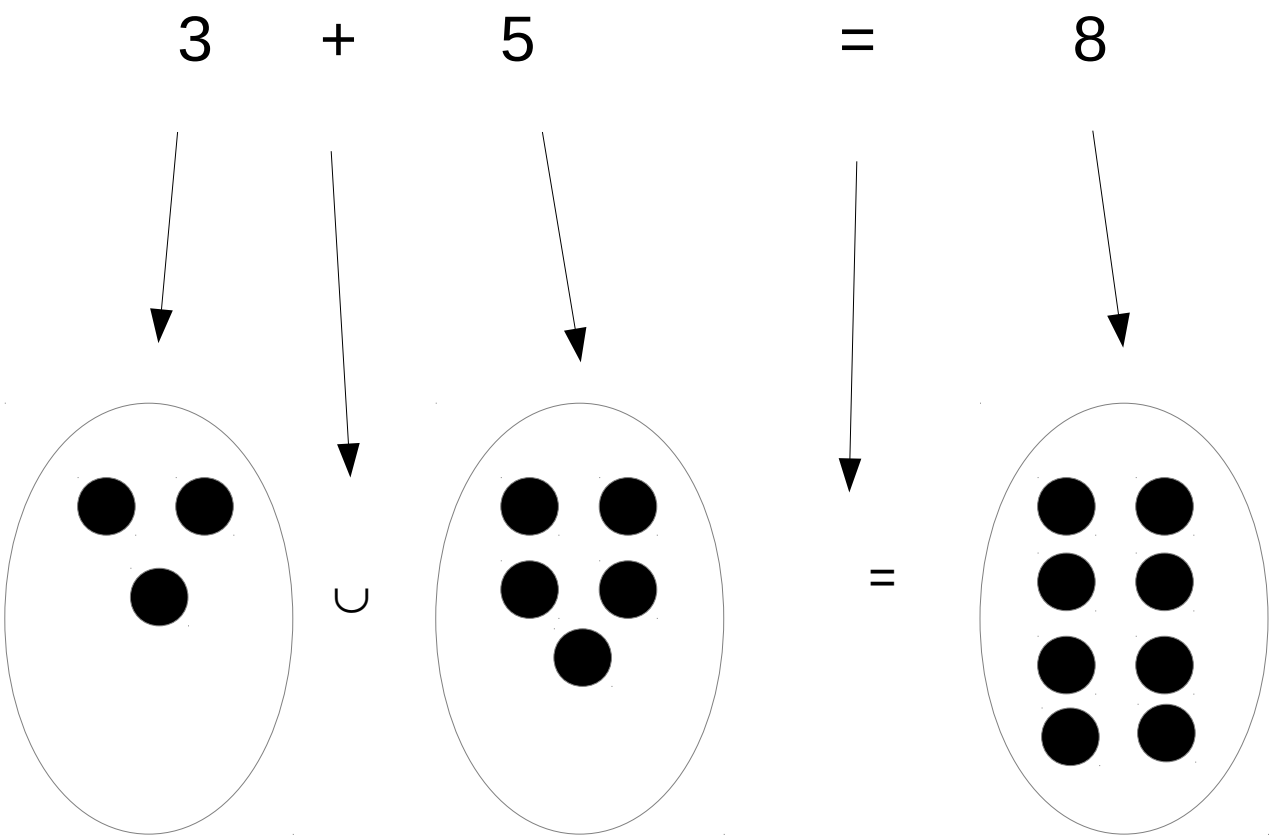
Semantik

- Eine Semantik für eine Sprache L ist ein Tupel (D, [[.]]) aus einer semantische Domäne und einer Abbildung [[.]]: L → D, der semantischen Abbildungsfunktion.

