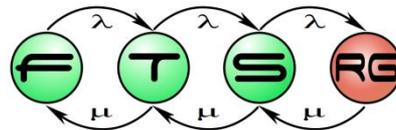


Überprüfung der Modelle

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group



Ariane 5 Trägerrakete

- Die leistungsfähigste europäische Trägerrakete



Ariane 5 Trägerrakete

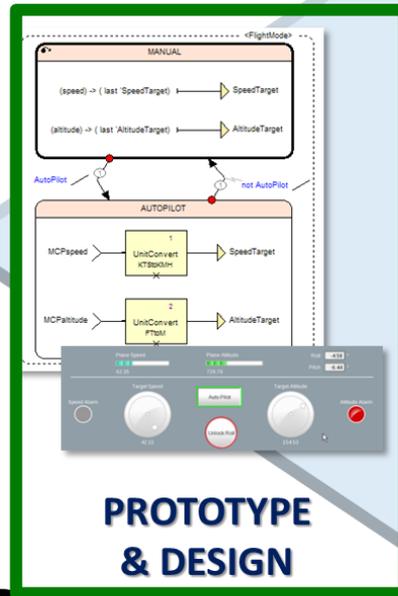
- Am 4. Juni 1996 hat sich die Rakete 37 Sekunden nach dem Start zerstört
 - die gelieferten vier Cluster-Satelliten sind auch zerstört worden
 - \$370 Millionen Verlust



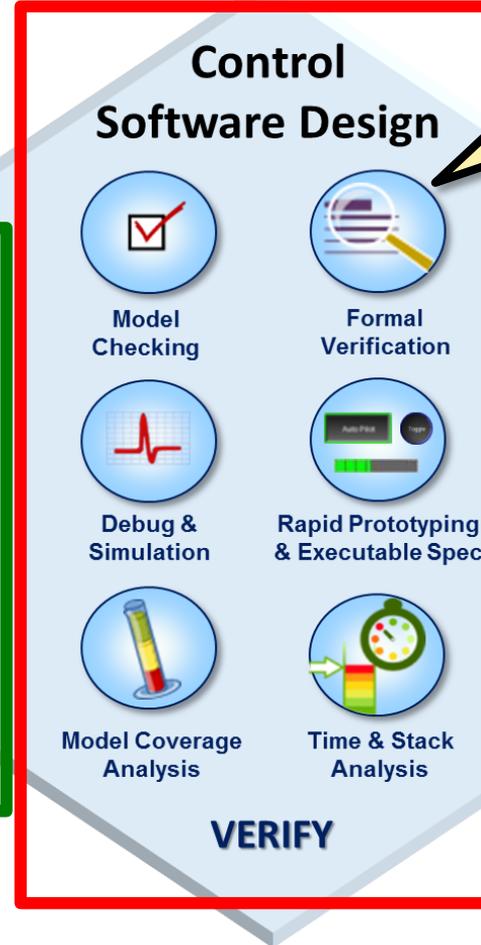
Ariane 5 Trägerrakete

- Am 4. Juni 1996 hat sich die Rakete 37 Sekunden nach dem Start zerstört
 - die gelieferten vier Cluster-Satelliten sind auch zerstört worden
 - \$370 Millionen Verlust
- Einer der teuersten Softwarefehler der Welt
 - primäre Ursache:
 - erfolglose Konversion einer Zahl von 64 Bit zu 16 Bit
 - sekundäre Ursache:
 - Die Modulen wurden nie zusammen getestet**

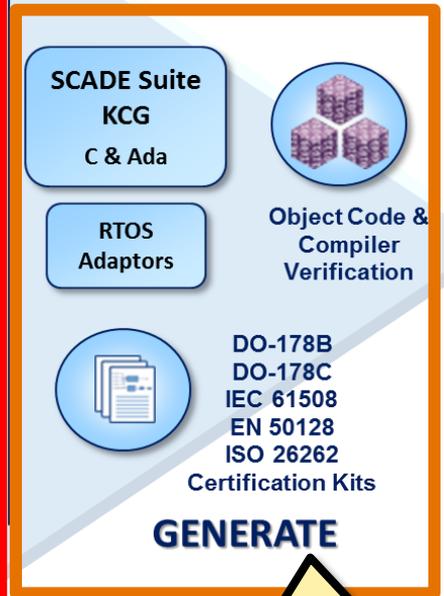
Beispiel: Esterel SCADE



**PROTOTYPE
& DESIGN**



**Überprüfung
der Modelle**



Flugzeuge,
kritische ES,
Automotive

Spezifikation

Modelle

Überprüfung

Generierung

Grundbegriffe

Stat. Überprüfung

Testen

Formale Verifikation

INHALT

Lebenszyklus der Modelle

Entwicklung
von Modellen

Software-
Entwicklung

Anforderungen,
Spezifikation

Anforderungen,
Spezifikation

Ausgangsmodelle

Entwurf

detailliertere
Modelle

Implementation

Überprüfung

Testen

Wartung

Wartung

```
97 m_FKX = fcos(PositiveInfinity);  
98 return;  
99 }  
100 m_FKX = (ro / (1-ro)) * (1- (ro*(float-1)  
101 m_FKX = (lambda*lambda/(k*mu*mu) * ro*ro)  
102 m_FKX = m_FKX / lambda;  
103 m_FKX = fcos(PositiveInfinity);  
104 m_FKX = fcos(PositiveInfinity);  
105 m_FKX = fcos(PositiveInfinity);  
106 }  
107 }  
108 }  
109 }  
110 }  
111 }  
112 }  
113 }  
114 }  
115 }  
116 }  
117 }  
118 }  
119 }  
120 }  
121 }  
122 }  
123 }  
124 }  
125 }  
126 }  
127 }  
128 }  
129 }  
130 }  
131 }  
132 }  
133 }  
134 }  
135 }  
136 }  
137 }  
138 }  
139 }  
140 }  
141 }  
142 }  
143 }  
144 }  
145 }  
146 }  
147 }  
148 }  
149 }  
150 }  
151 }  
152 }  
153 }  
154 }  
155 }  
156 }  
157 }  
158 }  
159 }  
160 }  
161 }  
162 }  
163 }  
164 }  
165 }  
166 }  
167 }  
168 }  
169 }  
170 }  
171 }  
172 }  
173 }  
174 }  
175 }  
176 }  
177 }  
178 }  
179 }  
180 }  
181 }  
182 }  
183 }  
184 }  
185 }  
186 }  
187 }  
188 }  
189 }  
190 }  
191 }  
192 }  
193 }  
194 }  
195 }  
196 }  
197 }  
198 }  
199 }  
200 }
```



Automatische Kodengenerierung

Entwicklung
von Modellen

Software-
Entwicklung

Anforderungen,
Spezifikation

Anforderungen,
Spezifikation

Ausgangsmodelle

detailierte Modelle

korrektes Modell, korrekterer Code

Implementation

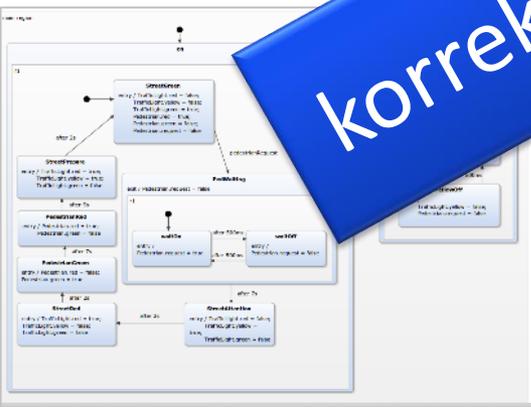
Überprüfung

Testen

Wartung

Wartung

```
import java.util.Hashtable;
import java.awt.*;
public class InternetClient implements Serializable
{
    private Socket socket;
    private ObjectOutputStream objectIn;
    private ObjectInputStream objectOut;
    // Creates a connection to the InternetClient
    public InternetClient(int port, String message)
    {
        // Connection to
        // To read object
        // To write object
    }
    // Int
    // Int
    // Mess
    // port supplied
}
```



Grundbegriffe

Stat. Überprüfung

Testen

Formale Verifikation

GRUNDBEGRIFFE

Modelle und Aufgaben

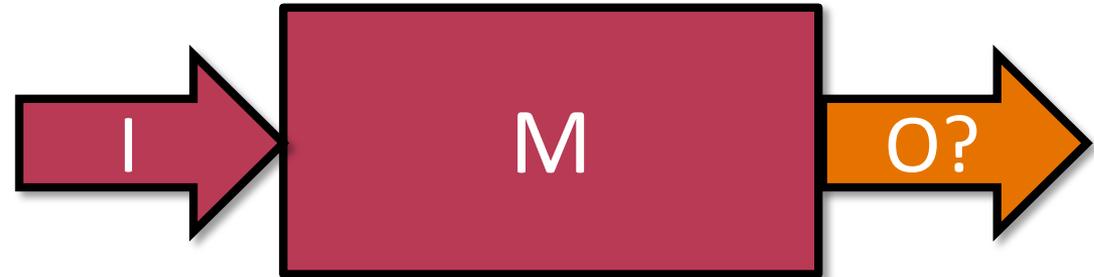
- **Synthese:**

*Welches Modell
entspricht der Spezifikation?*



- **Analyse:**

*Wie ist das Verhalten
des Modelles?*



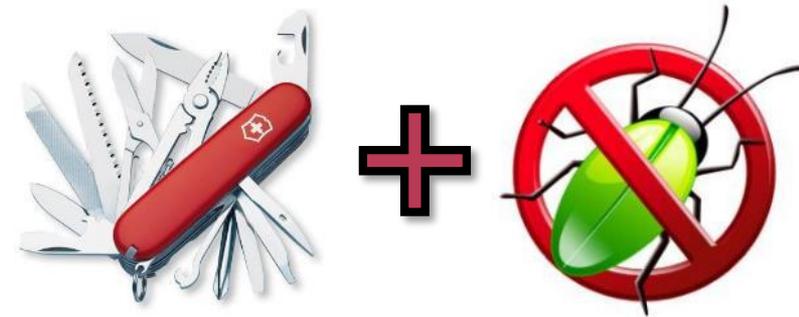
- **Steuerung:**

*Wie ist der gewünschte
Zustand erreichbar?*



Korrektheit

- Die **Korrektheit** ist die Fähigkeit des Modells oder Codes, dass es den gegenüber ihm gestellten Anforderungen entspricht.
 - **funktionale Korrektheit:** Das Entsprechen den funktionalen Anforderungen
 - Überprüfung der nichtfunktionalen Anforderungen (siehe die Vorlesungen über Leistungsmodellierung)
- **Aspekte:**
 - Erfüllung der Aufgabe
 - Fehlerfreiheit, Sicherheit
 - Kein verbotenes Verhalten



Klassifizierung der funktionalen Anforderungen

- **Erlaubtes Verhalten** (z.B. safety)
 - Welche Zustände darf das System (nicht) aufnehmen?
 - Was für Verhalten sind verboten?
 - Universelle Anforderungen
 - sollen immer wahr sein
- **Erwartetes Verhalten** (z.B. liveness)
 - Welche Zustände sollen erreichbar sein?
 - Welche Funktionen soll das System leisten können?
 - Existenzielle Anforderungen
 - sollen immer erreichbar sein

Klassifizierung der funktionalen Anforderungen

■ Erlaubtes Verhalten (z.B. safety)

- Welche Zustände darf das System (nicht) aufnehmen?
- Was für Verhalten sind erlaubt?
- Universelle Anforderungen
 - sollen immer wahr sein

„Die Verkehrsampeln einer Kreuzung **dürfen nie** gleichzeitig grün sein.“

■ Erwartetes Verhalten (z.B. usability)

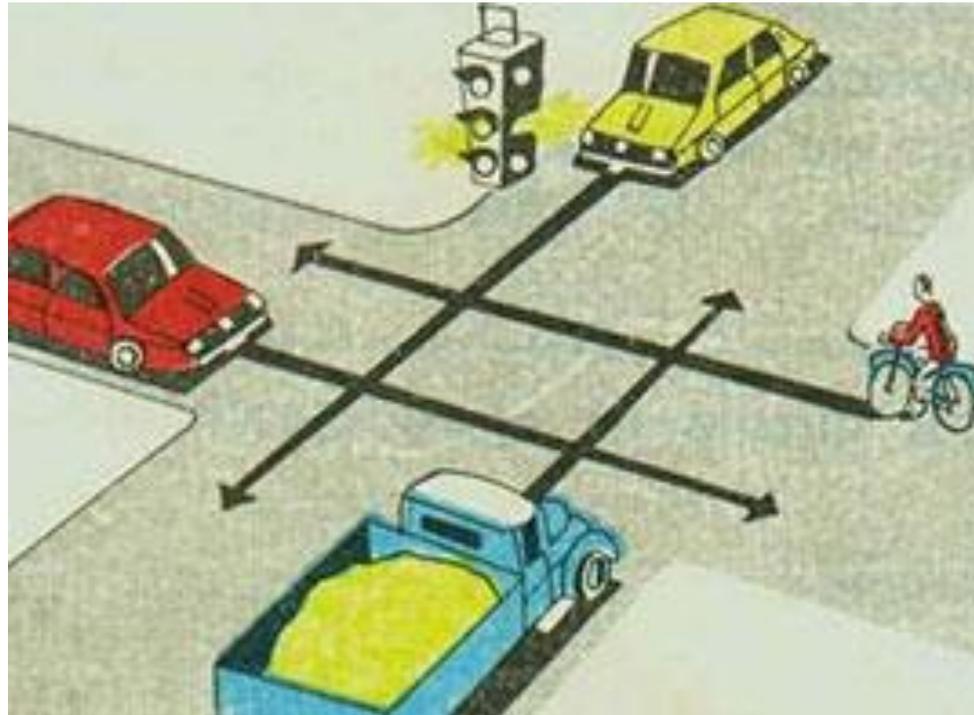
- Welche Zustände sollen erreicht werden können?
- Welche Funktionen soll das System ausführen?
- Existenzielle Anforderungen
 - sollen immer erreichbar sein

„Die Verkehrsampel **soll immer fähig sein** auf grün zu wechseln.“

Verklemmung (deadlock)

Verklemmung: Eine Zustandsmenge, aus welcher das System ohne äußeren Eingriff nicht weiterrreten kann.

