Softwarequalität

Vorlesung 10 – Testen: Black-box-Verfahren
(Testen reaktiver Systeme)

Prof. Dr. Joel Greenyer

13. Juni 2016

Mehrdimensionale Äquivalenzklassenbildung, weiteres Beispiel

In der letzten Vorlesung…

- Wie sieht hier eine gute Äquivalenzklassenbildung aus?

- Points:
  1. t1.point < t2.points
  2. t1.point = t2.points
  3. t1.point > t2.points

- Goals:
  1. t1.goals < t2. goals
  2. t1.goals = t2. goals
  3. t1.goals > t2. goals

Zellen können verschmolzen werden – Annahme: Hier verhält sich das Programm gleichartig

Tabelle zeigt erwartete Rückgabewerte (SOLL-Werte):

points \ goals	1.	2.	3.
1.	t2	t2	t2
2.	t2	null	t1
3.	t1	t1	t1

points \ goals	1.	2.	3.
1.	t2		
2.	t2	null	t1
3.	t1		

Combinatorial Test Design (CTD)
Interaktion von (Test-)Parametern

In der letzten Vorlesung…

- Statistische Untersuchungen zeigen
  - dass die Ursache von Fehlern meistens vom Wert **einer Variable** abhängt
  - Ein **Großteil der Fehler** kann gefunden werden, wenn **alle Interaktionen** von Werten **zweier Variablen** getestet werden (Level-2-Interaktion)

Table 1. Number of variables involved in triggering software faults

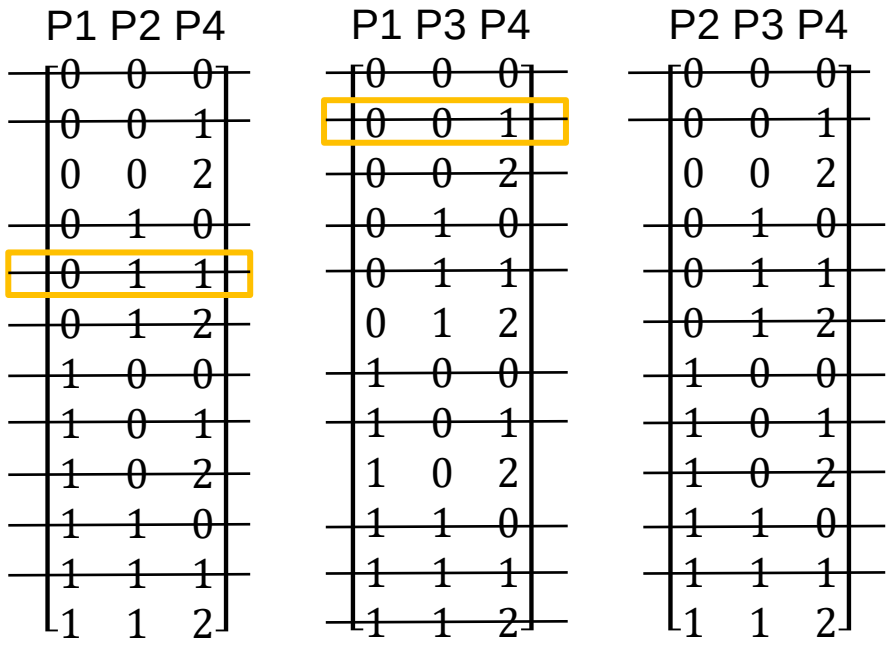
Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

http://csrc.nist.gov/groups/SNS/acts/ftfi.html

CTD – IPOG-Algorithmus

In der letzten Vorlesung…

- Schritt 3 nochmal im Detail:



P1	P2	P3	P4
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	0
1	0	0	1
1	0	1	2
1	1	0	0
1	1	1	1
1	0	1	0
0	1	0	1

… usw.

Testen transformationaler und reaktiver Systeme

- Bisher haben wir Tests **transformationaler** Systeme behandelt

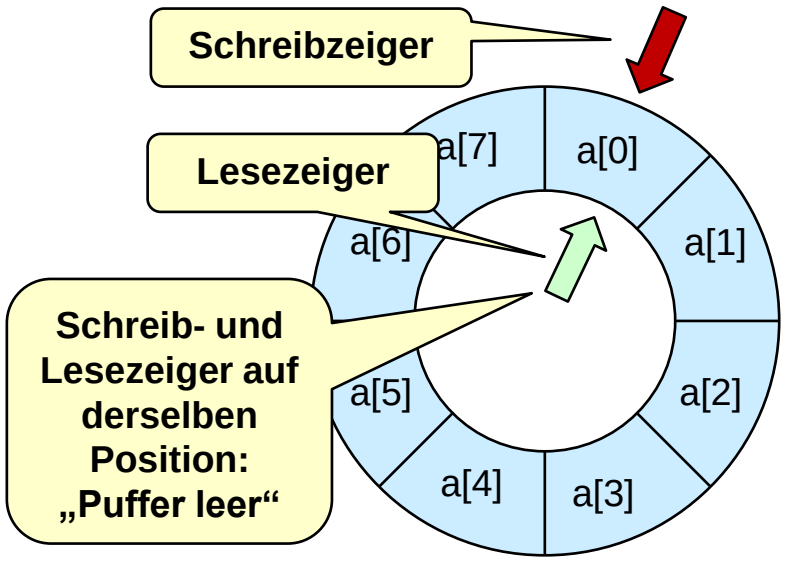
- Wichtige Eigenschaft **transformationaler Systeme**
  - Das Ergebnis einer Berechnung ist **nur von der aktuellen Eingabe abhängig** und **nicht von einer vorherigen Eingabe**
  - Es werden keine Ergebnisse zwischengespeichert
  - Systeme haben **keinen Zustand** (engl. „stateless“)

- Wichtige Eigenschaft **reaktiver Systeme**
  - System hat Zustand (engl. „stateful“),
  - Eingaben verändern Zustand
  - Tests sind **Sequenzen** von Ein- und Ausgaben
    - ggf. mit Überprüfung von erreichten Zuständen

Beispiel für ein einfaches reaktives System: Ringpuffer

- Ringpuffer speichert Werte in einem Array
  - Beim Schreiben/Lesen wird ein Schreib- bzw. Lesezeiger inkrementiert
  - Erreichen Zeiger das Ende des Arrays: wieder auf Null
  - Erreicht Schreibzeiger den Lesezeiger: Lesezeiger wird auch inkrementiert
  - Überholt Lesezeiger Schreibzeiger: Fehler

```
public class RingBuffer {

        int[] a;
        int rp = 0;
        int wp = 0;

        public void write(int value){
                ...
        }

        public int read(){
                ...
        }
}
```