- Die semantische Domäne kann ein formaler Formalismus sein (formale Semantik) oder eine informale Beschreibung (z.B. natürliche Sprache)



Apfel-Semantik für Natürliche Zahlen



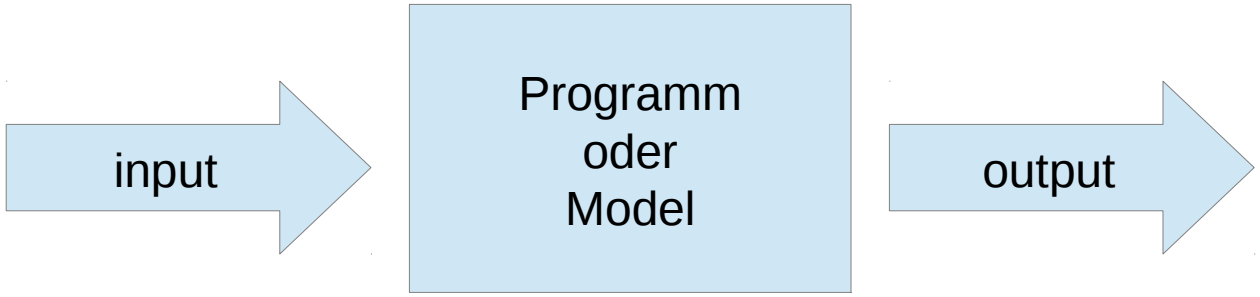
Sprache L

Semantischen Abbildung

Semantische Domäne D

Input/Output-Semantik - Trace-Semantik

- Das Modell/Programm generiert für eine Input-Folge I eine Output-Folge O.
  - Bsp: der UNIX-Befehl sort liefert für die Eingabe [7,4,9,1,3] (=I) den Output [1,3,4,7,9] (=O)
- Eignet sich auch für EVA-Programme, Methoden, ….
- Also nicht nur für formale Methoden :-)



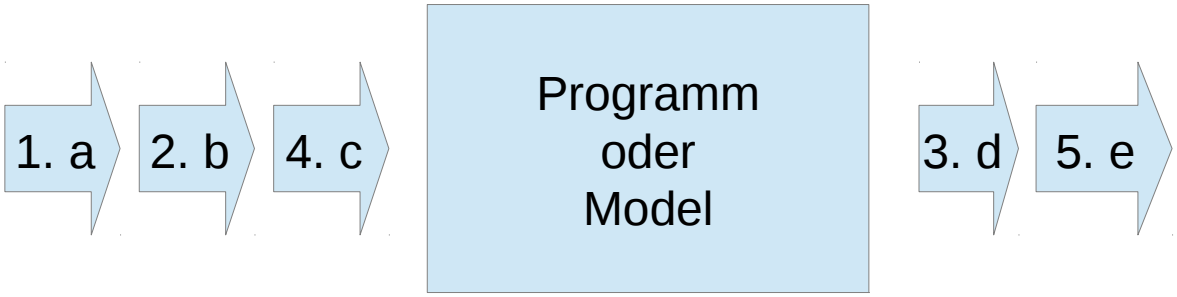
Traces

- Kurze Darstellung von Events mit Inputs und Output:
  - Eine Trace von Events:
    < evt1, evt2, ...>
    < drückeKnopfA, erhalteProduktA>
  - Eine Trace mit Input und Output:
    - ? steht für Input
    - ! steht für Output

    < drückeKnopf?A, erhalteProdukt!A>

Eine Trace ist eine geordnete Menge von Events

< a?u,b?v,d!w,c?x,e!y>



Events

- Ereignisse eines Programms
  - Tastendruck, Oberflächeninteraktion
  - Log-Ausgaben
- Systeminterne Aufrufe
  - Methodenaufruf bzw. Abschluss
  - Interrupts
  - Exceptions

„Sichtbares“ Verhalten

- Eine Trace beschreibt eine Ausführung mit ihrem „sichtbaren“ Verhalten
  - Evtl. Sichtbarmachung durch Instrumentierung oder Logs

- Ein System kann durch die Menge seiner möglichen Ausführungen $Tr \subseteq$ Events* beschrieben werden
- Da ein System zu jedem Zeitpunkt abgebrochen werden kann, ist Tr prefixabgeschlossen:
  - $\forall\ t \in Tr : \exists\ t' \in Tr\ ,\ e \in Events: t = t' \oplus \langle e \rangle$
  - $\oplus$ ist die Konkatenation