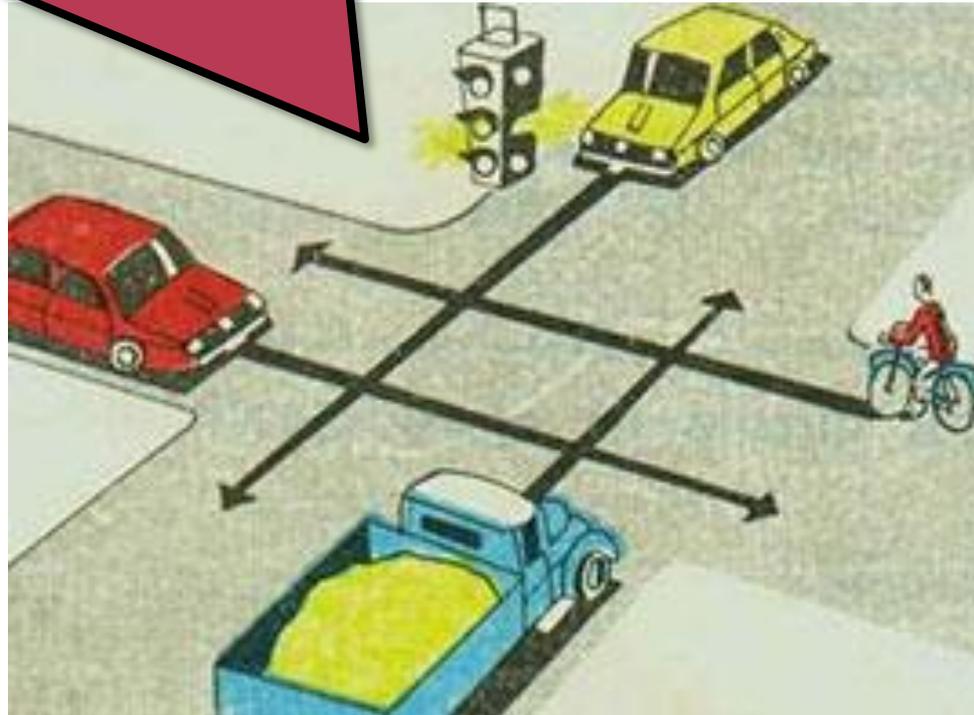
- z.B. aufeinander wartende Prozesse



Verklemmung (deadlock)

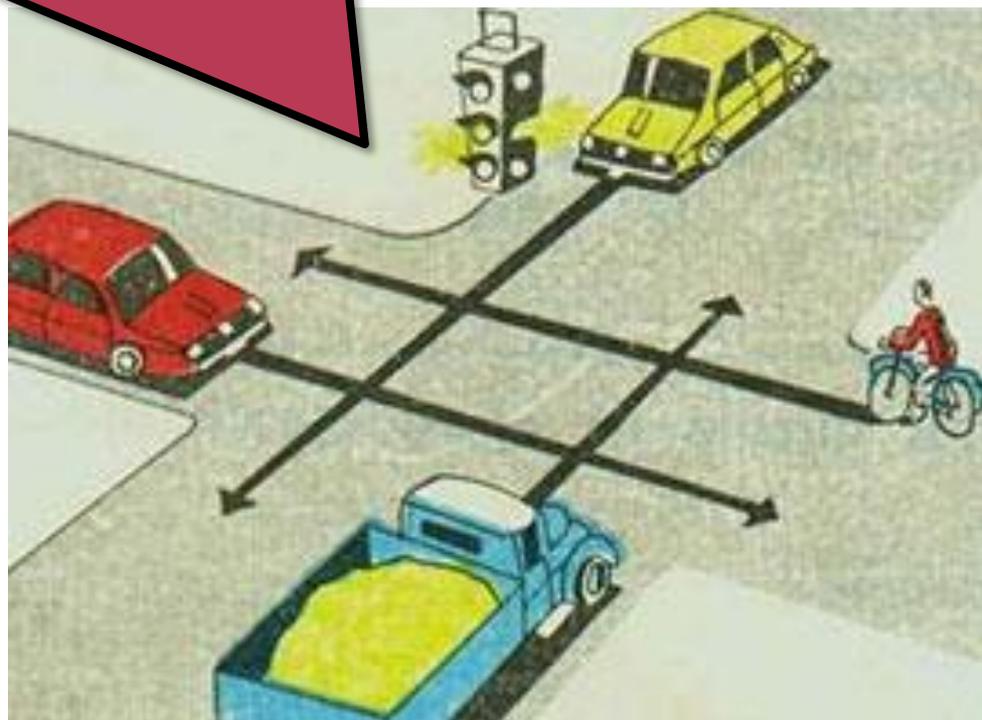
„Fahrzeuge, die von rechts kommen, haben, sofern die folgenden Absätze nichts anderes bestimmen, den Vorrang; Schienenfahrzeuge jedoch auch dann, wenn sie von links kommen.“

(Straßenverkehrsordnung §19 Abs. 1)



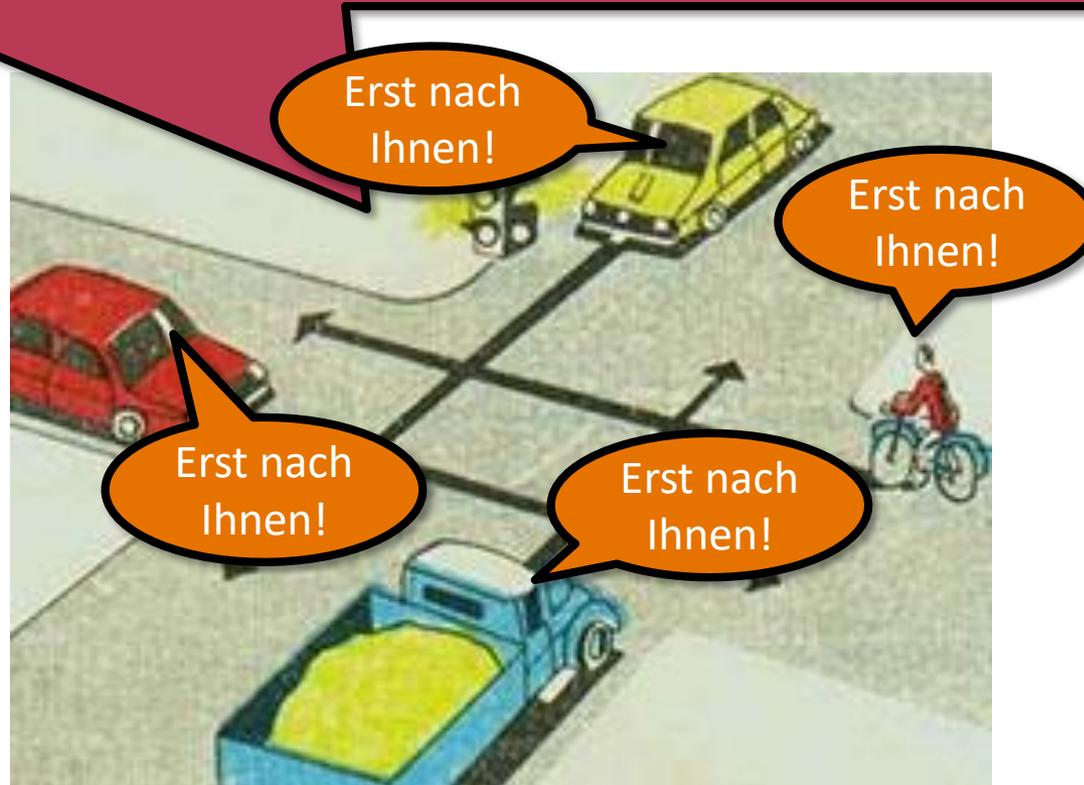
Auflösung einer Verklemmung

„Wenn 4 Fahrzeuge in einer Kreuzung mit Rechtsvorrang gleichzeitig ankommen, soll irgendeiner der Fahrer auf seinen Vorrang verzichten. Ansonsten werden sie laut StVO ewig stehenbleiben.“
(gyakorikerdesek.hu)



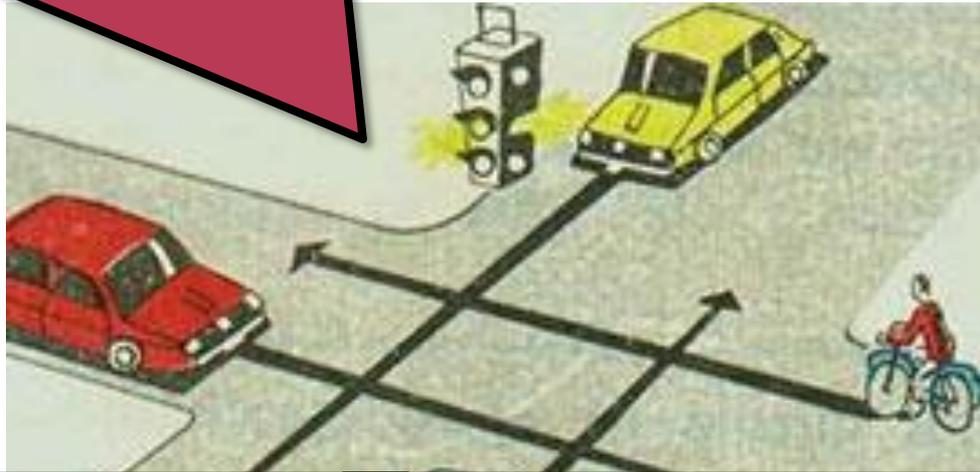
Auflösung einer Verklemmung

„Wenn 4 Fahrzeuge in einer Kreuzung mit Rechtsvorrang gleichzeitig ankommen, soll irgendeiner der Fahrer auf seinen Vorrang verzichten. Ansonsten werden sie laut StVO ewig stehenbleiben.“
(gyakorikerdesek.hu)



Neue Verklemmung

„Wenn 4 Fahrzeuge in einer Kreuzung mit Rechtsvorrang gleichzeitig ankommen, soll irgendeiner der Fahrer auf seinen Vorrang verzichten. Ansonsten werden sie laut StVO ewig stehenbleiben.“
(gyakorikerdesek.hu)



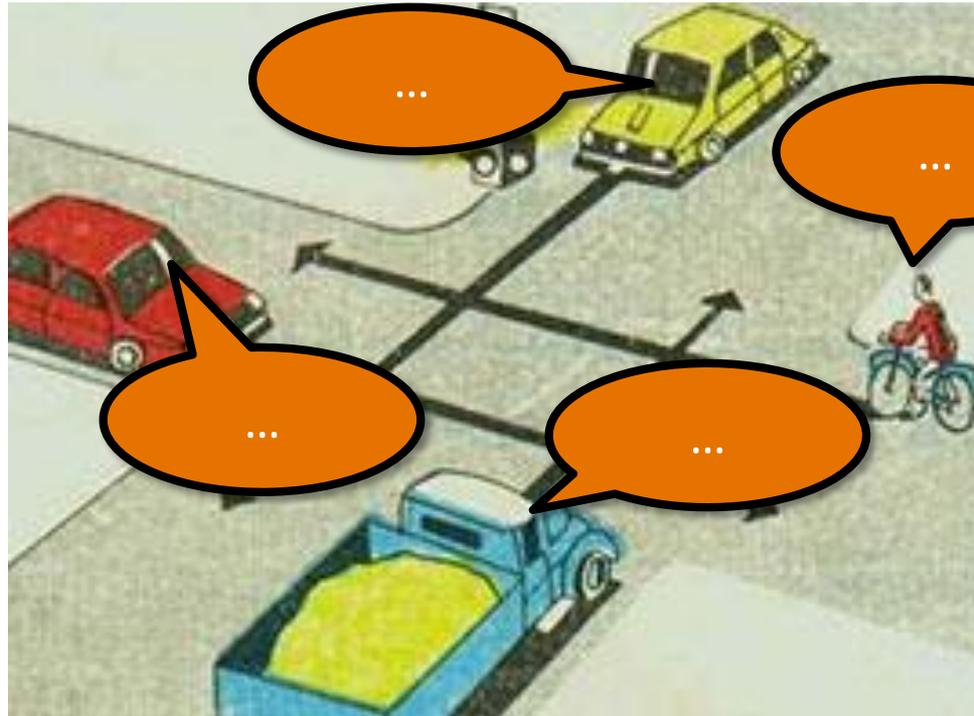
Auflösung der Verklemmung der Auflösung:

- Asymmetrische Algorithmen
- Algorithmen mit Zufallsprinzip
 - Siehe den Backoff-Verfahren bei Ethernet-Netzwerken

Endlosschleife (livelock)

Endlosschleife: Eine weitere Zust.menge, aus welcher das System ohne äußeren Eingriff nicht weiterrreten kann.

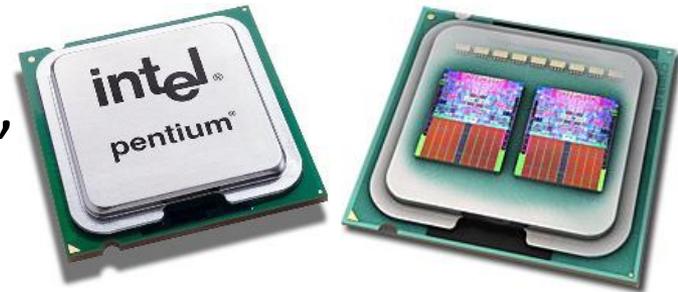
- z.B. gerade wegen der Auflösung einer **Verklemmung**
- z.B. das Google-Auto mit dem Fixie



Verklemmung (deadlock)

■ Häufiger Planungsfehler bei parallelen Systemen

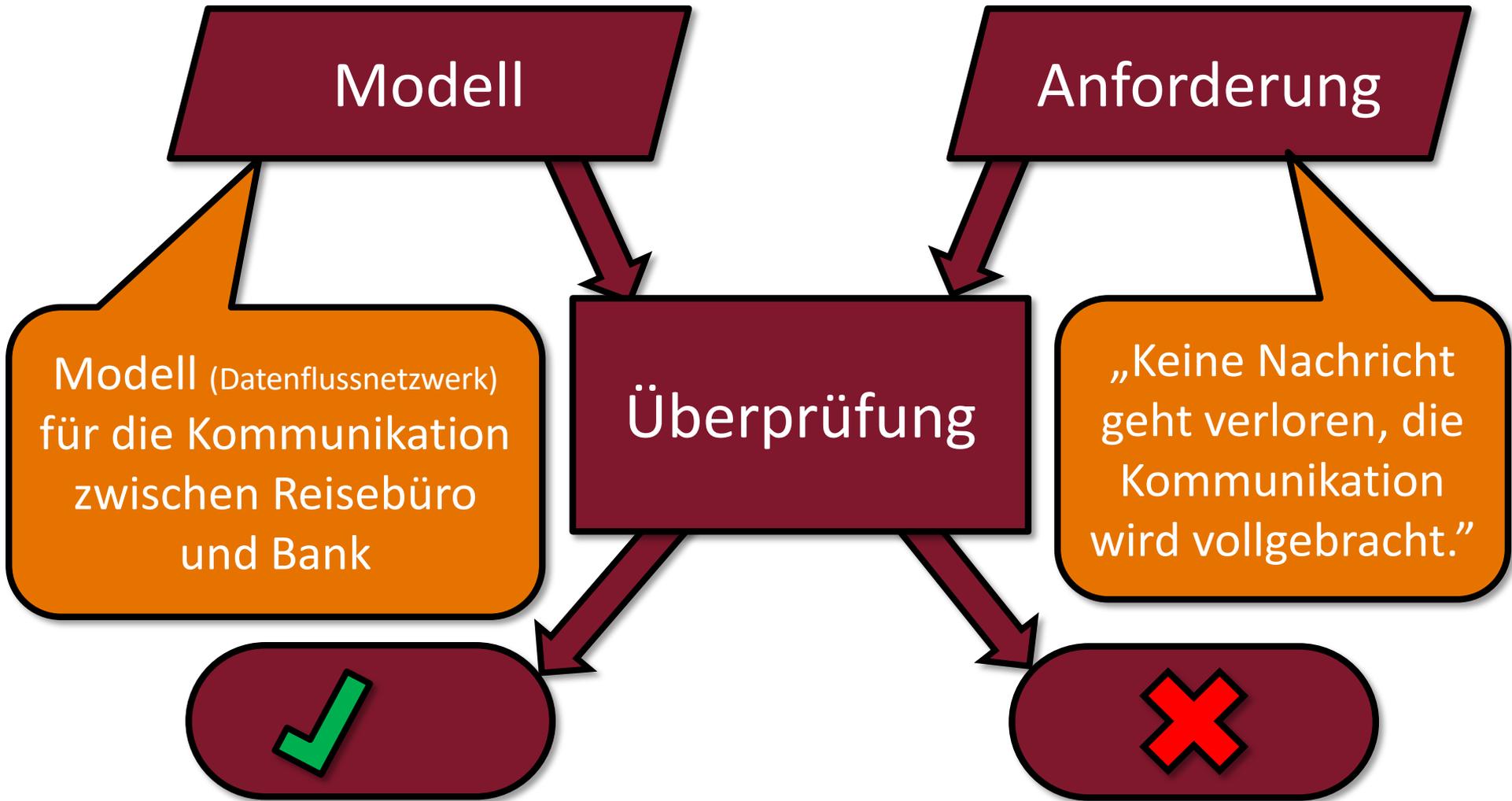
- Manchmal ist es schwer zu vermeiden, aufzulösen
 - eine für gut gehaltene Lösung kann auch so ein Problem verursachen
- Durch Testen schwer zu entdecken, kann scheinbar zufällig sein
- „Krise der Multi-core Prozessoren“



■ Beispiele

- Zwei Prozesse wollen Nachrichten austauschen, beide warten auf den anderen
- Zwei Prozesse brauchen zwei Ressourcen, jeder hat eine schon bekommen, jeder wartet auf die andere R.

Überprüfung von Modellen



Arten der Untersuchungen

■ Nach dem Zweck:

○ **Verifikation: Ist das System richtig gebaut?**

- Ist die Implementation entsprechend der Spezifikation?

○ **Validation: Ist das richtige System gebaut?**

- Erfüllt das System die Benutzeranforderungen?