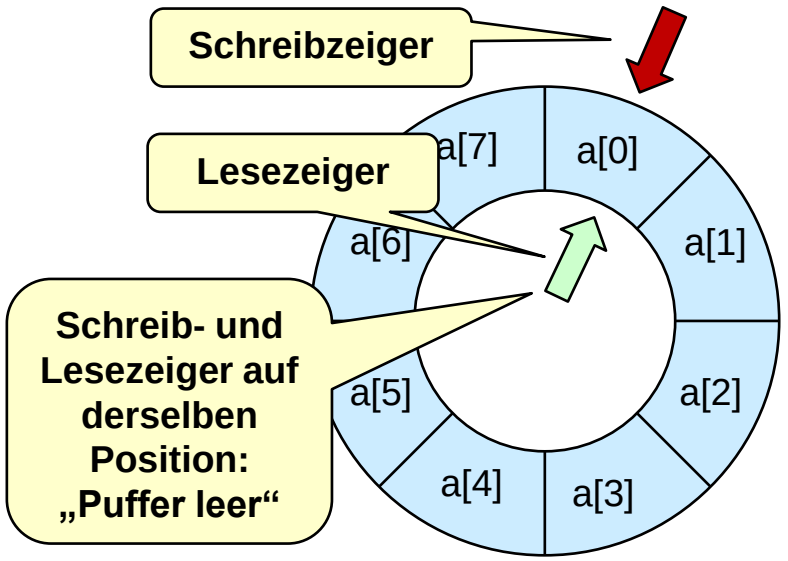


Für Array mit acht Feldern

Beispiel für ein einfaches reaktives System: Ringpuffer

- **Frage**: Was sind sinnvolle Testfälle?

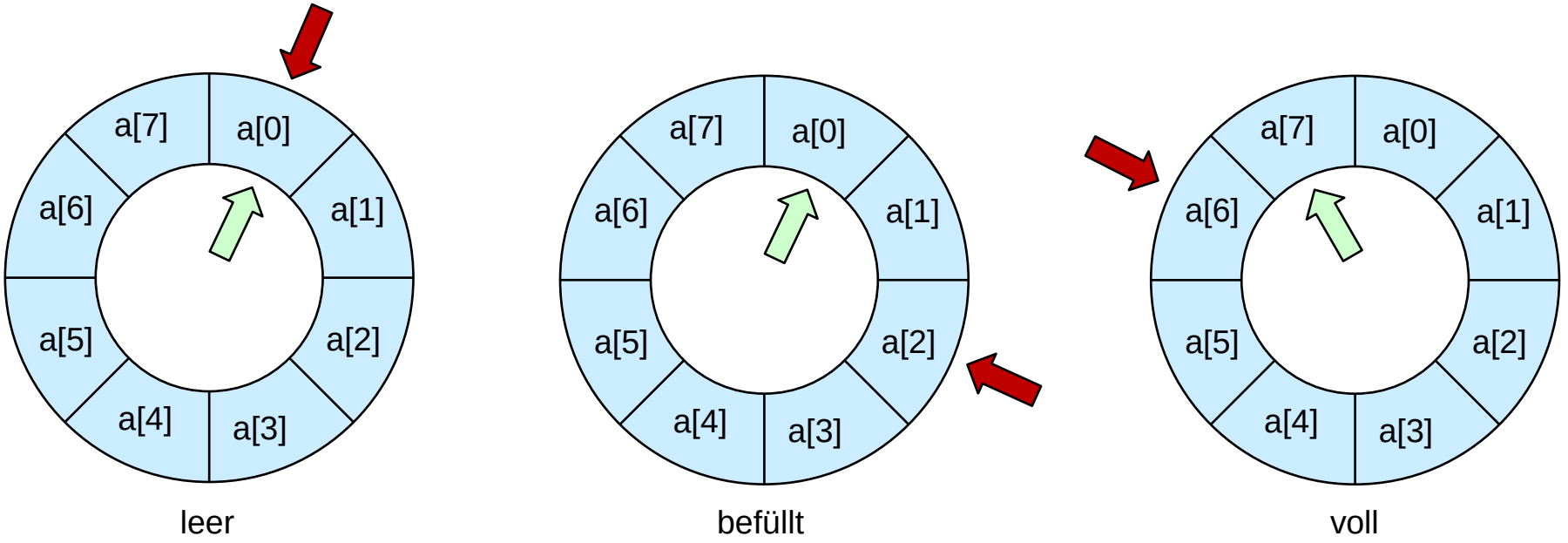
```
public class RingBuffer {

        int[] a;
        int rp = 0;
        int wp = 0;

        public void write(int value){
                ...
        }

        public int read(){
                ...
        }
}
```