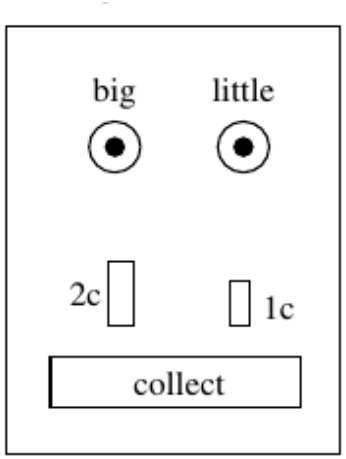
Traces von Modellen

- Problem:
  Modelle haben keine „Ausführungen“ an sich
- Definition der Traces über alle möglichen Traces

Traces von Code

- Aufrufe und Returns von Methoden
- Bezug auf beliebiges geschlossenes Teilsystem
  - Auch Klassen
- Einzelne Traces lassen sich technisch gewinnen

Vending Machine

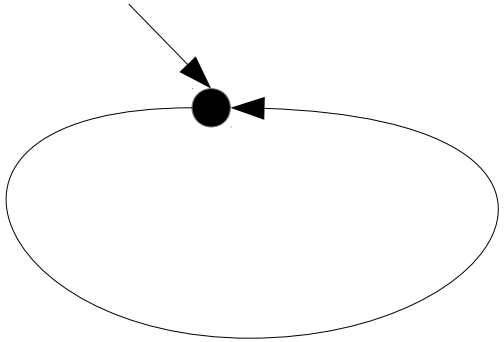


VendingMa
big little 1c 2c collect

Vending Machine

Welche Traces kann diese Klasse Ausführen?