■ Nach der Methode:

○ Statische Überprüfung

○ Dynamische Überprüfung

- stichprobenartig (Testen, Simulation)
- ausführlich/vollständig (Überprüfung der Modelle)

Grundbegriffe

Statische Überprüfung

Testen

Formale Verifikation

Grundbegriffe

Stat. Überprüfung

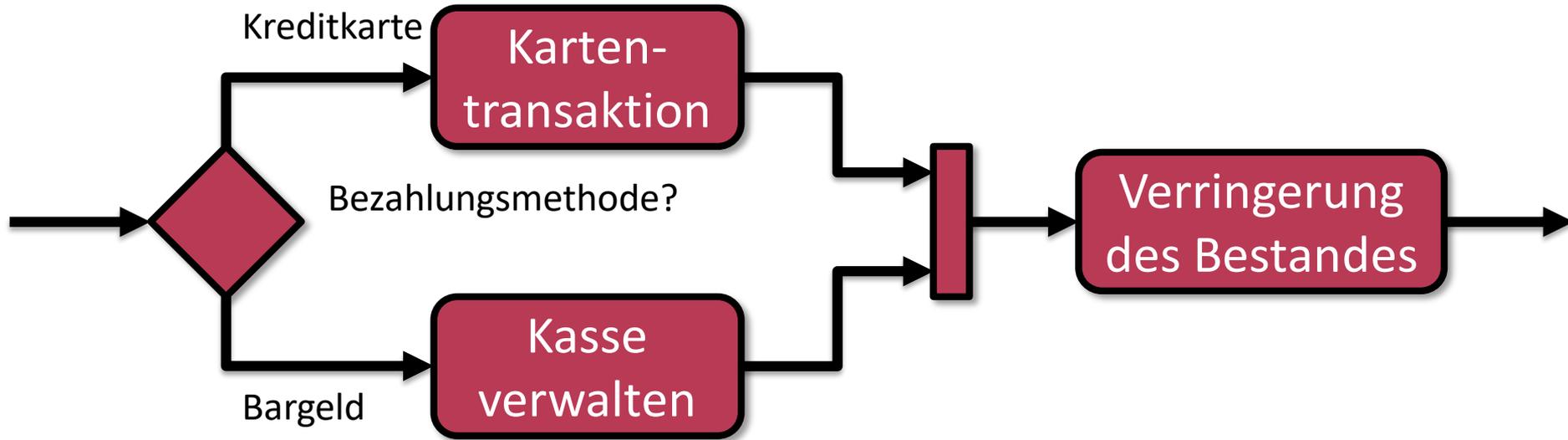
Testen

Formale Verifikation

STATISCHE ÜBERPRÜFUNG

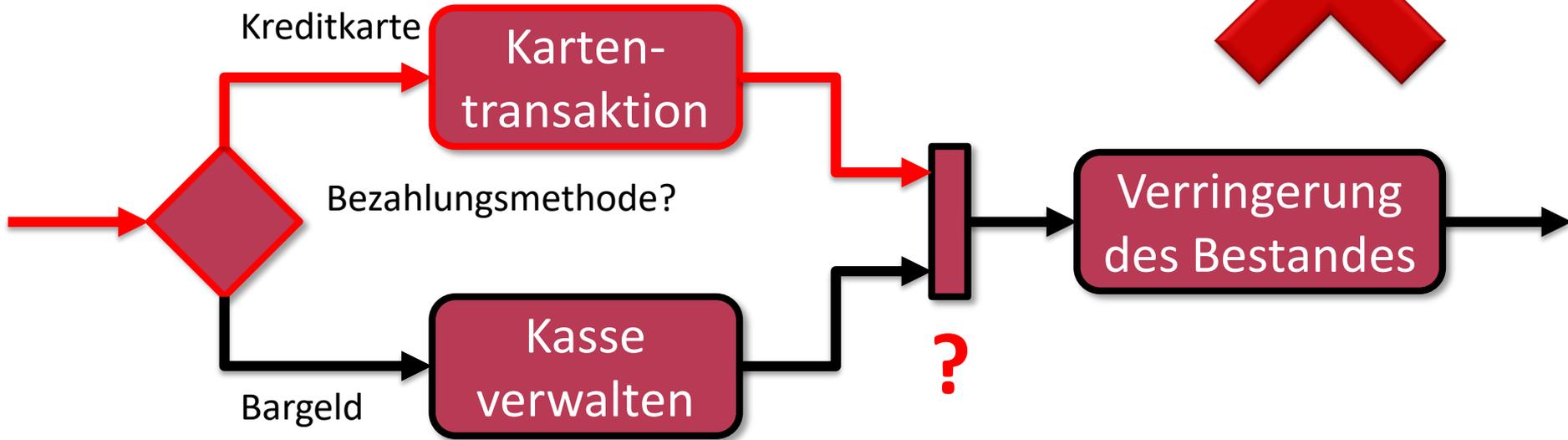
Decision und Join

- Ist das untenstehende Modell richtig?



Decision und Join

- Ist das untenstehende Modell richtig?

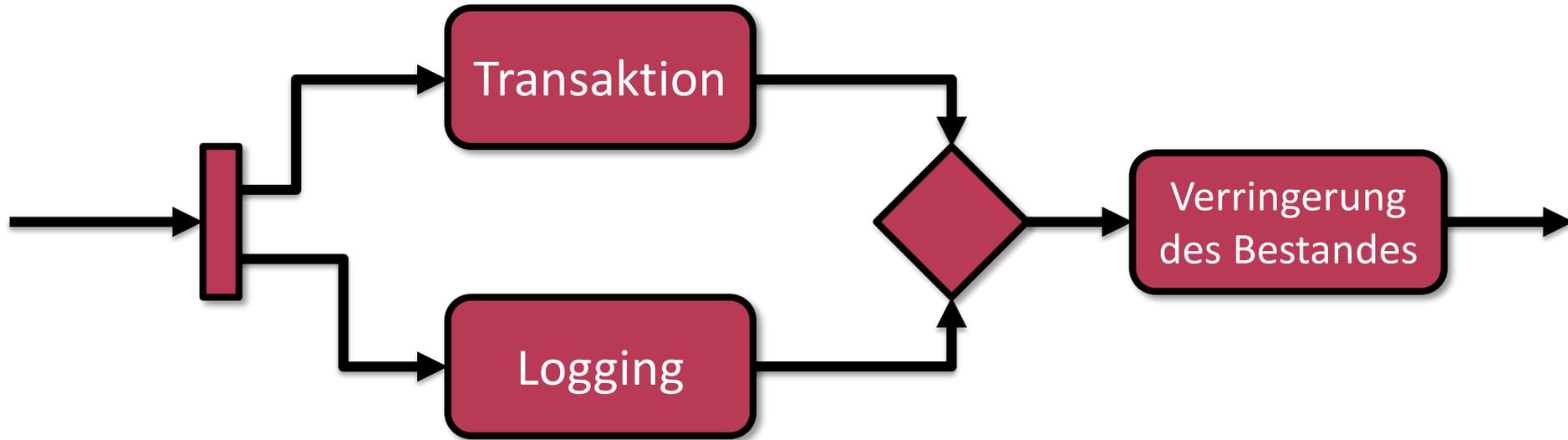


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

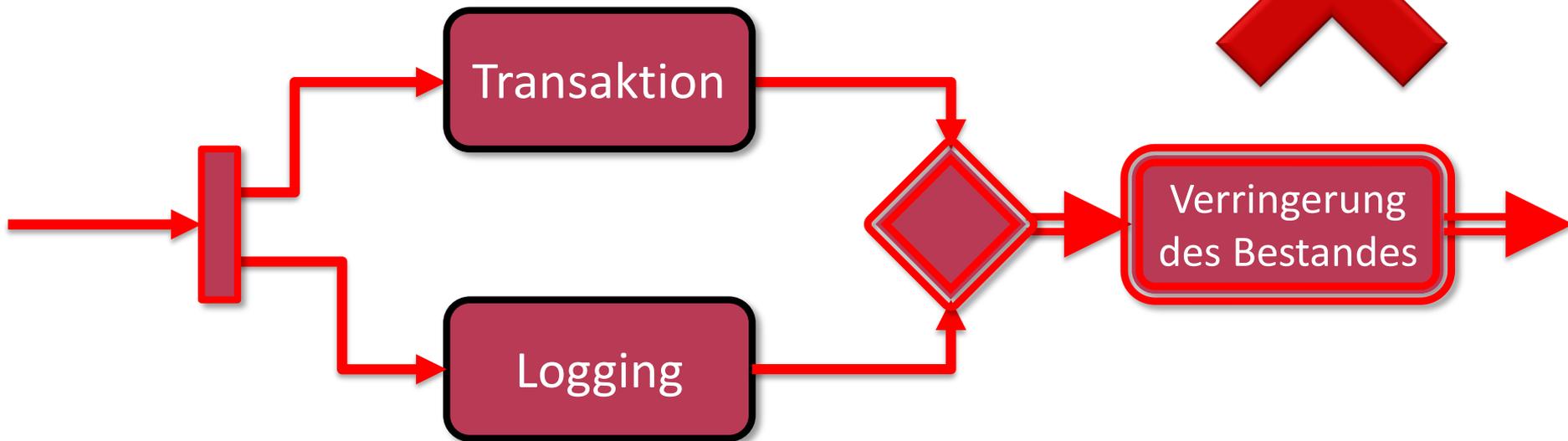
Fork und Merge

- Ist das untenstehende Modell richtig?



Fork und Merge

- Ist das untenstehende Modell richtig?



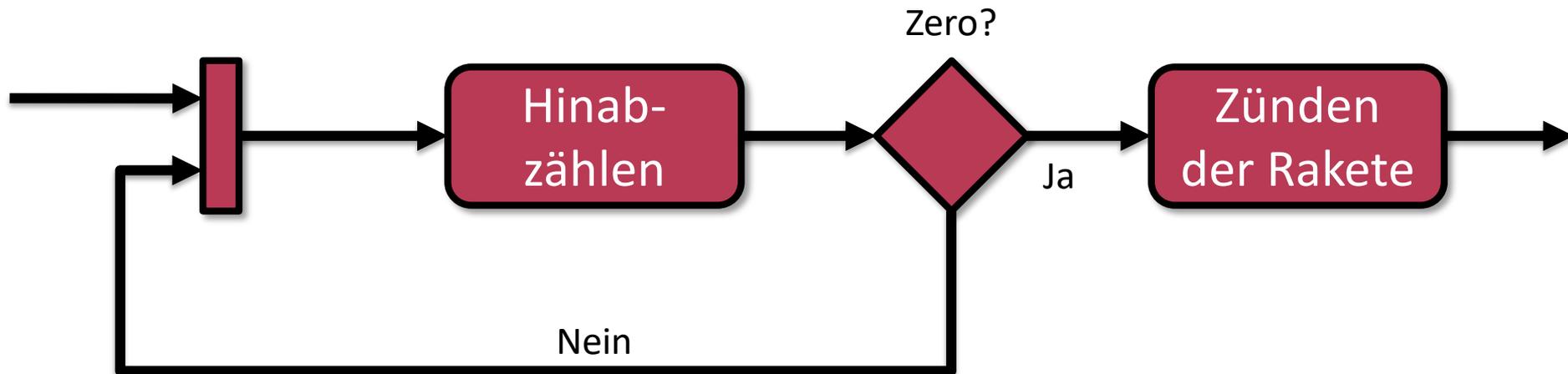
- Merge: Eine Fortsetzung ist möglich, sobald an einem Eingang ein Token angekommen ist

- Keine Synchronisation der Faden

→ **DOPPELTE „VERRINGERUNG DES BESTANDES“**

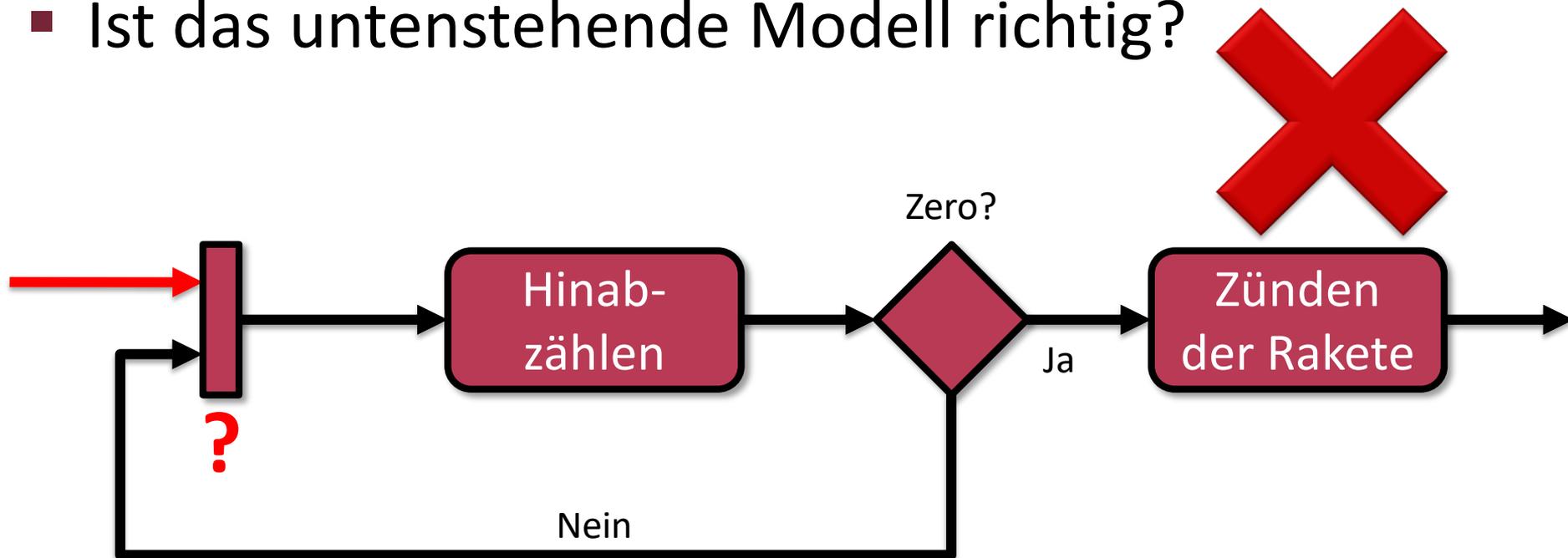
Schleife 1.

- Ist das untenstehende Modell richtig?



Schleife 1.

- Ist das untenstehende Modell richtig?

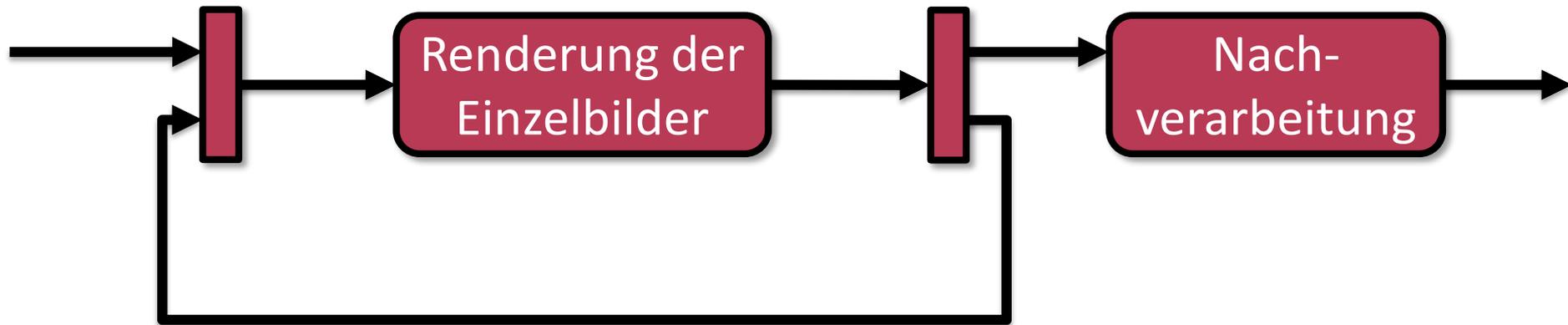


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

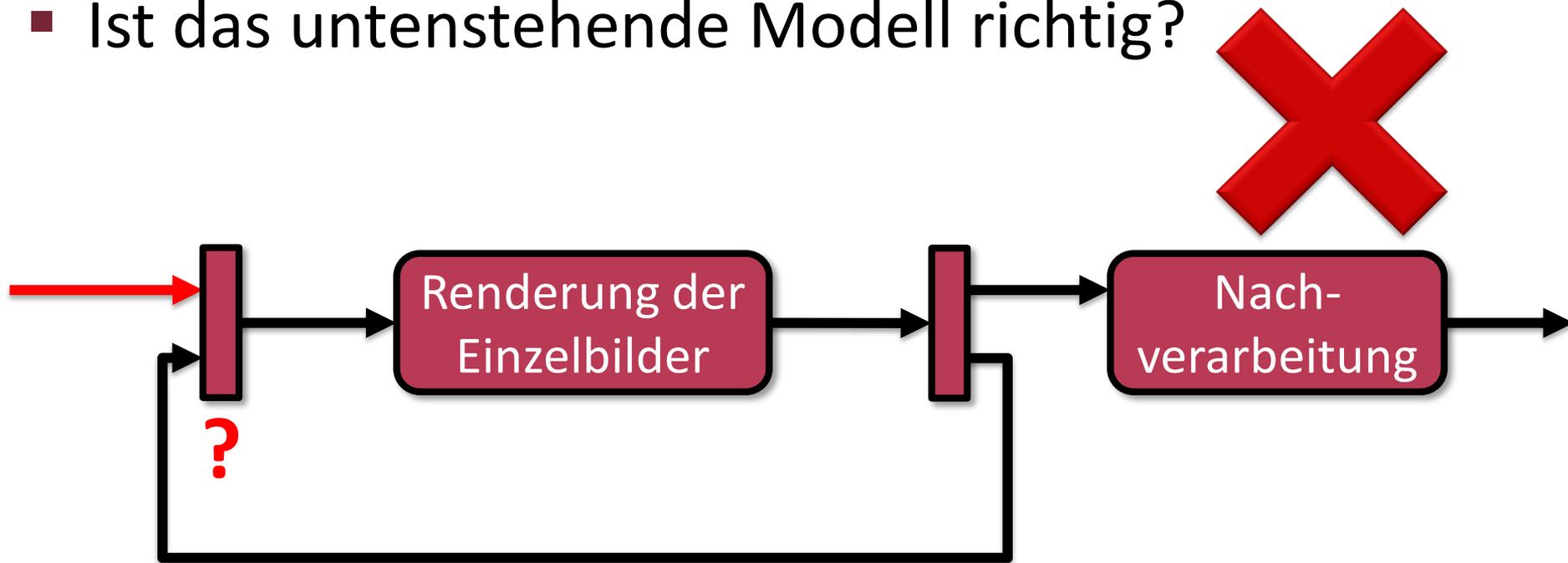
Schleife 2.