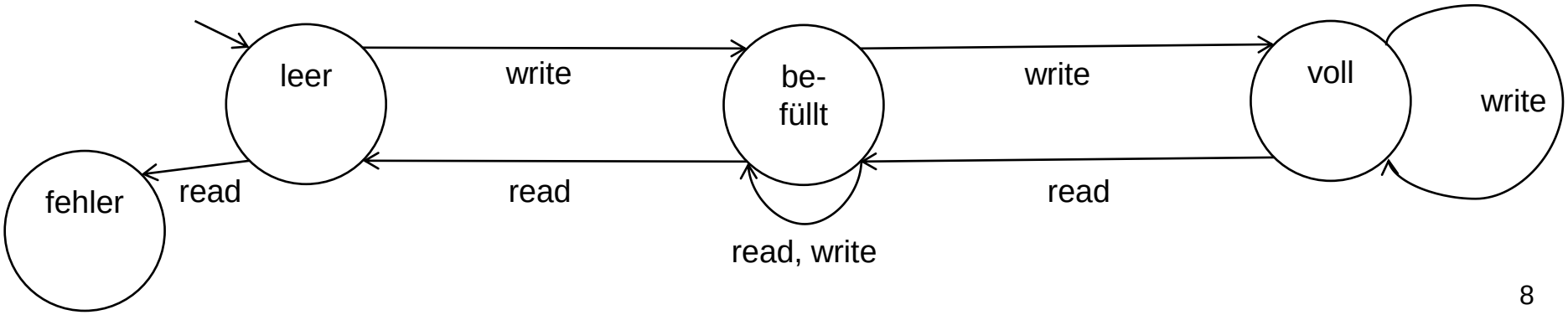


Für Array mit acht Feldern

Beispiel für ein einfaches reaktives System: Ringpuffer

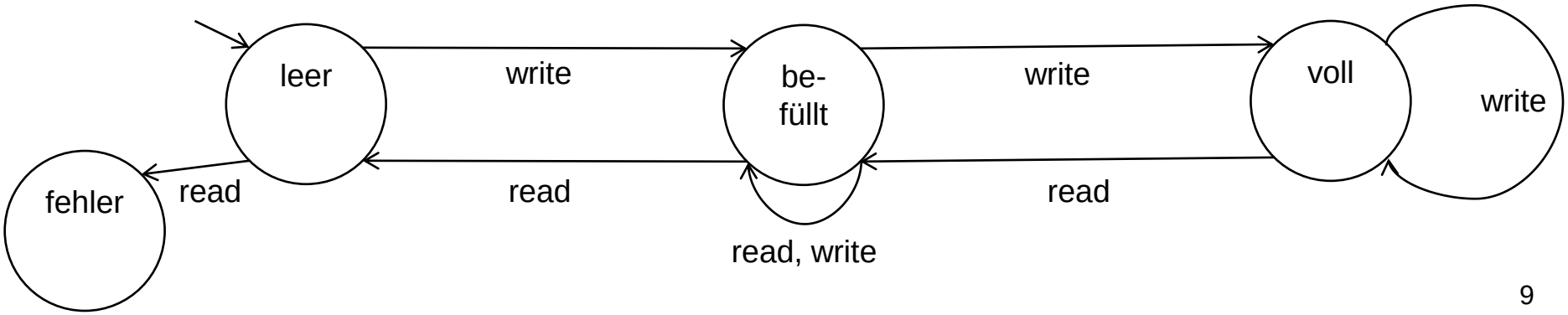
- Zustände des Ringpuffers schematisch und als Automat:





Testen Reaktiver Systeme

- Das Verhalten reaktiver Systeme kann durch einen Zustandsautomaten beschrieben werden
  - Hier: Transitionen entsprechen Eingabeaktionen, Zustände können eine **Abstraktion** des Zustands des Systems sein
  - Transitionen können auch Ausgabeaktionen sein (Beispiel: Getränkeautomat gibt Münze zurück)
- Der Zustandsautomat ist die Spezifikation
  - Bedeutet: Zu jedem Ablauf des Systems muss es einen Pfad im Automaten geben

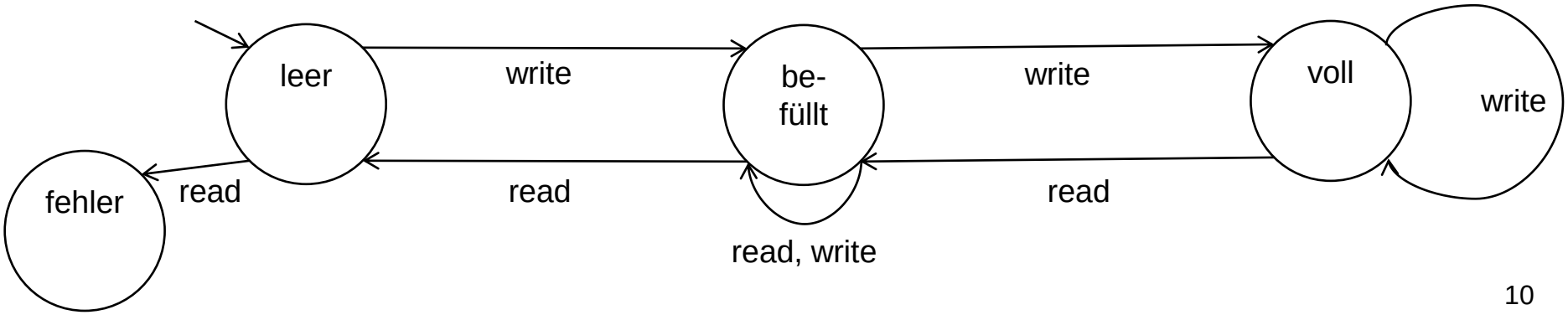


Testen Reaktiver Systeme

- **Idee für das Testen**:
  - Führe das System aus und überprüfe, ob der Ablauf einem Pfad im Zustandsautomaten entspricht
- **Abdeckung**? Verschiedene Kriterien möglich:
  1. Alle Zustände müssen erreicht werden
  2. Alle Transitionen müssen erreicht werden