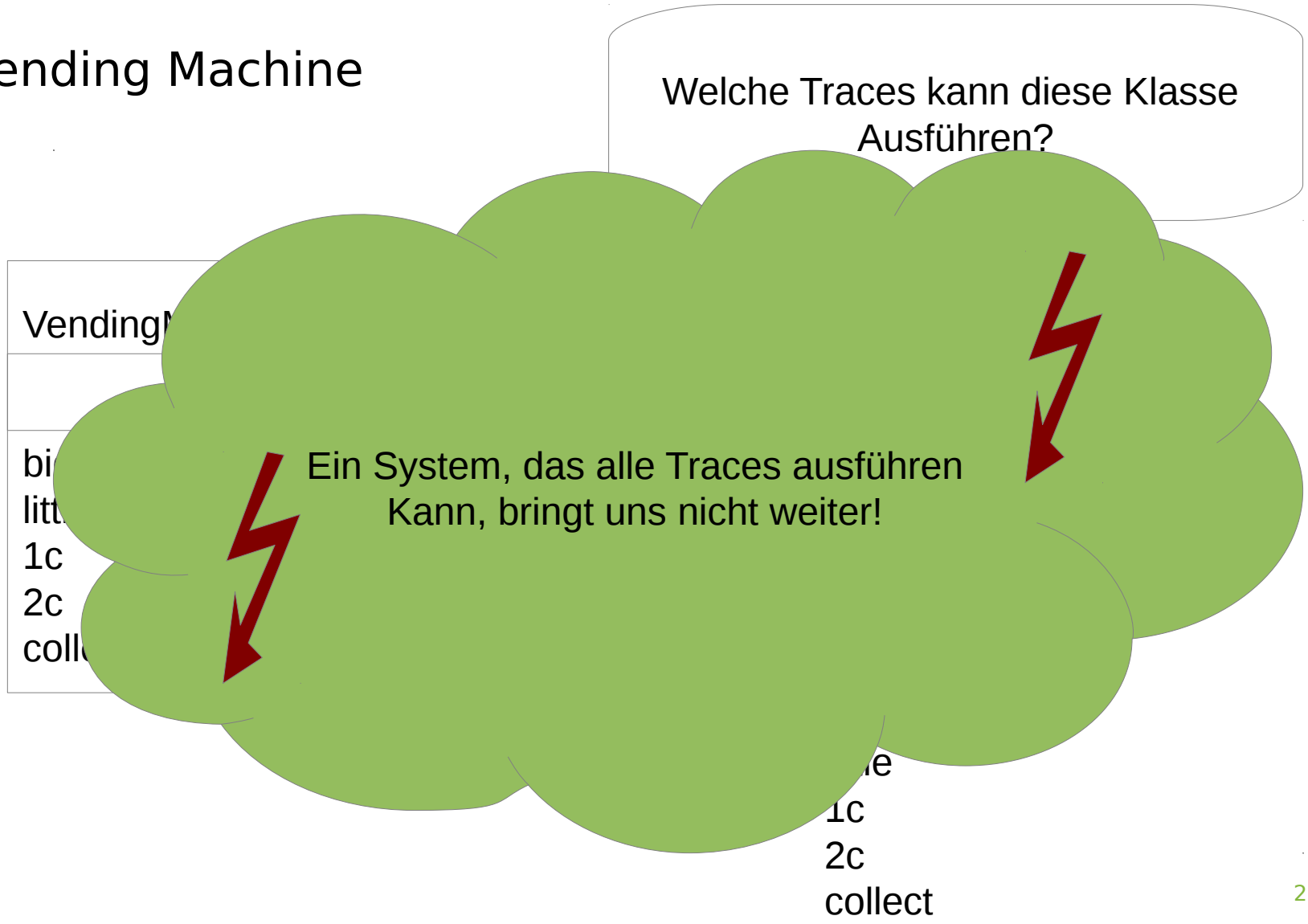
VendingMa
big little 1c 2c collect



big
little
1c
2c
collect

Vending Machine

Welche Traces kann diese Klasse Ausführen?



Klassendiagramme in UML

- Klassendiagramme sind in der Praxis oft unterspezifiziert
- Unterspezifizierte Klassendiagramme lassen sich nur verstehen durch
  - Implizite Informationen
    - Wissen, was ein komplexer Netzknoten ist.
    - Wissen, was die Aufgabe eines komplexen Netzknoten ist.
  - Hintergrundwissen
    - Wissen, dass eine Methode Namens *Add* „etwas hinzufügt“
  - Explizite Beschreibung der Aufgaben
- Man kann Klassen, Methoden etc. genauer spezifizieren.
  - Weitere Diagrammtypen
  - Vor- und Nachbedingungen, Invarianten

Beispiel Methode „Add“

- Eine Methode int add(int a, int b) aus einem Modell kann ohne weitere Spezifikation jede Funktion zweier Integer in ein Integer darstellen:
  - { return a – b; }
  - { return a > b?a:b; }
  - { return a +b; }
  - { return b; }
  - ....
- Das System sollte genauer spezifiziert sein

Vor und Nachbedingungen

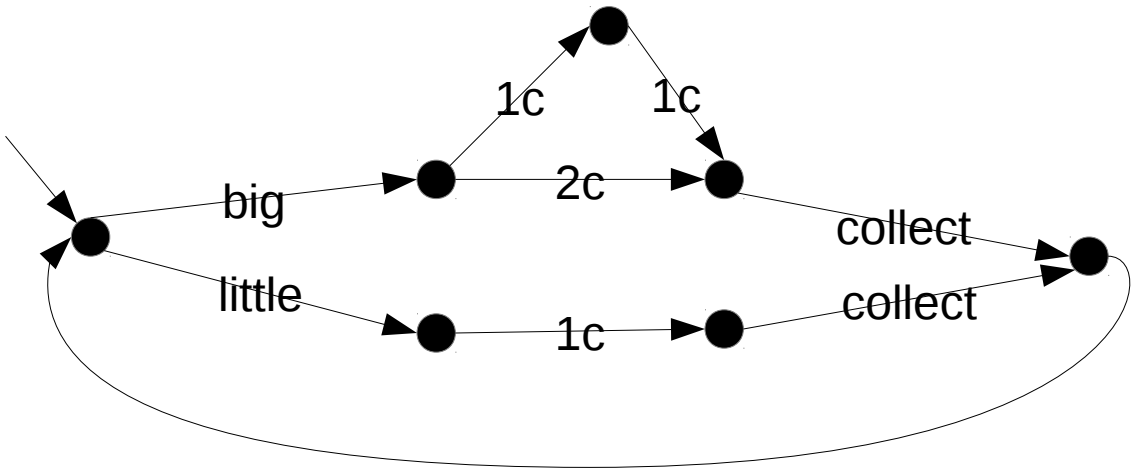
- UML erlaubt Methoden mittels Vor- und Nachbedingungen zu spezifizieren.
- int add(int a, int b)
  - Vorbedingung: a + b < maxInt
  - Nachbedingung: result = a + b
- Wenn die Vorbedingung erfüllt ist, ergibt die Methode einen definierten Wert, der durch die Nachbedingung gegeben ist.