- Ist das untenstehende Modell richtig?



Schleife 2.

- Ist das untenstehende Modell richtig?

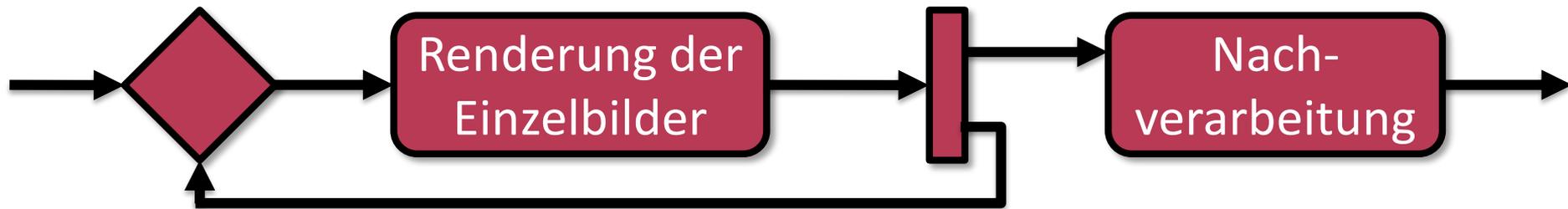


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

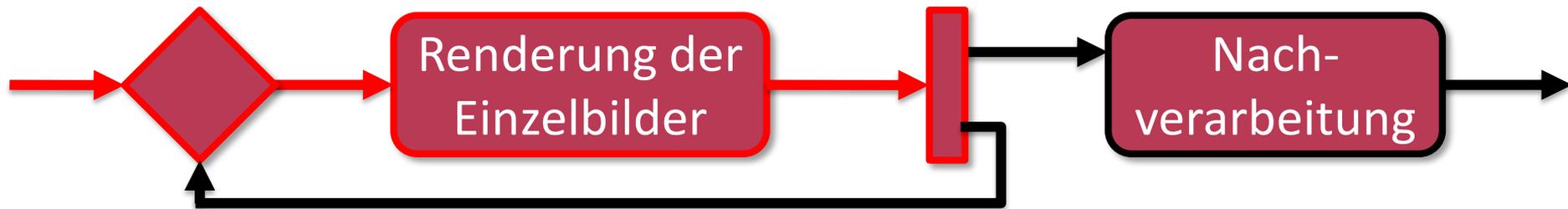
Schleife 3.

- Ist das untenstehende Modell richtig?



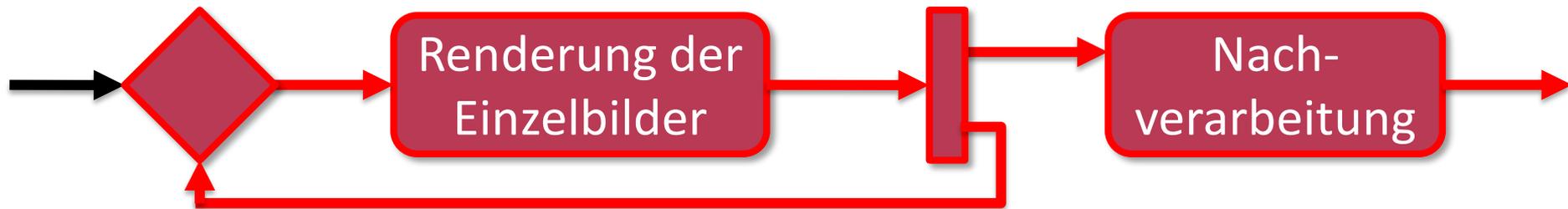
Schleife 3.

- Ist das untenstehende Modell richtig?



Schleife 3.

- Ist das untenstehende Modell richtig?

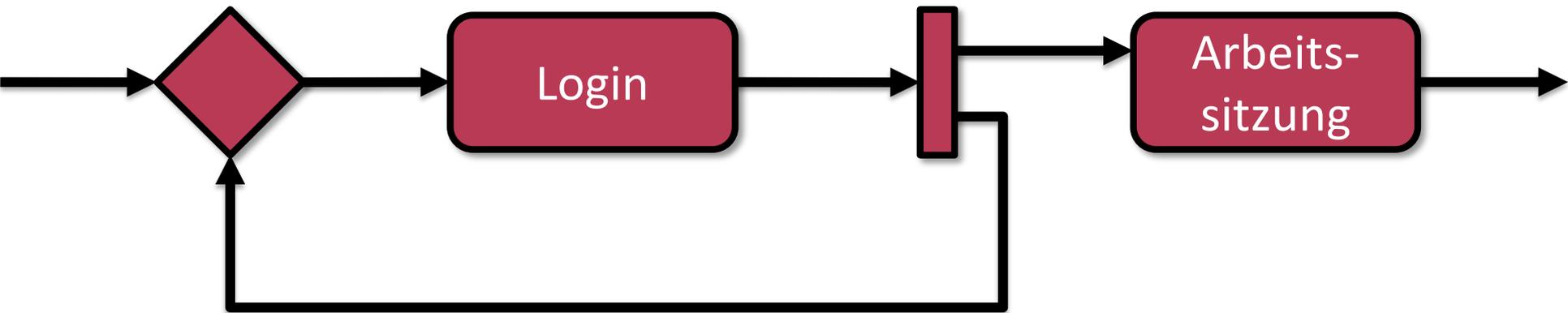


- Ein neues Bild in jeder Iteration
 - Nachverarbeitung jedes Bildes in neuem Faden (Wie viel mal?)

→ **GEFÄHRLICH**, WEIL ES UNENDLICH VIELE FADEN PRODUZIEREN KANN

Ciklus 3.

- Ist das untenstehende Modell richtig?
 - Und jetzt?

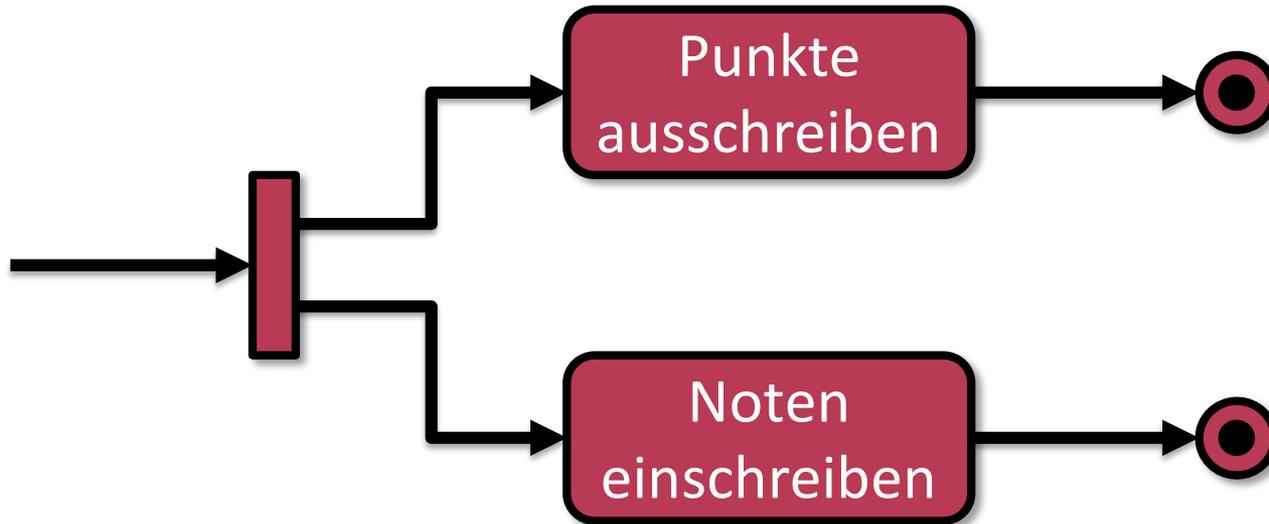


- Nach jedem Login ein weiteres Login ...
 - ...und eine Arbeitssitzung (working session) ...?

→ Die fehlerhafte Implementierung „produziert“ Threads

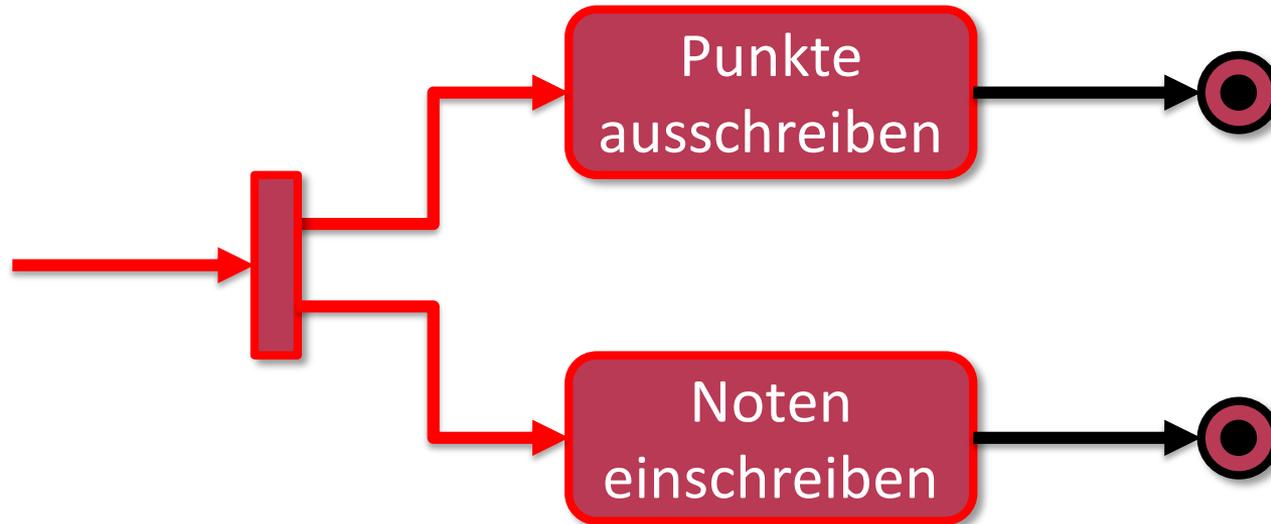
Endknotenpunkte

- Ist das untenstehende Modell richtig?



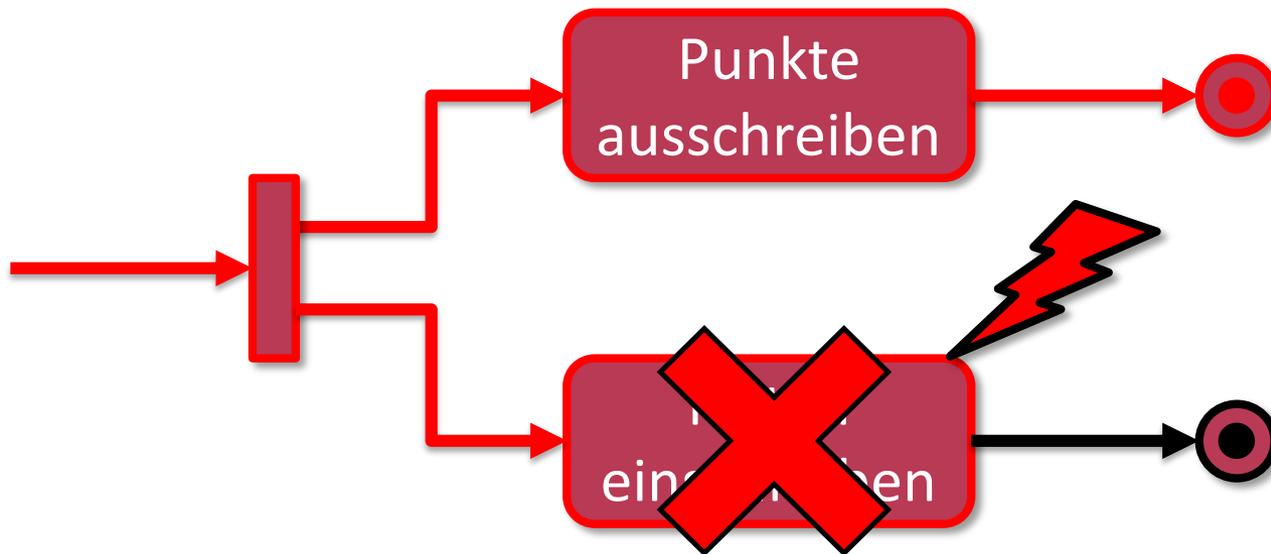
Endknotenpunkte

- Ist das untenstehende Modell richtig?



Endknotenpunkte

- Ist das untenstehende Modell richtig?