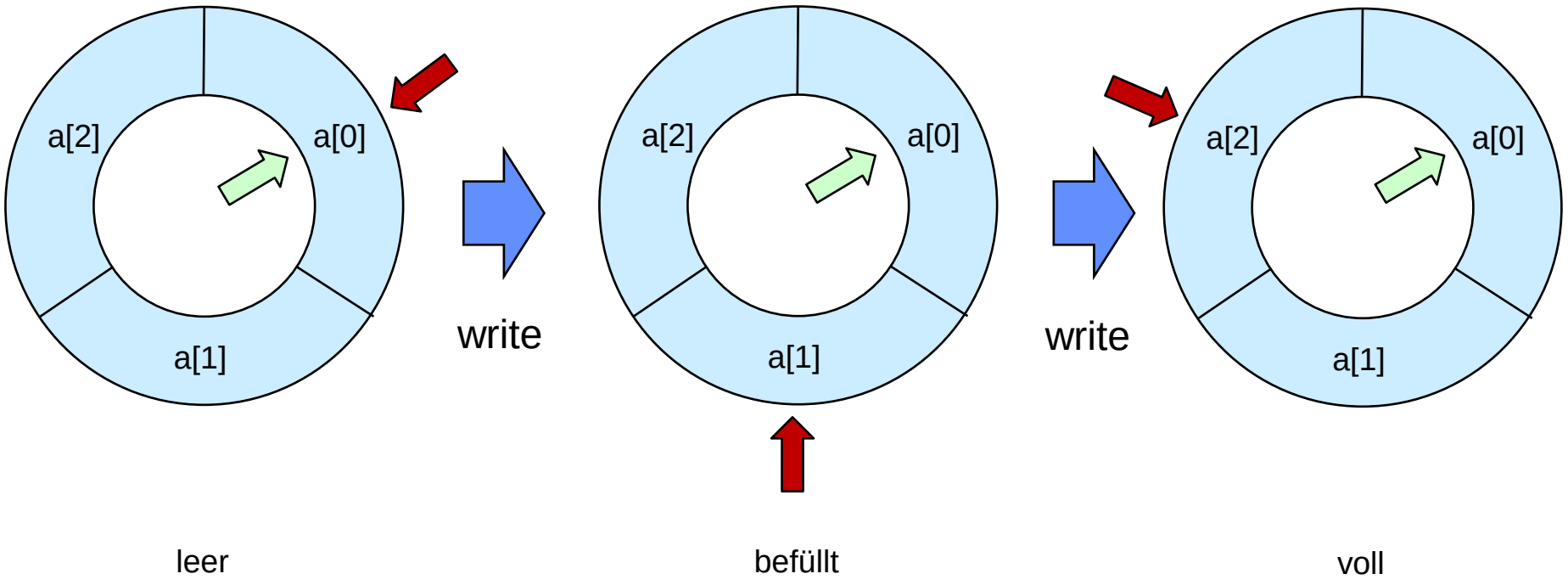
  } Gängigste Kriterien

  3. Alle möglichen Sequenzen bis zu einer bestimmen Länge
  4. Alle möglichen Zyklen von jedem Zustand einmal
  5. …

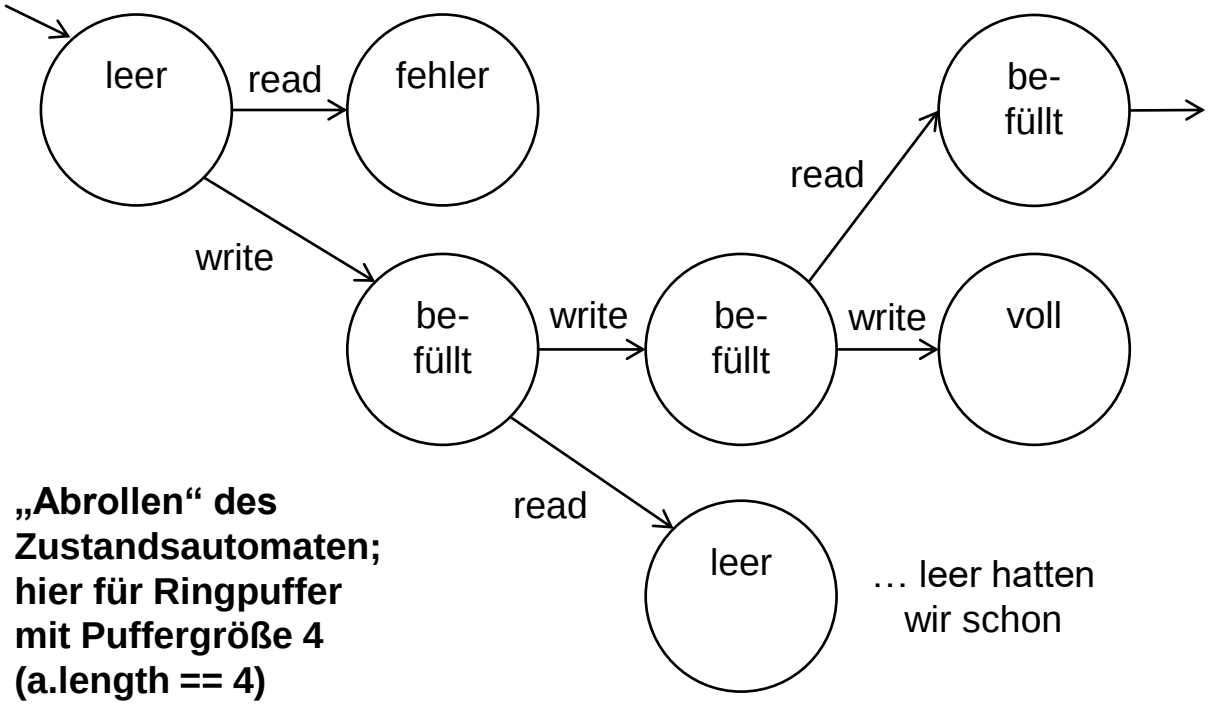


Ringpuffer mit Array der Länge 3

- Hinweis: Ein leerer Ringpuffer mit Array der Länge 3 ist nach zweimal Schreiben voll.



Finden von Tests, die alle Zustände abdecken



… führt nicht unmittelbar zu neuem Zustand (wir wollen zu „voll“, das geht aus „befüllt“ nur mit „write“)

„Abrollen“ des Zustandsautomaten; hier für Ringpuffer mit Puffergröße 4 (a.length == 4)

… leer hatten wir schon

Mit zwei Tests decken wir alle Zustände des Automaten ab:

Test 1:

Eingabe: read()

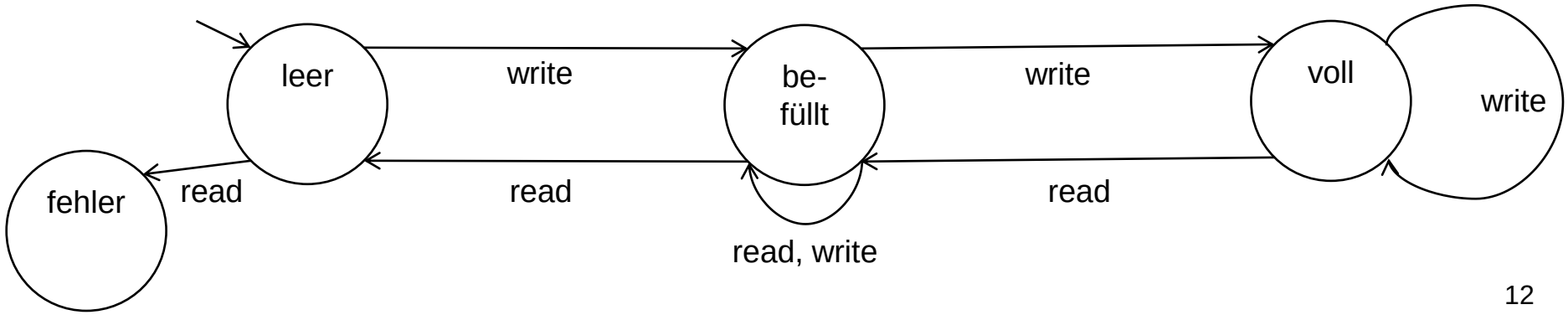
Sollergebnis: Fehler

Test 2:

Eingabe: write(1), write(2), write(3)

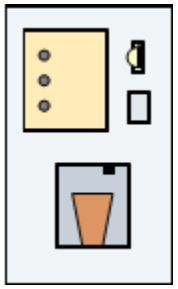
Sollergebnis: Puffer ist voll.

