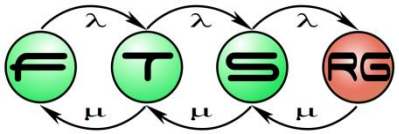
Überprüfung der Modelle

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

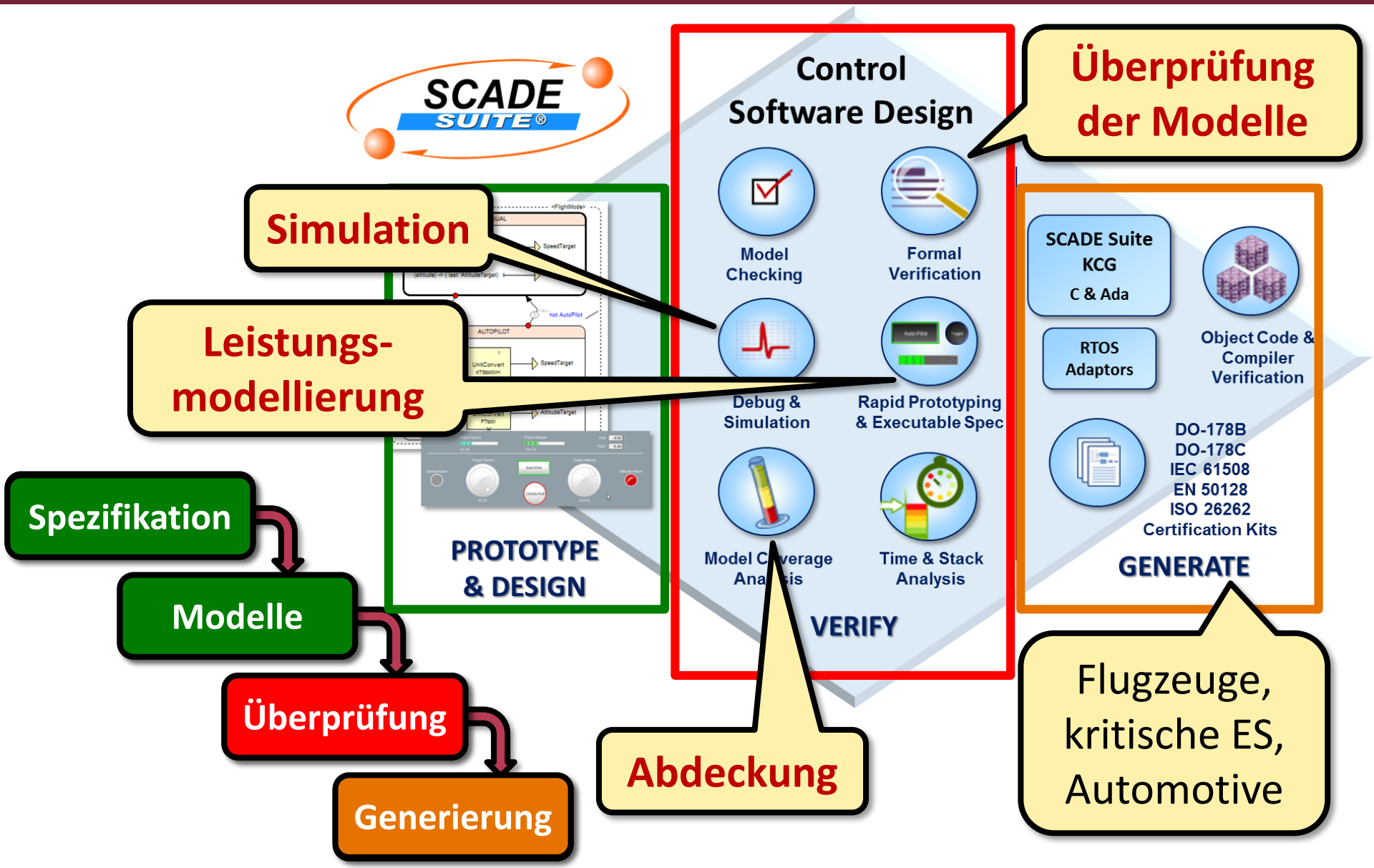
Ariane 5 Trägerrakete