- Instanzvariablen:
  - Ohne Prime: Wert beim Start der Methode (z.B. x)
  - Mit Prime: Wert beim Ende der Methode (z.B. x')

Erweiterte Spezifikation der Vending Maschine

VendingMa
int pay; String sel;
big little 1c 2c collect

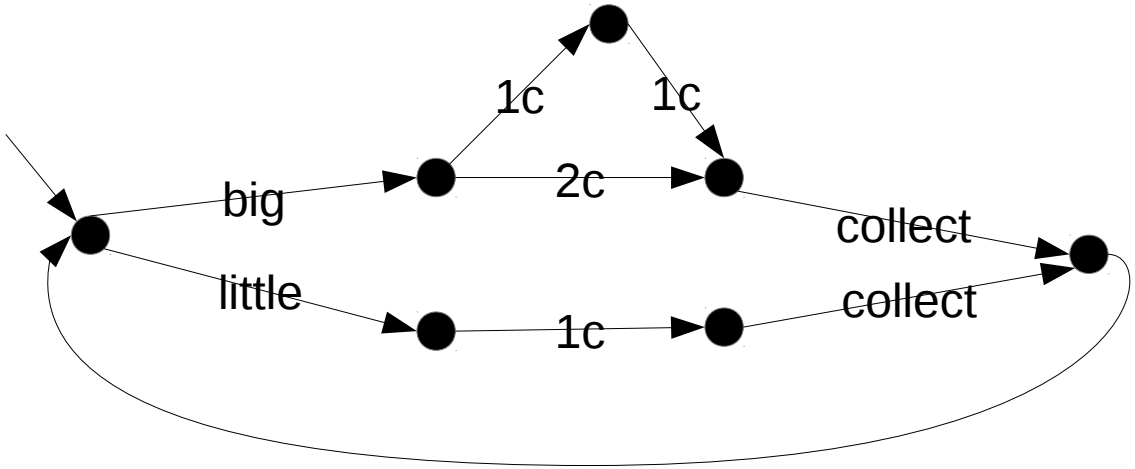
Op	Vor	Nach
big	pay = 0 and sel = „“	pay' = 2 and sel' = „big“
little	pay = 0 and sel = „“	pay' = 1 and sel' = „little“
1c	pay > 0 and sel <> „“	pay' = pay - 1
2c	pay > 1 and sel <> „“	pay' = pay - 2
collect	pay = 0 and sel <> „“	pay' = 0 and sel' = „“