Genauer: a[0] = 1, a[1] = 2, a[2] = 3 und (wp – rp) % a.length == 1



Tests aus Use Cases ableiten

- Typischerweise werden Anforderungen in Form von **Use Cases** erfasst.
- Beispiel Kaffeemaschine:



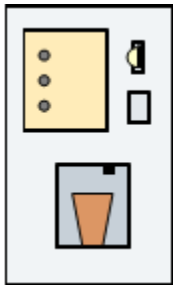
Use Case buy coffee
Primary Actor: User
Precondition: coffee filled, cups magazine filled
Postcondition: …
Main scenario:
1. User inserts coin
2. User chooses coffee
3. Machine dispenses cup
4. Machine pours coffee
5. Machine displays take cup message
6. User takes cup
7. Machine displays ready message

Use Case coin return
Primary Actor: User
Precondition: …
Postcondition: …
Main scenario:
1. User inserts coin
2. User presses coin return button
3. Machine returns coin
4. Machine displays ready message

- Was sind sinnvolle Testfälle?
  - Beide Use Cases durchspielen und prüfen ob die Kaffeemaschine das erwartete tut (okay, einfach)

Tests aus Use Cases ableiten

- Use Cases können Erweiterungen und Alternativen enthalten
  - Diese sollten durch Testfälle abgedeckt werden
- Beispiel:



Use Case buy coffee
Primary Actor: User
Precondition: coffee filled, cup magazine filled
Postcondition: …
Main scenario:
1. User inserts coin
2. User chooses coffee
3. Machine dispenses cup
4. Machine pours coffee
5. Machine displays take cup message
6. User takes cup
7. Machine displays ready message

Alternative coffee empty
Precondition: coffee storage empty
3.a) Machine returns coin
4.a) Machine displays coffee empty message

Test 1(buy coffee)
Setup: setup Machine… fill coffee storage, fill cups
Test sequence:
…

Test 2 (buy coffee + coffee empty)
Setup: Setup Machine… empty coffee storage
Test sequence:
1. User inserts coin
2. User chooses coffee
3. Machine returns coin
4. Machine displays coffee empty message

User ist Tester

Hier Ausgaben prüfen

Tests aus Use Cases ableiten

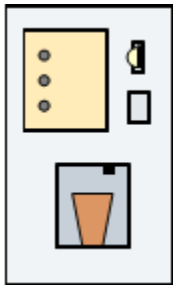
- Use Cases können **gleichzeitig** oder **nebenläufig** auftretende Sequenzen von Ereignissen beschreiben
- Gleichzeitiges Auftreten von Use Cases
  - Wenn z.B. ein System gleichzeitig auf Eingaben von mehreren Benutzern oder Sensoren reagieren muss
    - Beispiel: Elektronisches Fensterhebersystem im Auto – mehrere Insassen bedienen Fenster, Steuerung des Fahrers hat Priorität und Fahrer kann Steuerung auf Rücksitzen deaktivieren
  - Wenn Alternativ-Szenarien in separaten Use Cases spezifiziert sind und nicht als Erweiterungen und Alternativen eines Use Cases
    - Z.B. Use Cases „Kaffee mit Zucker“ und „Kaffee mit Milch“ sind nicht Erweiterungen des Use Cases „Kaffee kaufen“
    - Zum Teil ist es schwierig verschiedene Kombinationen von Alternativen und Erweiterungen aufzuschreiben

Tests aus Use Cases ableiten

- Beispiel für gleichzeitig auftretende Use Cases

Use Case buy coffee with sugar
Primary Actor: User
Precondition: …, sugar filled
Postcondition: …
Main scenario:
1. User inserts coin
2. User chooses sugar
3. User chooses coffee
4. Machine dispenses cup
4. Machine pours coffee
5. Machine adds sugar
5. Machine displays take cup message
6. User takes cup
7. Machine displays ready message

Use Case buy coffee with milk
Primary Actor: User
Precondition: …, milk filled
Postcondition: …
Main scenario:
1. User inserts coin
2. User chooses milk
3. User chooses coffee
4. Machine dispenses cup
4. Machine pours coffee
5. Machine pours milk
5. Machine displays take cup message
6. User takes cup
7. Machine displays ready message



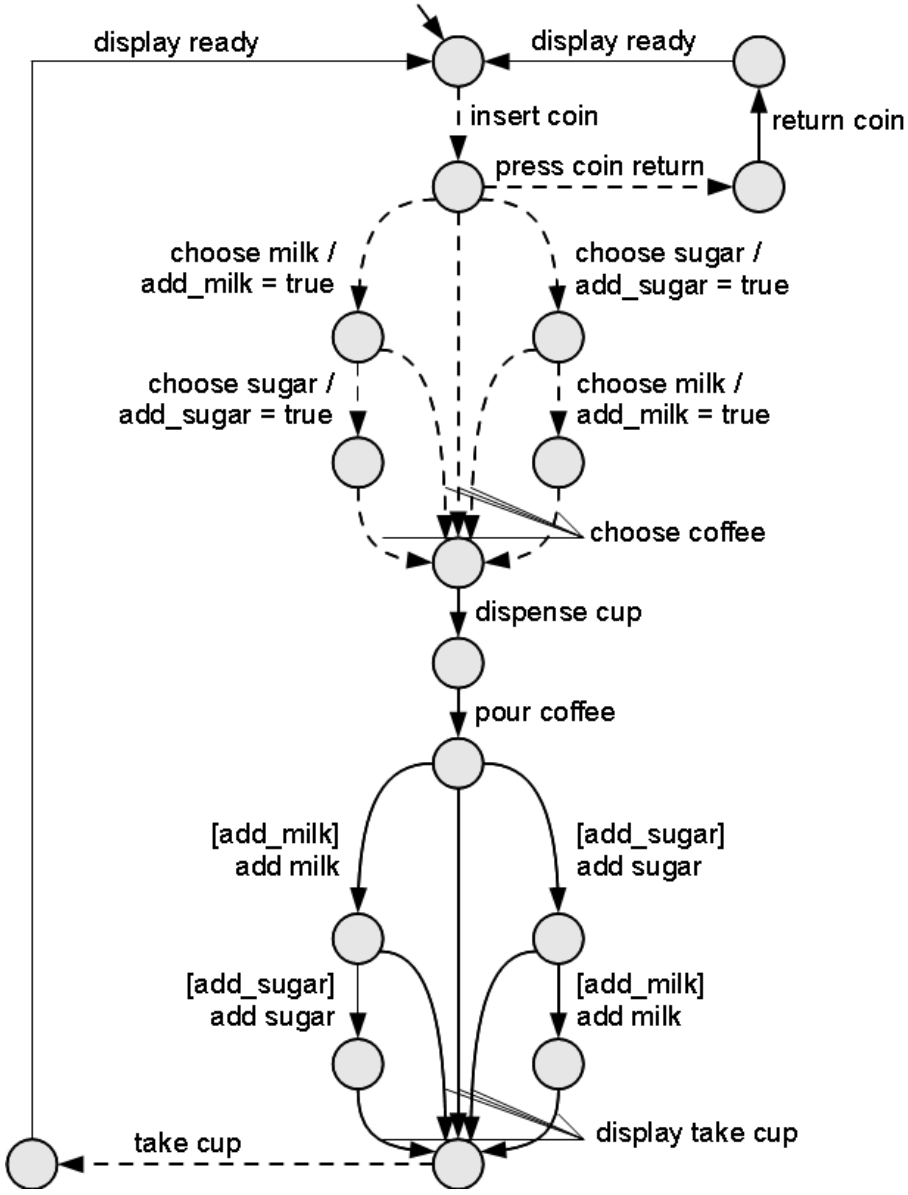
- Leider ist für nicht genau definiert, was das bedeutet
  - Soll der Benutzer auch Milch *und* Zucker wählen können?
  - Wenn ja, soll dann erst Milch oder erst Zucker in den Becher?