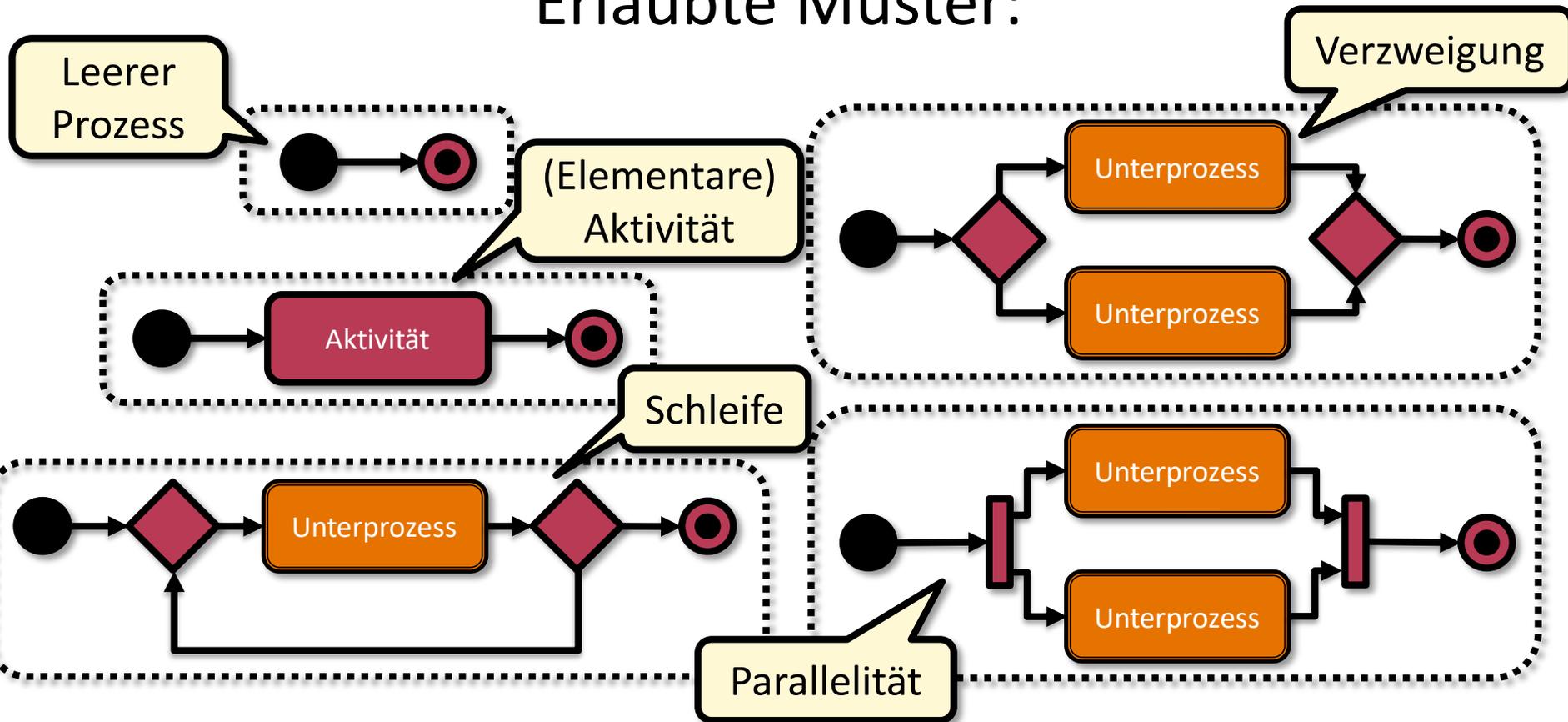


- Der terminierende Knoten stoppt den **ganzen** Prozess
→ **DIE ANDERE AKTIVITÄT WIRD NIE ABLAUFEN**

Wohlstrukturierte Prozessmodelle

- **Die Lehre:** Mit wohlstrukturierten Modellen können diese Fehler vermieden werden

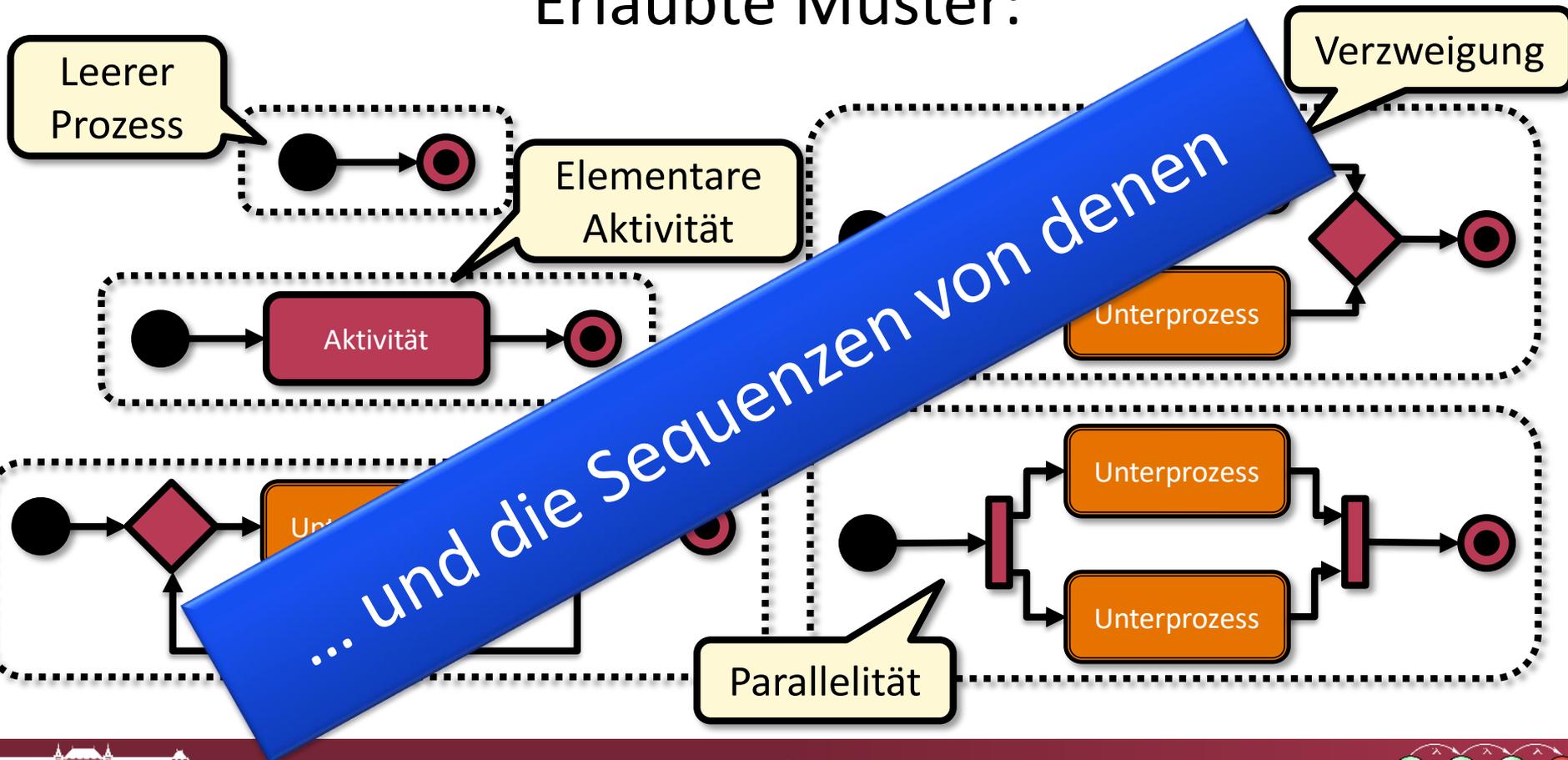
Erlaubte Muster:



Wohlstrukturierte Prozessmodelle

- **Die Lehre:** Mit wohlstrukturierten Modellen können diese Fehler vermieden werden

Erlaubte Muster:



Statische Überprüfung der Datenbearbeitung

- Eine Funktion multipliziert zwei ganze Zahlen
 - abgeleitete Anforderung:
 - „Ist mind. die eine Zahl gerade, so ist auch das Produkt“
 - Sie kann durch den ganzen Code begleitet werden
 - „Ausführung im Kopf“
- **Symbolische Ausführung**
 - Anstatt den konkreten Werten wird das Programm mit Wertemengen ausgeführt
 - Die interessante Eingabewerte werden bestimmt
 - z.B. wo es interne Verzweigungen gibt
 - Welche Eingaben steuern die einzelnen Zweige an?

Statische Überprüfung: Syntaktische Überprüfung

- Syntaktische Überprüfung: die Modellierungsumgebungen verbinden die logisch zusammengehörenden Elemente

Schnittstellendeklaration:

```
var clock: integer = 60
```

Verwendung im Modell:

```
after 1 s [clock>0]/ clock-=1
```

- Syntaxgesteuerte Editoren

- Fehler während Bearbeitung → **Couldn't resolve reference**
- Moderne Umgebungen: Vorschlägen der Kandidaten

- Kode+Diagramm gemeinsam

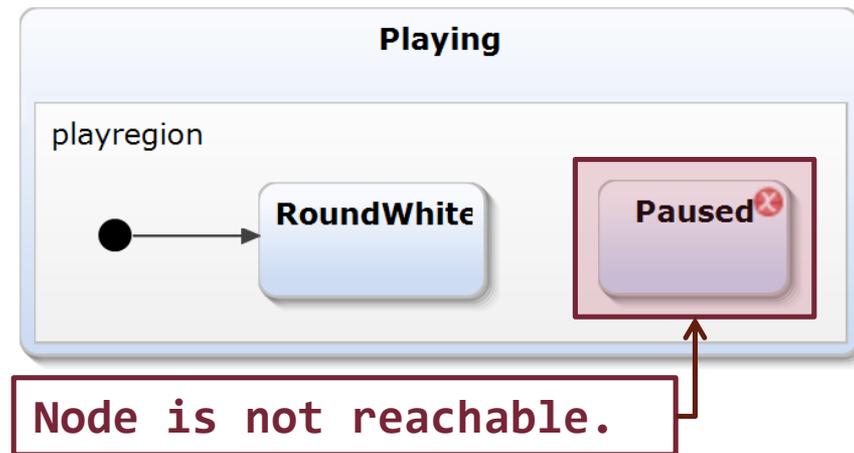
```
after 1 s [clock>0]/ clock-=1
```



- Programmieren: **fehlerhaft** während der Bearbeitung
- Modellieren: **korrekt** während der Bearbeitung

Statische Überprüfung: Strukturelle Überprüfung

- Strukturelle Überprüfung: Untersuchung des Modellgraphen
- Suchen nach Fehlermuster während der Bearbeitung
- z.B. unerreichbarer Zustand:



- Weitere Überprüfungen: fehlender Anfangszustand, Verklemmung, fehlerhafte Wertzuordnungen, etc.

- Unterstützung von Entwurfsrichtlinien:
Weitere Regeln können auch eingeführt werden
 - *Always* und *Oncycle*: für jedes Taktsignal feuervernde Ereignisse
 - Beliebige Frequenz → Typischerweise fehlerhaftes Verhalten

In der Hausaufgabe ist die Benutzung von *Always*- und *Oncycle*-Ereignissen strengstens **untersagt.**

Grundbegriffe

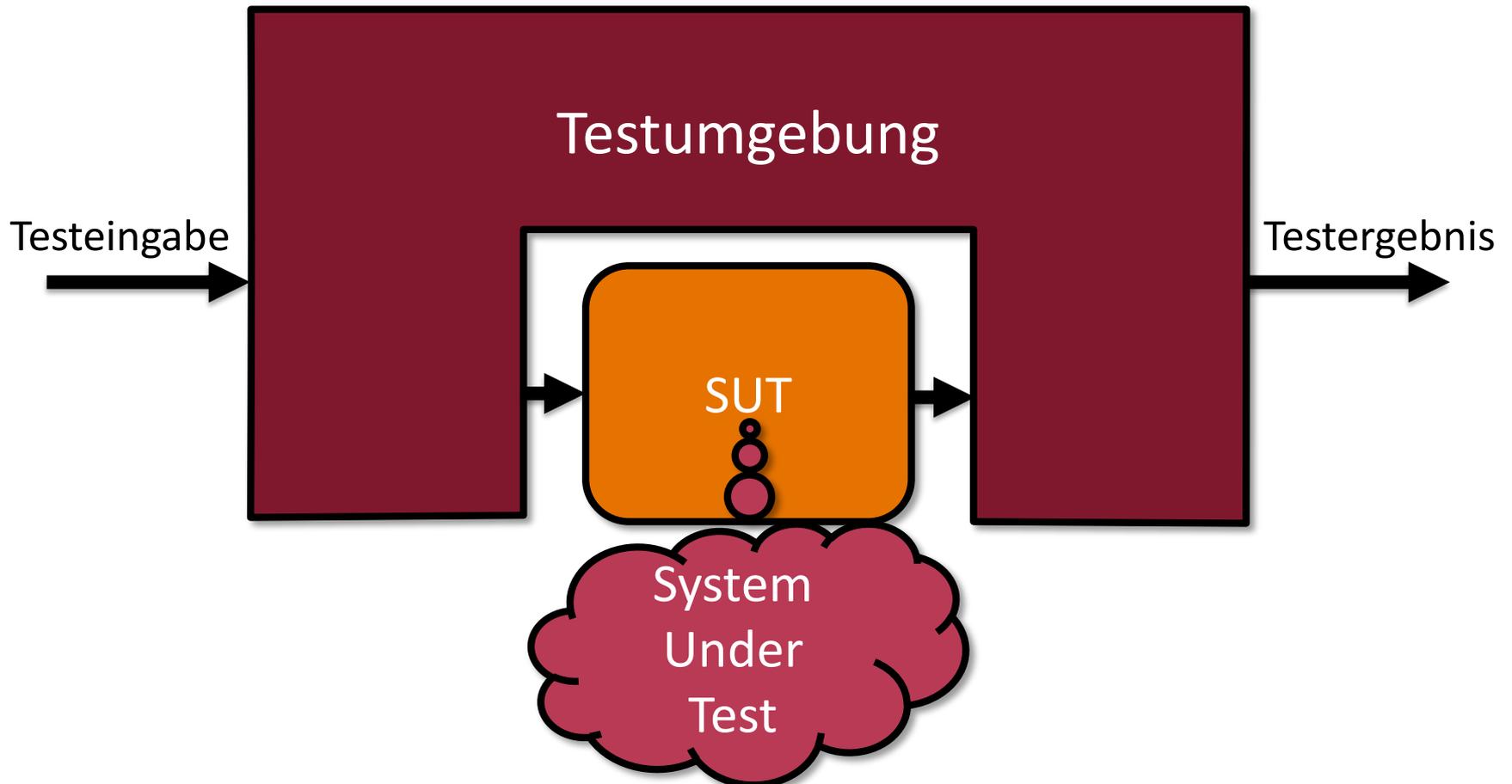
Stat. Überprüfung

Testen

Formale Verifikation

TESTEN

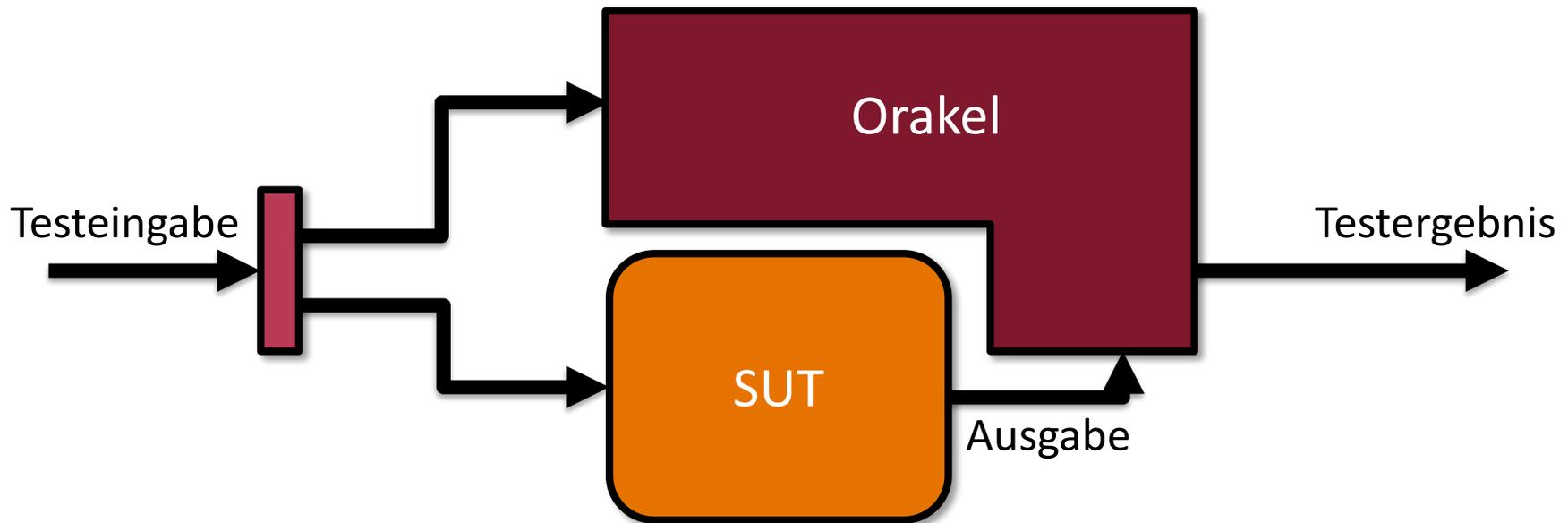
Testen von Modellen I.



Testen von Modellen II.