Transition System

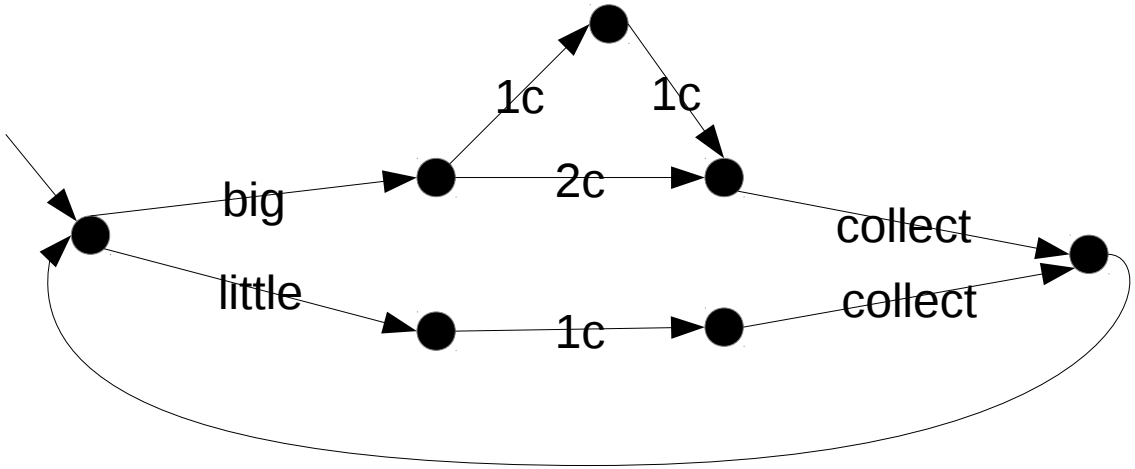
Def. Ein (über Act beschriftetes) Transitionssystem T = (Q , → , q0) besteht aus

- Menge von Zuständen Q,
- einer Transitionsrelation → ⊆ Q × Act × Q,
- und einem Anfangszustand q0 ∈ Q.



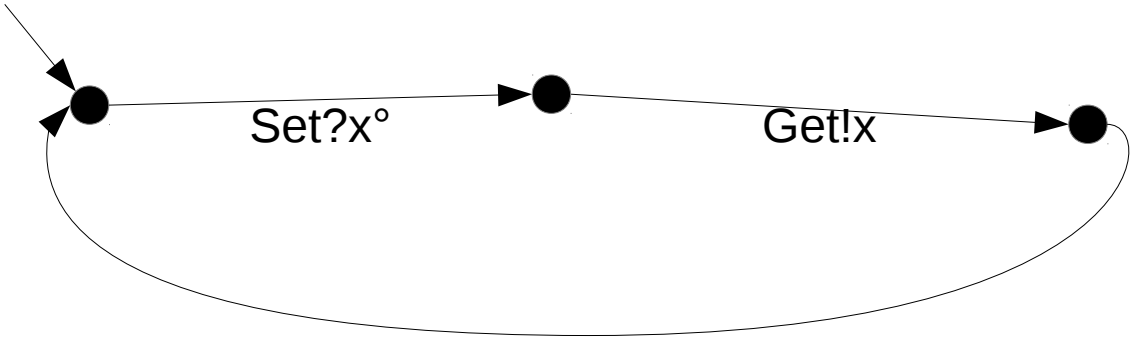
Transitionssysteme und Traces

- Transitionssysteme können als eine Beschreibung aller Traces eines System Tr interpretiert werden



Parameter mit Variablen

- Events mit Parameter können durch Variablen beschrieben werden
  - <set?x, get!y>
- Variablen sollen den Wert behalten:
  - <set?x, get!x>
    - <set?2, get!2>
    - aber nicht <set?3, get!7>
- Neuer Wert: Wenn eine Variable in einem TS einen neuen Wert erhalten soll, kann dies durch ein ° gekennzeichnet werden.



Rückgaben

- Output-Parameter können Berechnungen enthalten

- <setX?x, setY?y, GetSum!x+y>
- <set?x?y, GetSum!x+y>

Nicht sichtbare Übergänge (tau-Transitionen)

- Einige Events sollen im „Innern“ des Systems verborgen bleiben.
- Diese können mittels nicht sichtbarer Transitionen in den Transitionssystemen und den Traces wiedergegeben werden.

Andere Möglichkeiten

- Sequenzdiagramme
  - Nur unter der zusätzlichen Annahme, dass alle Läufe modelliert sind
- Zustandsdiagramme und Aktivitätsdiagramme
  - Ähnlich gut wie Vor- und Nachbedingung
  - Problem: Interpretation

Nächste Woche

- Vergleich von Modellen, Evolution und Refactorings