Use Cases → Zustandsautomaten → Tests

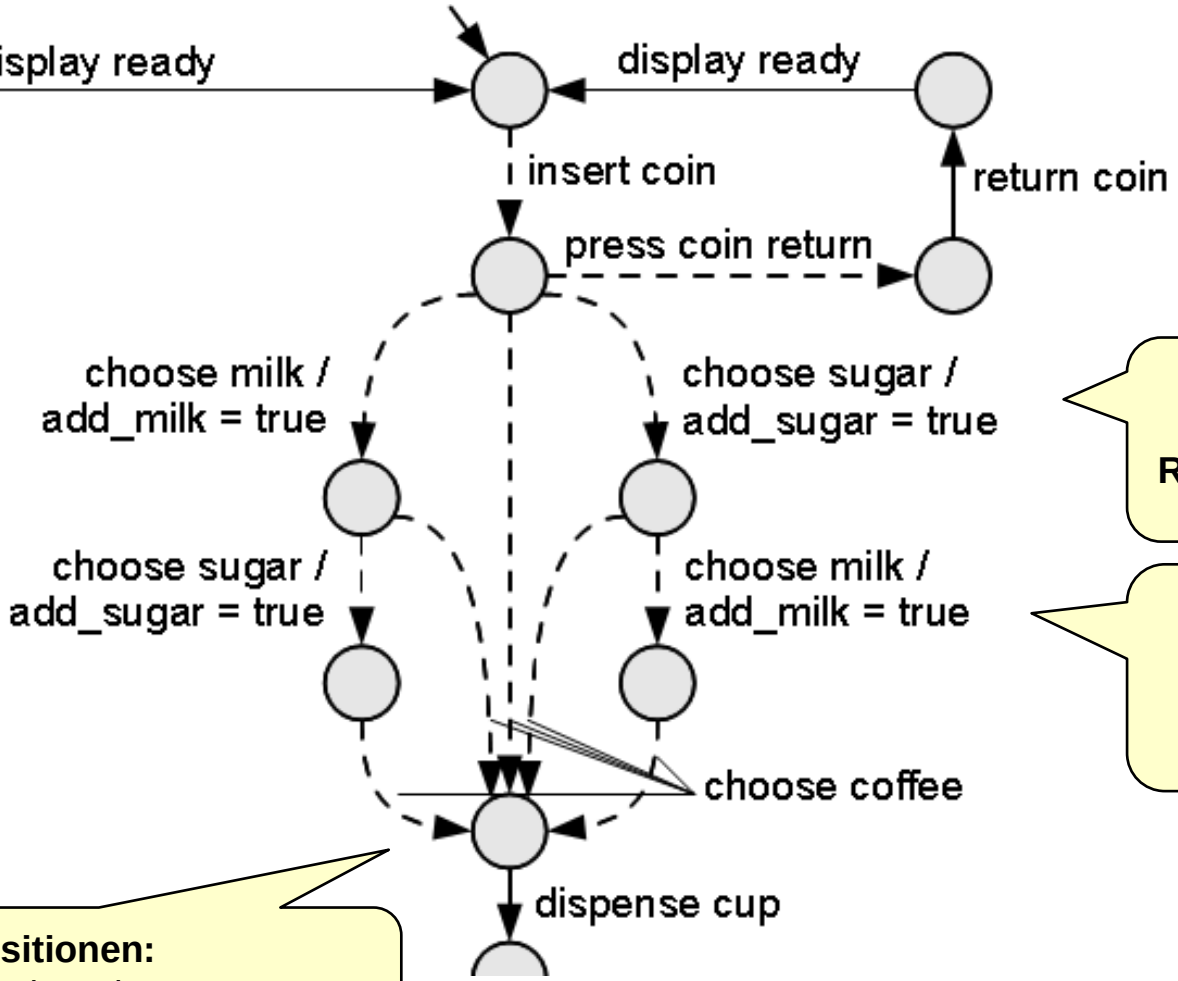
- Eine Spezifikation aus informellen Use Cases kann in ein präzises Modell des erwarteten Verhaltens überführt werden
  - Das kann sehr aufwendig sein
  - Aber oft sinnvoll um (mit dem Kunden) Anforderungen zu präzisieren
- Beispiel:

Use Case buy coffee …
Use Case buy coffee with sugar …
Use Case buy coffee with milk …
Use Case coin back …