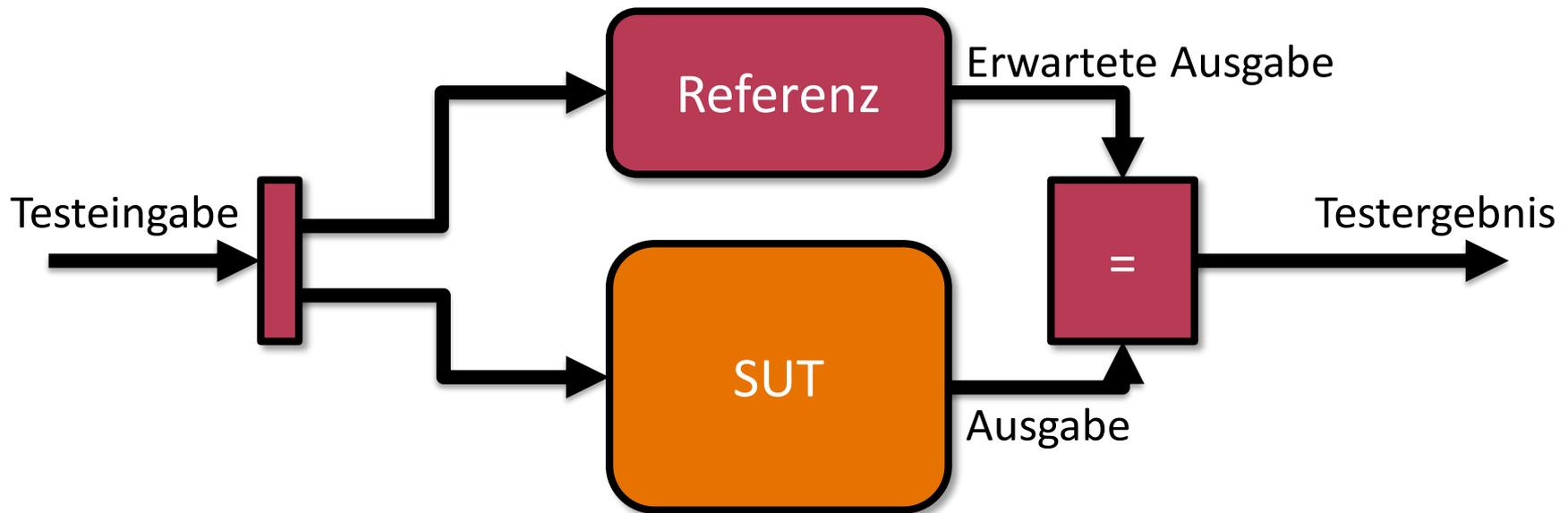
■ Das Orakel:

Erzeugung und Vergleichung der erwarteten Ausgaben

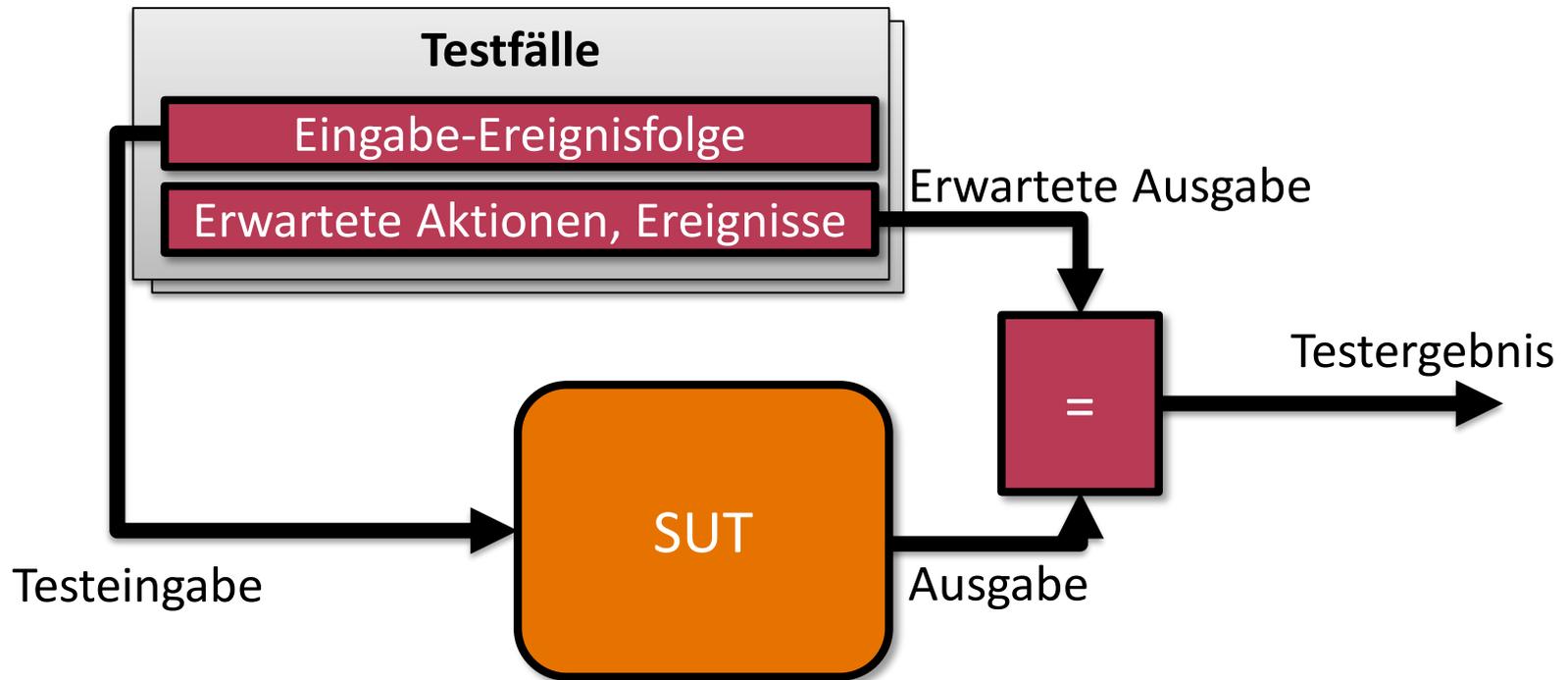


Testen von Modellen III.

- **Referenz:** Sie produziert die erwartete Ausgaben anhand der Testeingaben.



Testen von Modellen: Yakindu Zustandsmaschine



Testfall Beispiel: Im Einstellungsmenü kann die Anfangsbedenkzeit zwischen 1 und 3 Minuten in 5 Sekunden-schritten eingestellt werden.

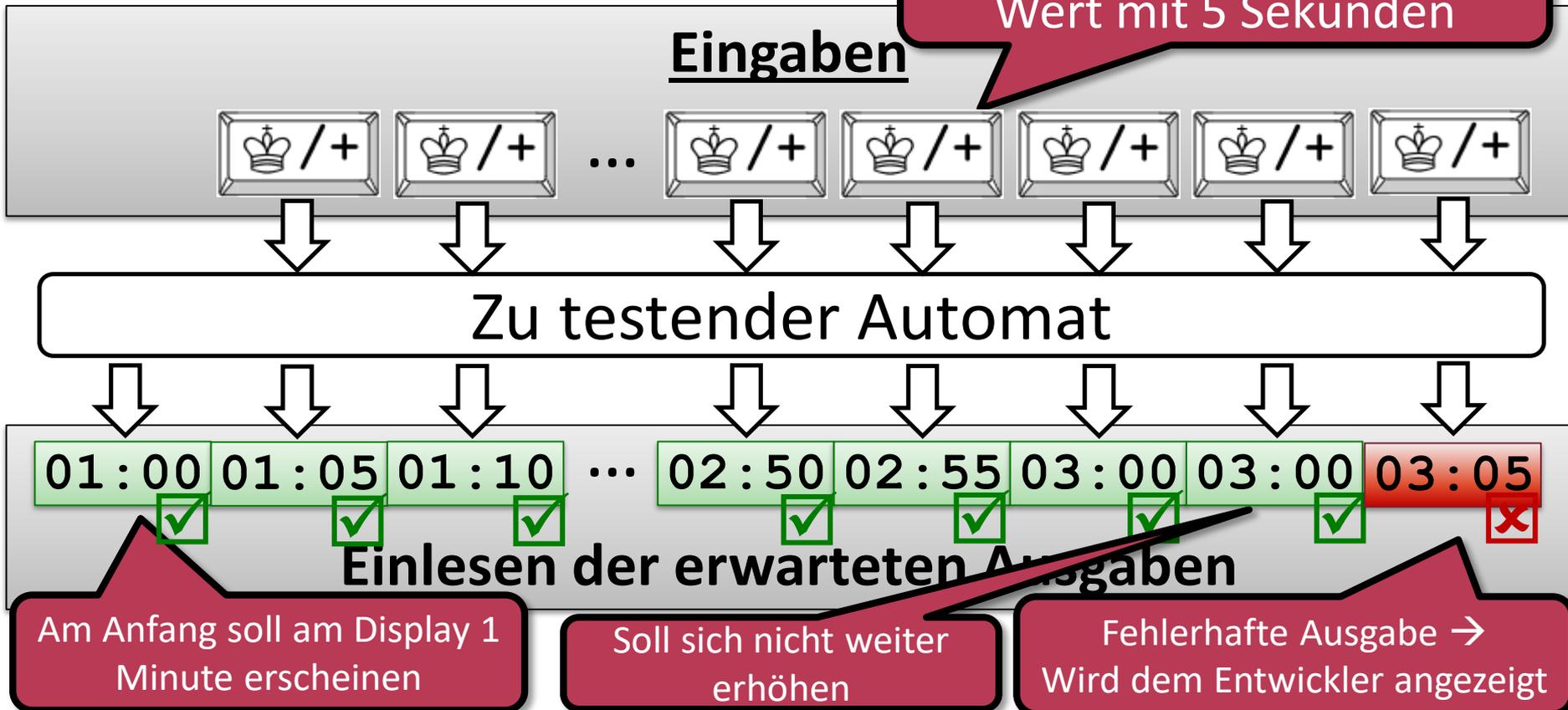
Eingaben

Zu testender Automat

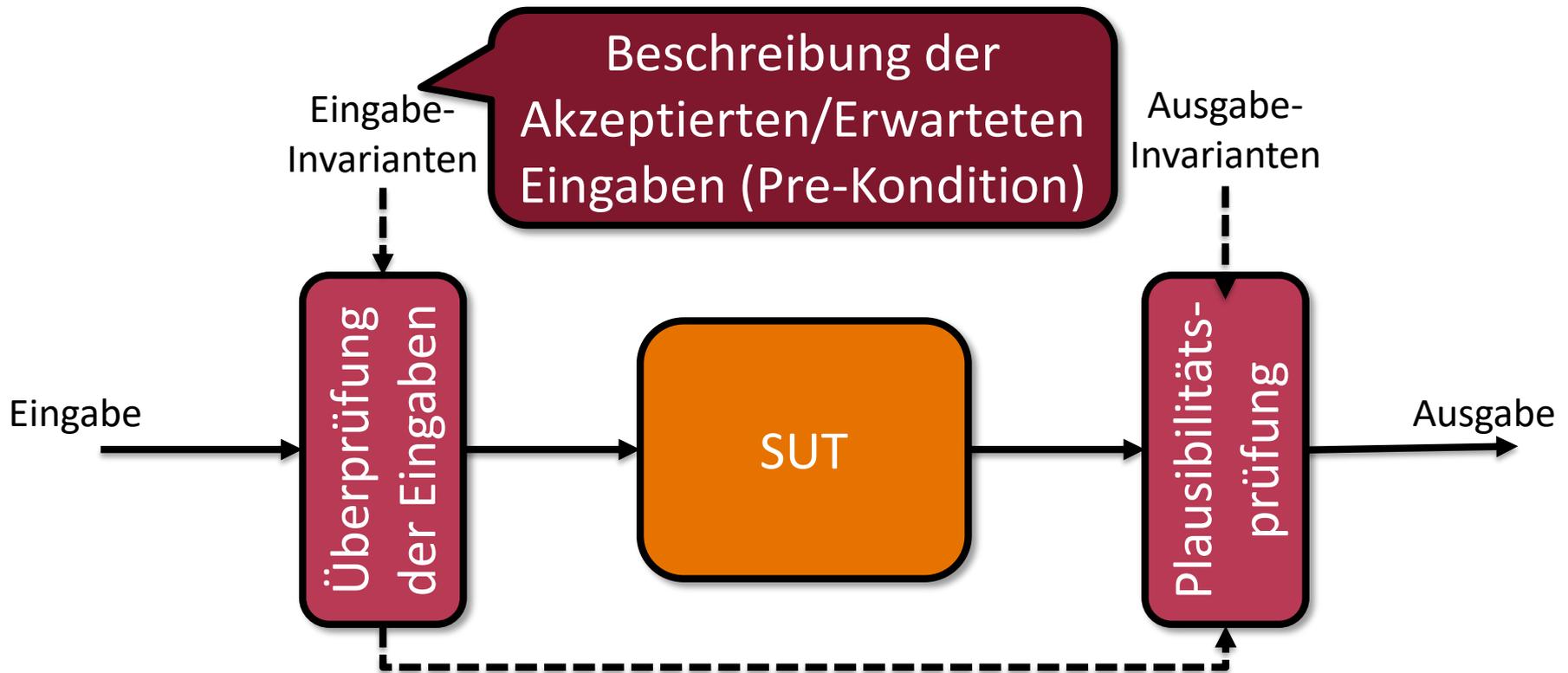
Einlesen der erwarteten Ausgaben

Testen von Modellen: Yakindu Zustandsmaschine

Testfall Beispiel: Im Einstellungsmenü kann die Anfangsbedenkzeit zwischen 1 und 3 Minuten in 5 Sekunden-schritten eingestellt werden.

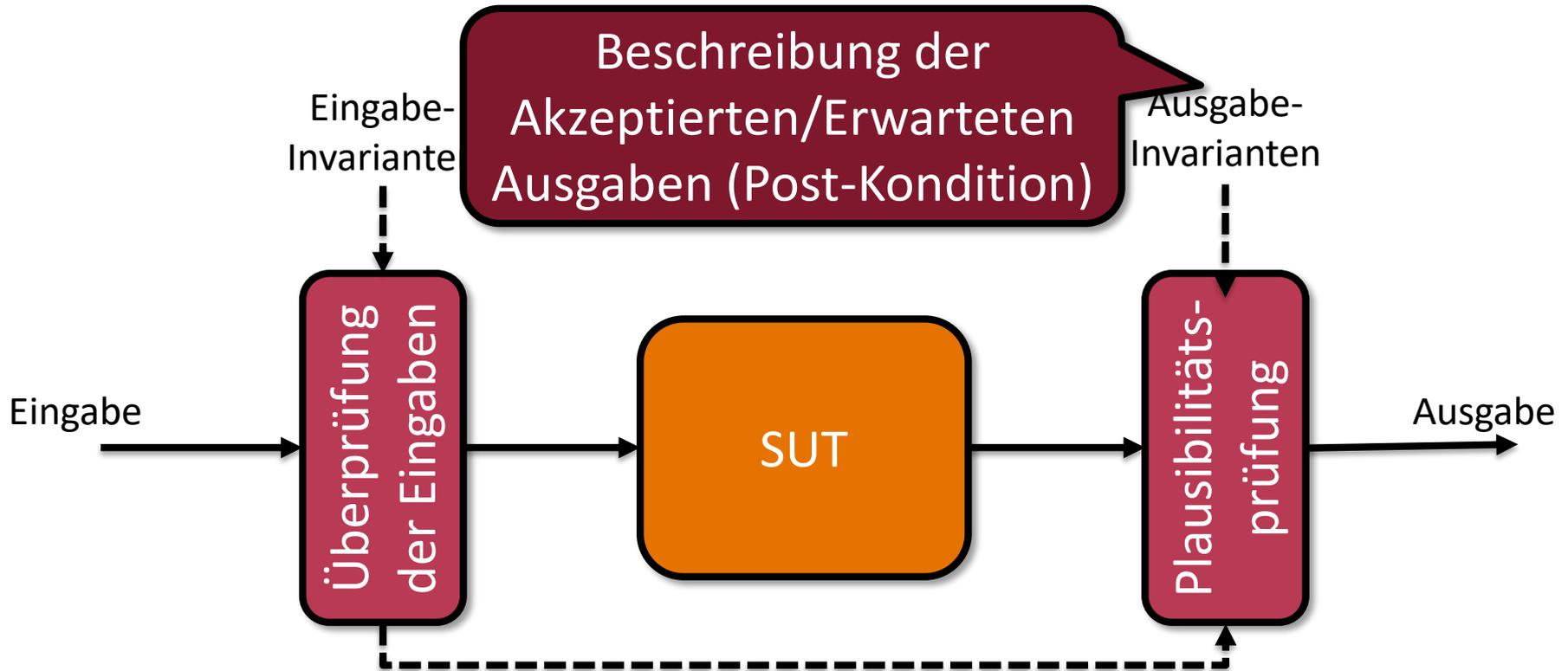


Selbsttest (monitor)