Eine Interpretation der Use Cases



Use Cases → Zustandsautomaten → Tests



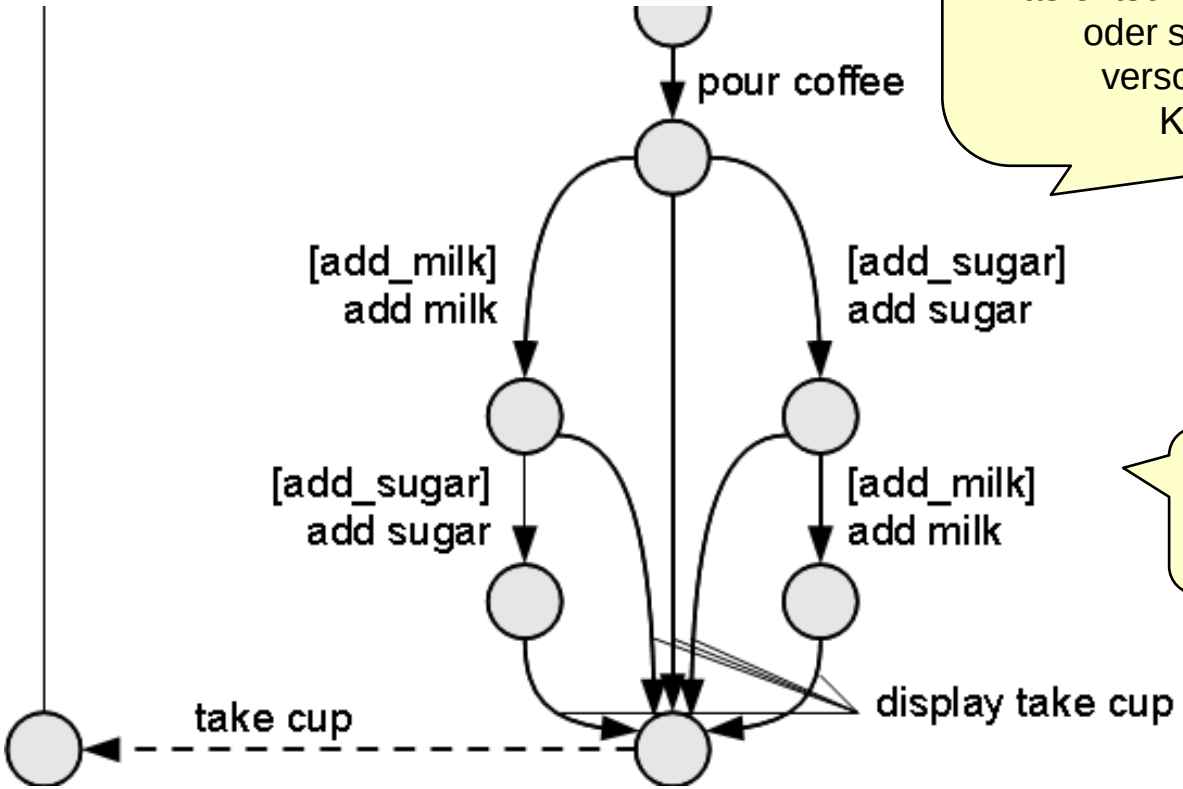
Milch und Zucker kann in beliebiger Reihenfolge gewählt werden

Automat mit Variablen, Zuweisungen und Bedingungen

Transitionen:
Gestrichelt: Eingaben des Benutzers
Durchgezogen: Aktionen des Systems

Use Cases → Zustandsautomaten → Tests

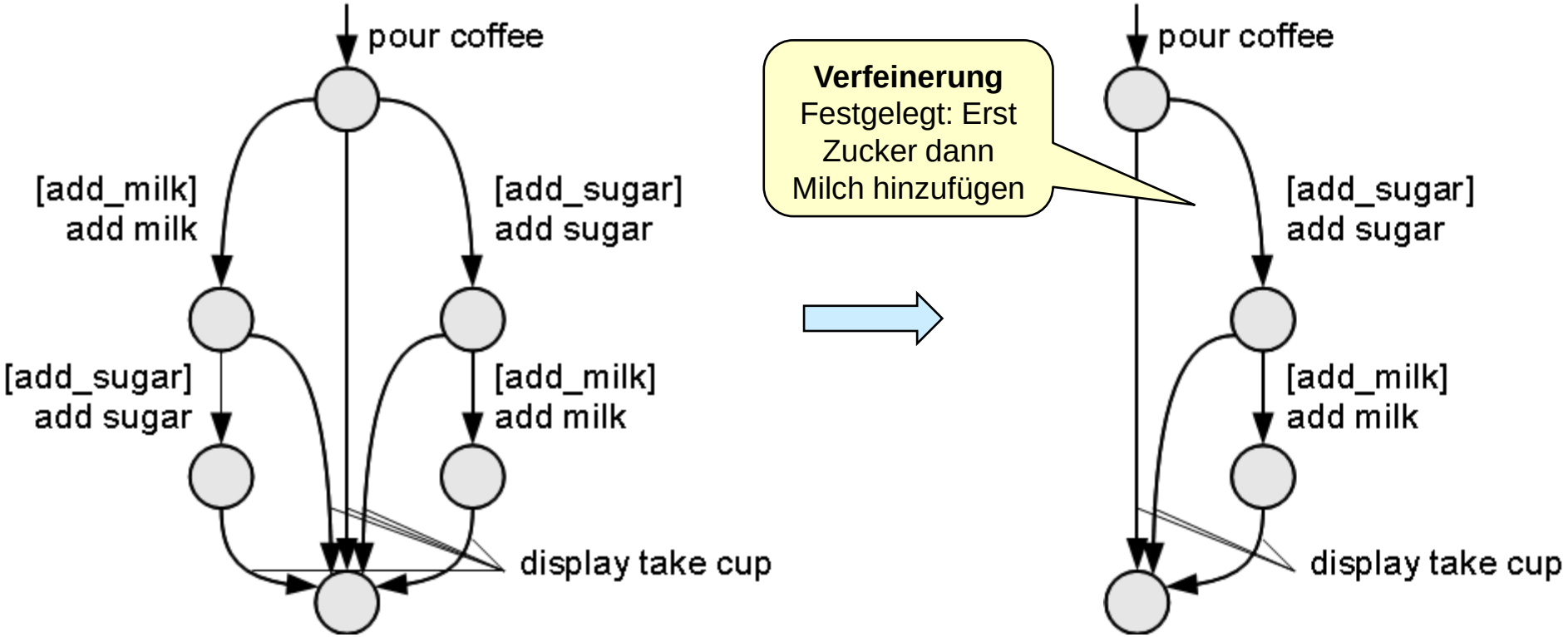
Fügt die Kaffemaschine nun zuerst Milch oder Zucker hinzu?
Die Spezifikation lässt beide Möglichkeiten zu (**Nichtdeterminismus**). Das entscheidet später jemand. (Ggf. so oder so besser, ggf. anders in verschiedene Varianten der Kaffeemaschine, …)