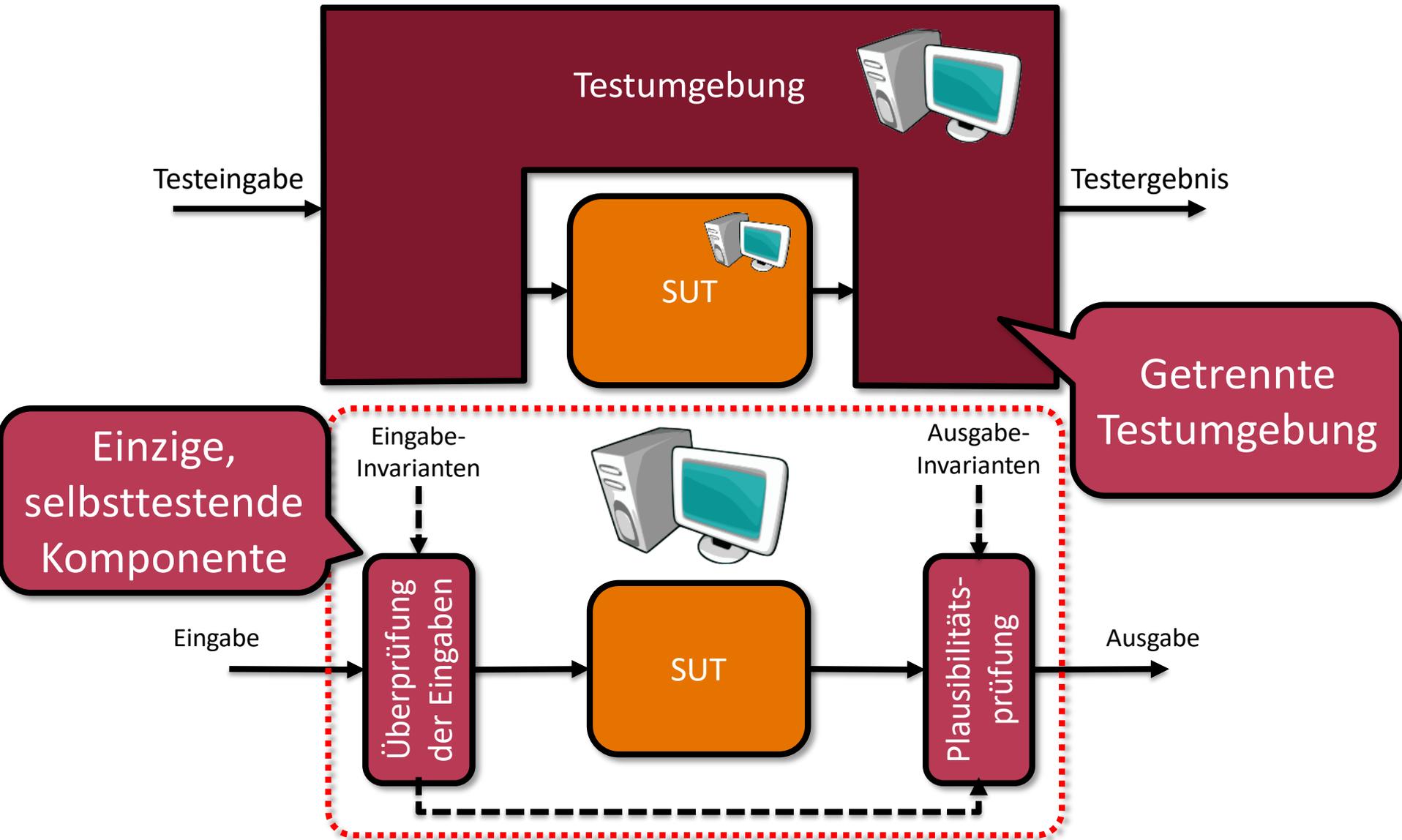
- **Invariante Eigenschaft:**
sie soll fortlaufend wahr sein

Selbsttest (monitor)



- **Invariante Eigenschaft:**
sie soll fortlaufend wahr sein

Selbsttest vs. Externes Test



Selbsttestendes Programm

Pre-Kondition: die Diskriminante soll nicht-negativ sein

```
void Roots(float a, b, c,  
           float &x1, &x2)  
{  
    float d = sqrt(b*b-4*a*c);  
  
    x1 = (-b + d)/(2*a);  
    x2 = (-b - d)/(2*a);  
}
```

Post-Kondition: beide Lösungen sollen Nullstellen sein

```
void RootsMonitor(float a, b, c,  
                  float &x1, &x2)  
{  
    // Pre-Kondition  
    float D = b2-4*a*c;  
    if (D < 0)  
        throw "Invalid input!";  
  
    // Ausführung  
    Roots(a, b, c, x1, x2);  
  
    // Post-Kondition  
    assert(a*x12+b*x1+c == 0 &&  
           a*x22+b*x2+c == 0);  
}
```

Selbsttestendes Programm

Exception (Ausnahme):

Von dem normalen abweichender, unerwarteter Fall, der **irgendwo anders behandelt wird**.

Ursache: falsche Bedienung.

```
float d = sqrt(b*b-4*a*c);
```

Assert (Behauptung):

Fehlerzustand, auf dessen Behandlung der Code **nicht vorbereitet wurde**.

Ursache: fehlerhafte Implementation oder Laufzeitfehler.

```
void RootsMonitor(float a, b, c,  
                 float &x1, &x2)
```

```
{
```

```
// Pre-Kondition
```

```
float D = b2-4*a*c;
```

```
if (D < 0)
```

```
    throw "Invalid input!";
```

```
// Ausführung
```

```
Roots(a, b, c, x1, x2);
```

```
// Post-Kondition
```

```
assert(a*x12+b*x1+c == 0 &&  
       a*x22+b*x2+c == 0);
```

```
}
```

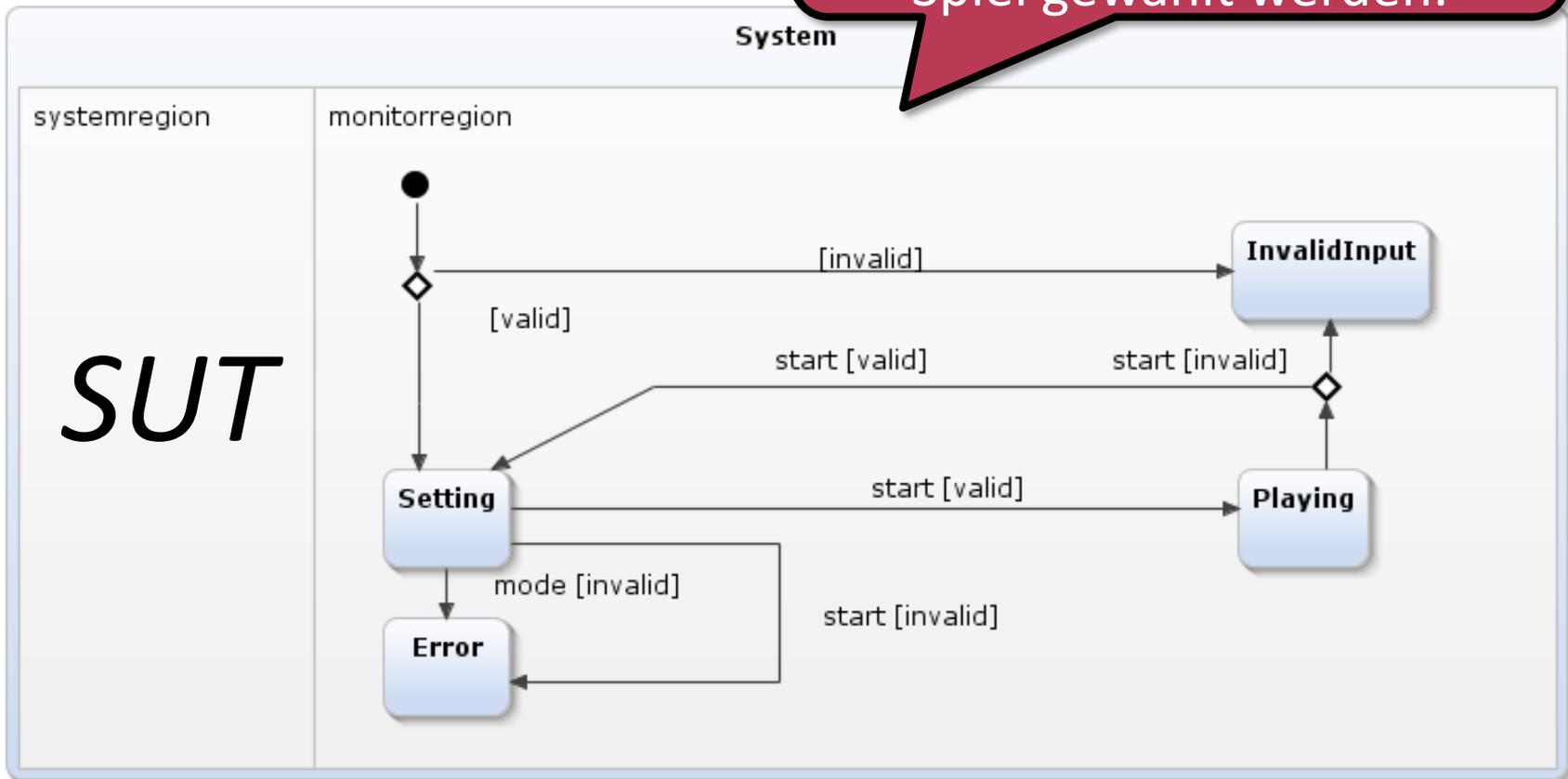
Selbsttest in Yakindu

- Die *SUT* und *monitor* Regionen laufen in parallel

- Korrekter Fall:

- Richtige Eingabe (valid)
- Korrekte Funktion

In der Hausaufgabe kann zwischen Einstellung und Spiel gewählt werden.



Selbsttest in Yakindu

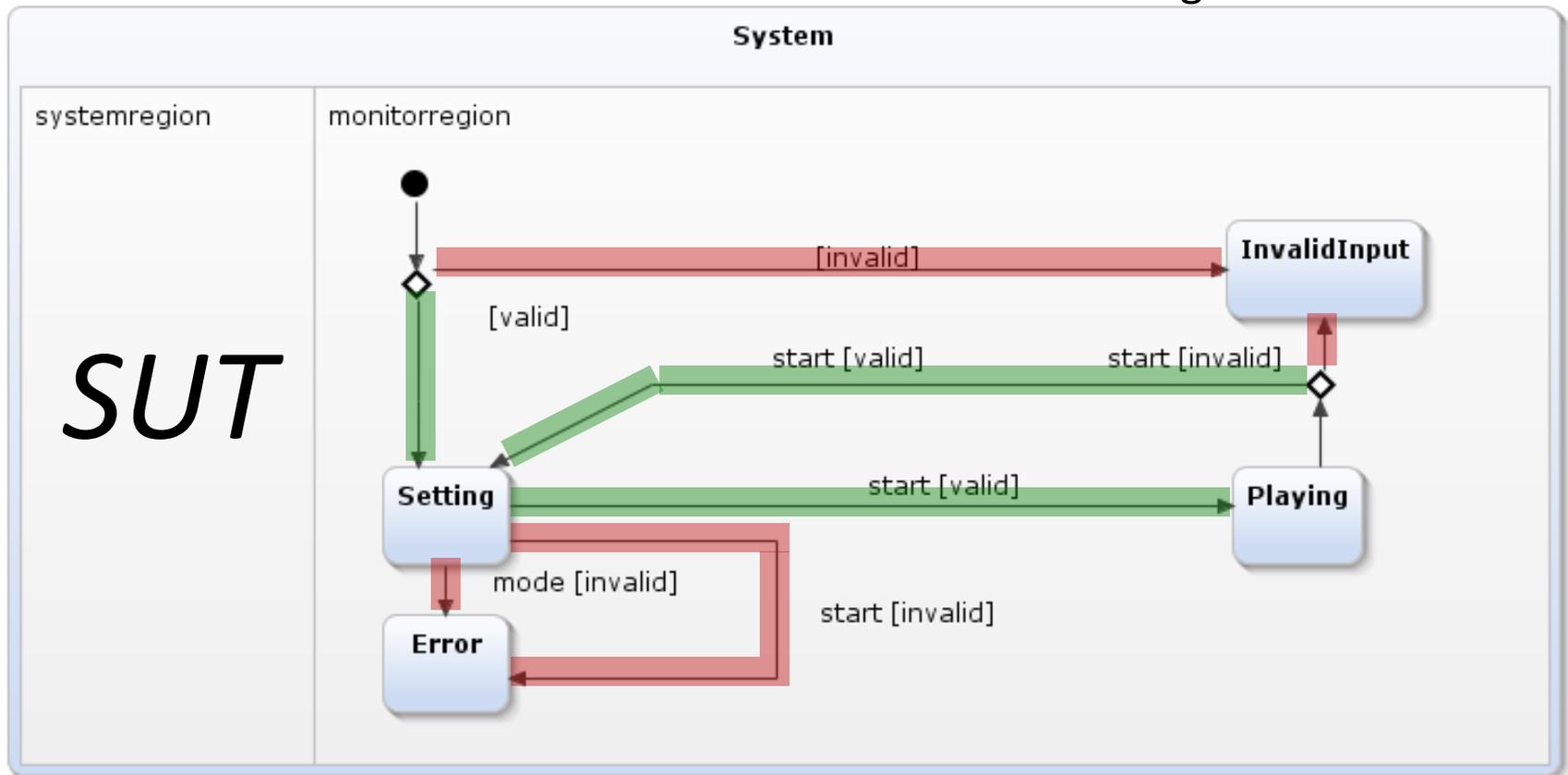
- Die *SUT* und *monitor* Regionen laufen in parallel

- Korrekter Fall:

- Richtige Eingabe (valid)
- Korrekte Funktion

- Fehlerhafter Fall:

- Fehlerhafte Eingabe → InvalidInput
- Fehlerhafte Ausgabe → Error



Selbsttest in Yakindu

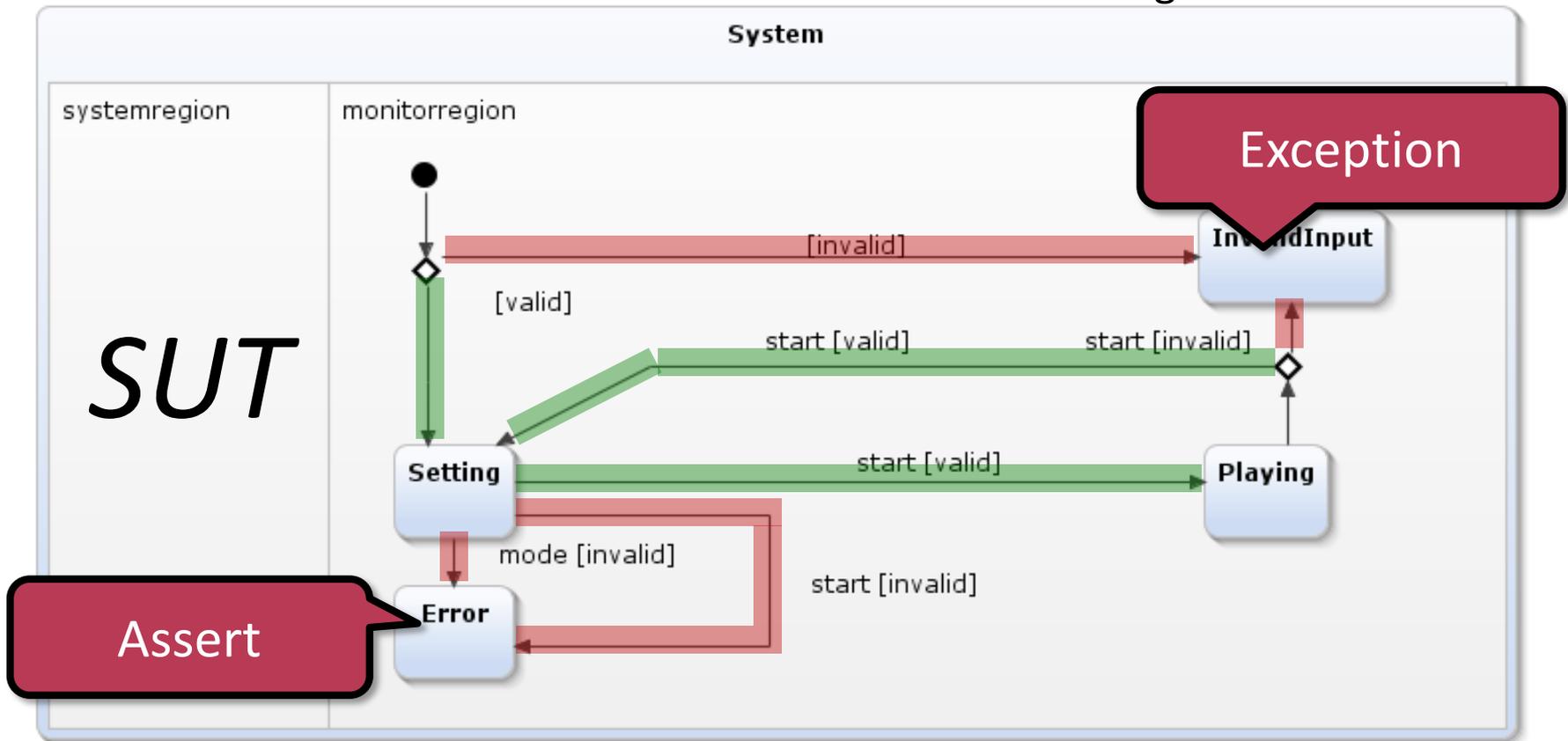
- Die *SUT* und *monitor* Regionen laufen in parallel

- Korrekter Fall:

- Richtige Eingabe (valid)
- Korrekte Funktion

- Fehlerhafter Fall:

- Fehlerhafte Eingabe → InvalidInput
- Fehlerhafte Ausgabe → Error



Testen von Modellen

- Ausführen des Modells: Simulation
 - Auf gegebene Eingaben überprüfetes Verhalten
- Testfälle:
 1. Testeingaben
 - z.B. die Mitte und die zwei Enden eines Wertebereiches
 2. Erwartete Ausgaben

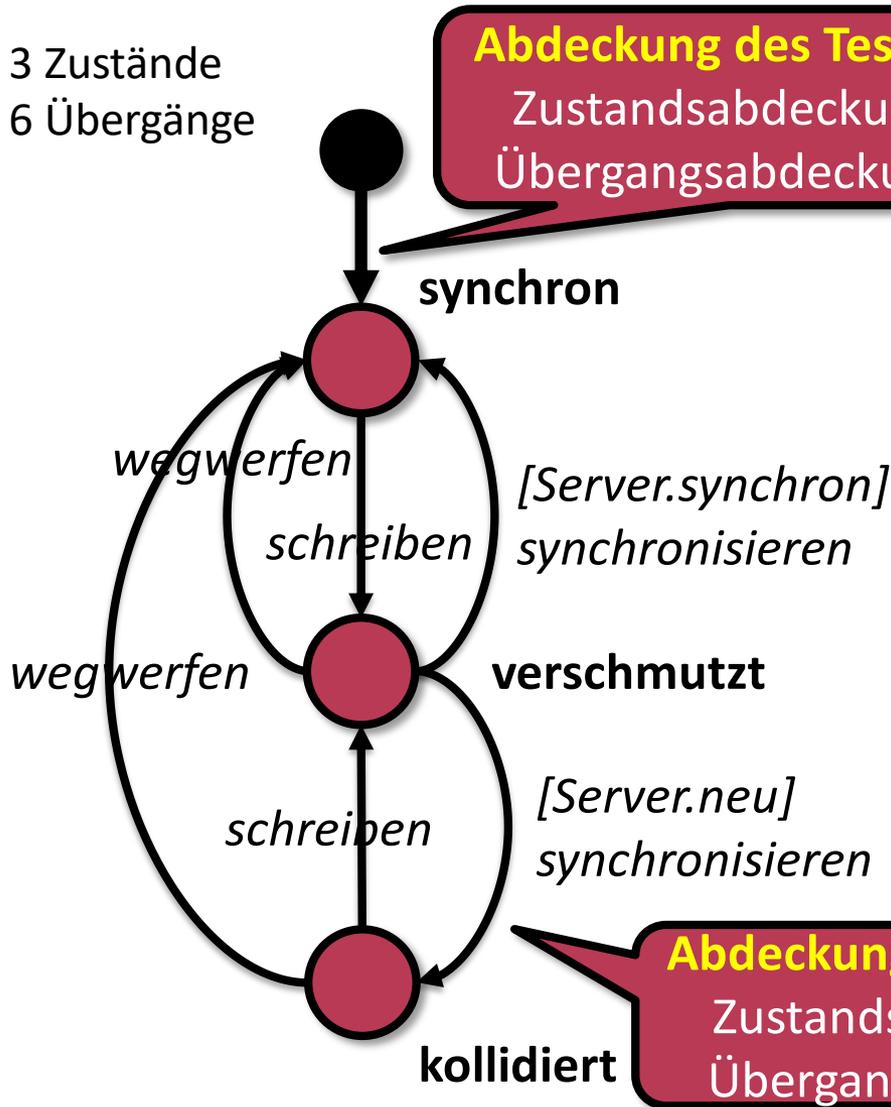
Mit welchen Eingaben soll getestet werden?

Testabdeckung

- Für einen gegebenen Testsatz: Verhältnis der während der Testausführung berührten Teile im Modell.
 - Zustandsabdeckung (in Zustandmaschinen):
$$\frac{\text{Anzahl der berührten Zustände}}{\text{Anzahl aller Zustände}}$$
 - Übergangsabdeckung (in Zustandmaschinen):
$$\frac{\text{Anzahl der berührten Übergänge}}{\text{Anzahl aller Übergänge}}$$
 - Aktivitätsabdeckung (in Kontrollflüssen):
$$\frac{\text{Anzahl der berührten Aktivitäten}}{\text{Anzahl aller Aktivitäten}}$$

Beispiel: Wolkenbasierte Datenspeicherung

3 Zustände
6 Übergänge



1. Testfall:

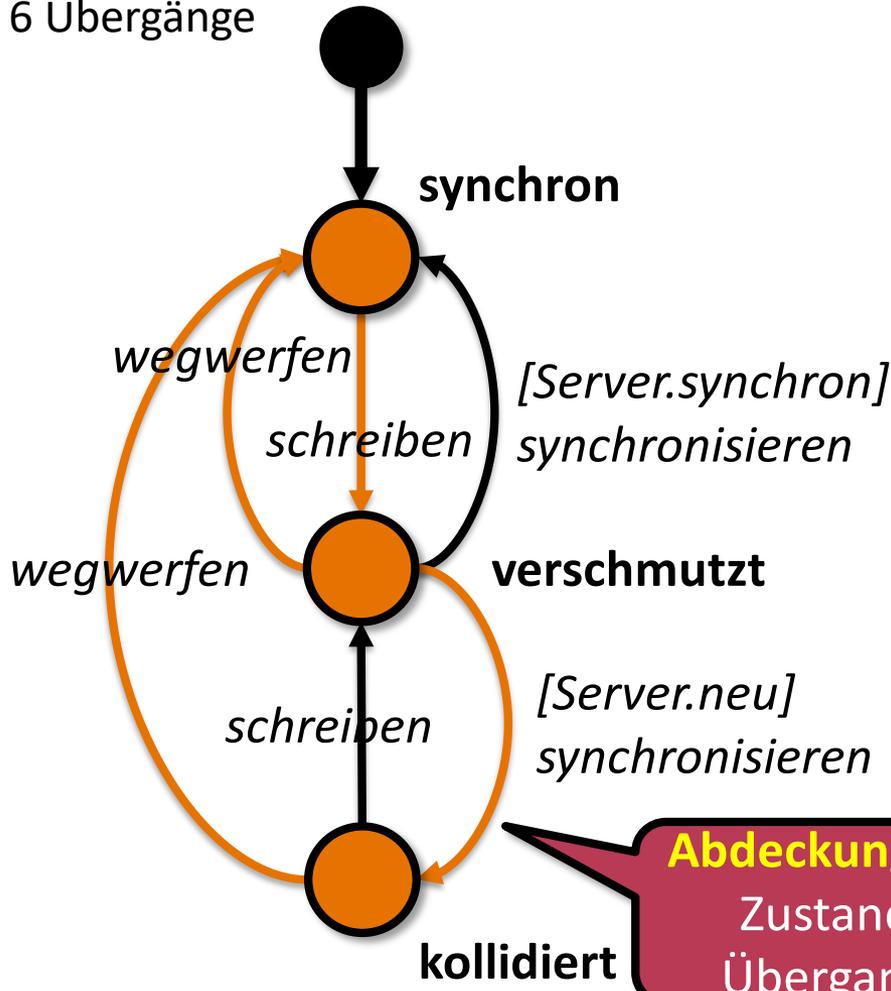
- a) schreiben
- b) wegwerfen

2. Testfall:

- a) schreiben
- b) synchronisieren
[Server = neu]
- c) wegwerfen

Beispiel: Wolkenbasierte Datenspeicherung

3 Zustände
6 Übergänge



3. Testfall:

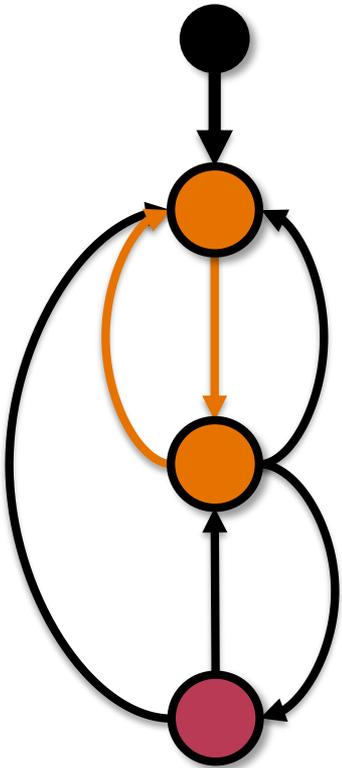
- schreiben
- synchronisieren
[Server = neu]
- schreiben
- synchronisieren
[Server = sync.]

Abdeckung

Nach dem 1. Testfall:

Zustandsabdeckung: $2/3=66\%$

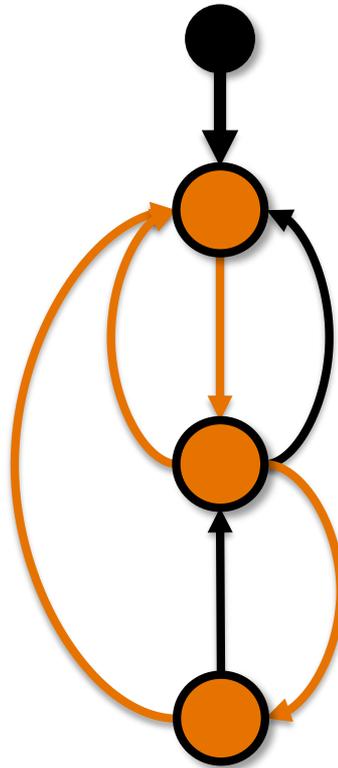
Übergangsabdeckung: $2/6=33\%$



Nach dem 2. Testfall:

Zustandsabdeckung: $3/3=100\%$

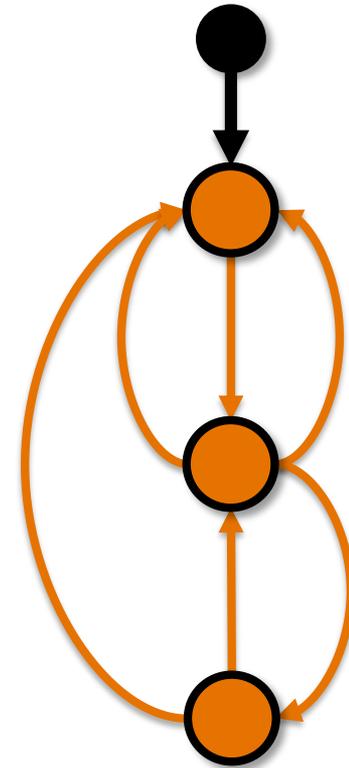
Übergangsabdeckung: $4/6=66\%$



Nach dem 3. Testfall:

Zustandsabdeckung: $3/3=100\%$

Übergangsabdeckung: $6/6=100\%$



Benutzen der getesteten Modelle

- **Softwaretesten:**
 - Wiederverwendung von Testsätzen (mit 100% Abdeckung)
 - Abdeckende Testeingaben (als Eingabe)
 - Vom Modell produzierte Ausgaben (als erwartete Ausgaben)
- **Monitoring:** Simulieren des Modells während Laufen der Software
 - Gleiche Eingaben für das Modell und das Programm
 - Vergleichung der Ausgaben → **Fehlererkennung**
- **Untersuchung der Aufzeichnungen/des Log:**
 - Monitoring mit den aufgezeichneten Ein- und Ausgaben

Benutzen der getesteten Modelle

■ Softwaretesten:

- Wiederverwendung von Testsätzen (als Eingabe)
- Abdeckende Testeingaben (als Eingabe)
- Vom Modell produzierte Ausgaben (als erwartete Ausgaben)

Vor der
Ausführung

■ **Monitoring:** Simulieren des Modells während Laufen der Software

- Gleiche Eingaben für das Modell und das Programm
- Vergleichung der Ausgaben → **Fehlererkennung**

Während der
Ausführung

■ **Untersuchung der Aufzeichnungen:**

- Monitoring mit den aufgezeichneten

Nach der
Ausführung

Testdokumentation

- Die Testfälle und –Ergebnisse sollen dokumentiert werden!

- Testspezifikation
- Was wird getestet?
 - Anhand welcher Anforderungen?
 - Mit welchen Eingaben?
 - Welche Ausgaben sind erwartet?

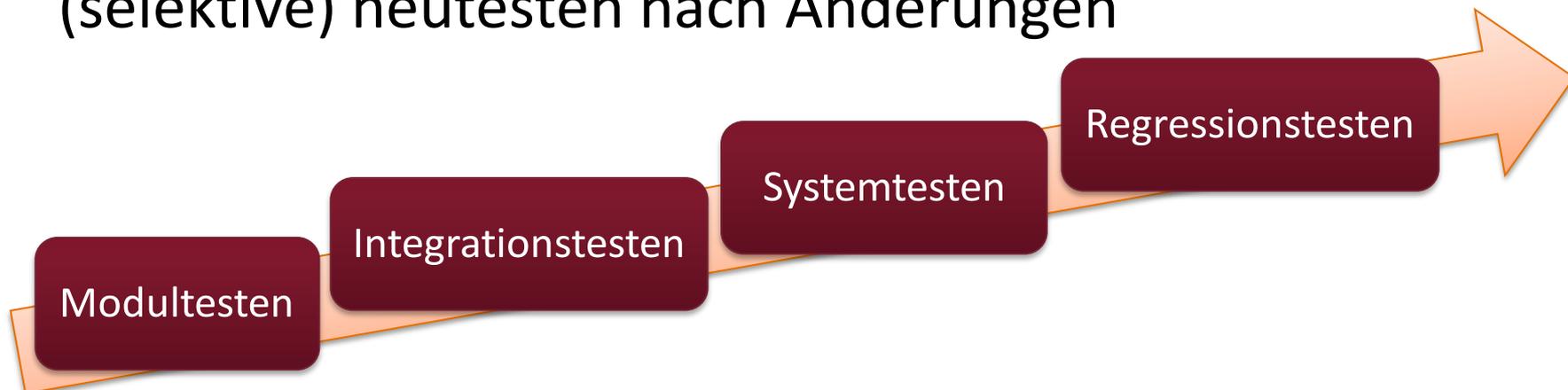
- Test-Report
- Wurde es ausgeführt?
 - Wenn ja, mit welchem Ergebnis?

- Nachweisbarkeit :
 - Aufdeckung von ungetesteten Kodeteilen und unüberprüften Anforderungen
 - Zurückverfolgbarkeit der Testergebnisse



Testarten, Testphasen

- **Modultesten:**
eine Komponente wird abgetrennt und so getestet
- **Integrationstesten:**
mehrere Komponenten werden gemeinsam getestet
- **Systemtesten:**
das ganze System zusammen wird getestet
- **Regressionstesten:**
(selektive) neutesten nach Änderungen



Grundbegriffe

Stat. Überprüfung

Testen

Formale Verifikation

FORMALE VERIFIKATION

Formale Verifikation

- **Formale Verifikation:** Beweis der Korrektheit von Modellen/Programmen mit mathematischen Mitteln
 - Für mehr dazu siehe: Formale Methoden MSc-LVA
- **Möglichkeiten:**
 - **Modellprüfung** (model checking)
 - Erschöpfende Untersuchung aller möglichen Verhalten
 - **Automatischer Beweis der Korrektheit**
 - Automatisches Beweisverfahren anhand Axiomensysteme
 - **Konformanzuntersuchungen**
 - Überprüfung der Übereinstimmung von Modellen

Modellprüfung

- **Modellprüfung:** Erschöpfende (vollständige) Untersuchung aller potentiellen Verhalten eines Modells anhand gegebener Anforderungen
 - Suchen nach Fehlverhalten
- **Gegenbeispiel**

Testen	Modellprüfung
Stichprobenartig	Erschöpfend/vollständig
Überprüft erwartete Ausgaben	Überprüft Zustandsfolgen
Kleiner Rechenaufwand	Grosser Rechenaufwand
Nicht beweisstark	Formaler Beweis