Bedingung
(zuvor Variable add_milk auf true gesetzt)

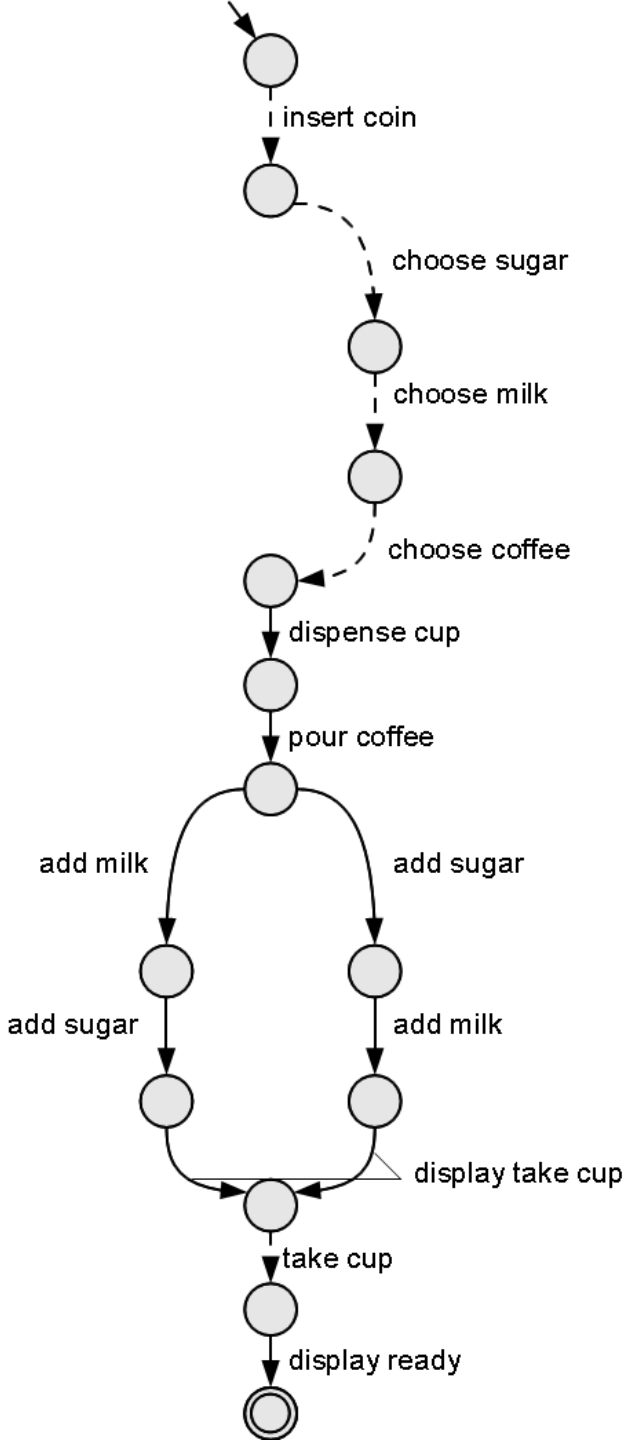
Use Cases → Zustandsautomaten → Tests

- Beim Testen ist oft bekannt, wie die Anforderungen verfeinert und implementiert wurden



Use Cases → Zustands-automaten → Tests

- Manchmal ist nicht bekannt, wie Anforderungen verfeinert wurden
- Oder das System hat **unvorhersehbares** Verhalten
  - Manche Umwelteinflüsse sind beim Test **unkontrollierbar**
  - Software verhält sich **zufällig**, z.B.
    - Thread-Scheduling, Math.random(),
    - Iteration über HashSet (Java) kann unterschiedlich sein
- Dann ist ein Tests nicht nur eine Sequenz von Ein- und Ausgaben, sondern ein gerichteter Graph
  - Verzweigung wenn unterschiedliche Reaktionen des Systems erlaubt



Zusammenfassung

- **Black-Box-Tests** – Was haben wir gemacht?
  - Tests erstellen nach Verfahren „Alle Anforderungen abdecken“
  - Äquivalenzklassenmethode
    - Definieren von Äquivalenzklassen („Wo wird sich das Programm wahrscheinlich gleichartig verhalten“)
    - Mehrere Parameter → mehrdimensionale Äquivalenzklassen
    - Verschmelzen von mehrdimensionalen Äquivalenzklassen
  - Kombinatorisches Testen
    - Abdecken aller Parameter-Interaktionen eines bestimmten Levels
    - IPOG-Algorithmus
  - Testen reaktiver/zustandsbasierter Systeme
    - Zustandsautomat dient als Spezifikation, Abdeckungskriterien basierend auf Zustandsautomat
    - Tests aus Use Cases ableiten

Softwarequalität

Vorlesung 10 – Testen: White-box-Verfahren (Kontrollflussorientierte Verfahren)

Prof. Dr. Joel Greenyer

13. Juni 2016

White-/Glass-Box-Tests

In der letzten Vorlesung…

- **White-/Glass-Box-Tests** (Wdh.)
  - Testen eines Subjekts mit Kenntnis über dessen inneren Aufbau
  - Idee: Sicherstellen, dass möglichst viele Programmteile durch Tests ausgeführt werden (**Coverage**) – In Code, der nicht ausgeführt wurde, kann auch kein Fehler gefunden werden
  - **Sollwerte** werden weiterhin **von einer Spezifikation** abgeleitet
    - Sollwerte sind prinzipiell **nicht** aus dem Code ableitbar!

- Nachteil von Glass-Box-Tests:
  - Fehler können gefunden werden, aber nicht **fehlende** Funktionen

Beispiel

- **Frage**: Welche Tests machen hier Sinn?

```
/**
 * calculate the Manhattan distance
 * of two points p1 = (x1, y1), p2 = (x2, y2)
 * by calling manhattan(x1-x2, y1-y2)
 * @param a
 * @param b
 * @return |a| + |b|
 */
public int manhattan(int a, int b){
        if (a < 0)
                a = -a;
        if (b < 0)
                b = -b;
        return a+b;
}
```