- 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



INHALT

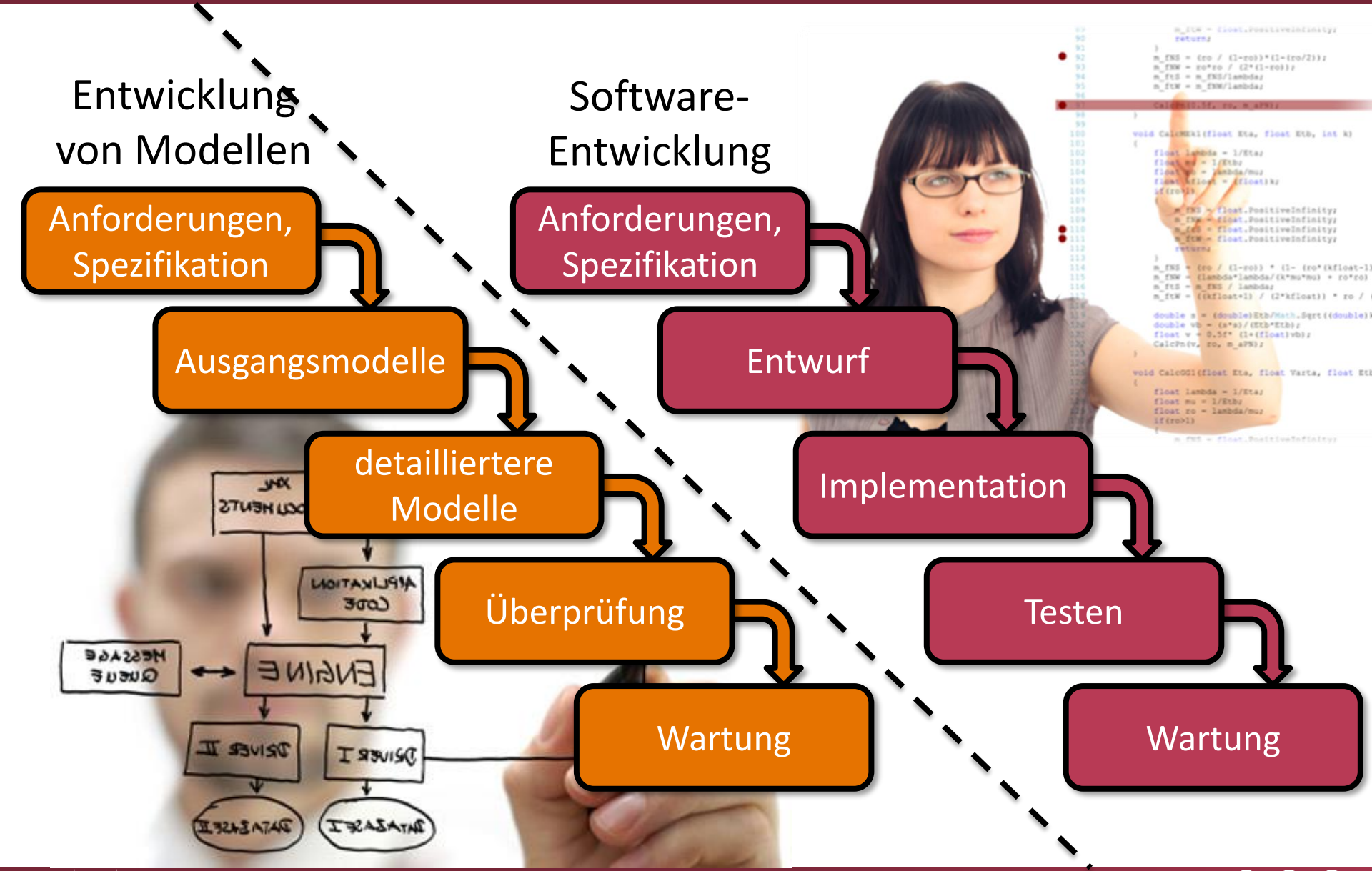
Lebenszyklus der Modelle

Entwicklung von Modellen

- Anforderungen, Spezifikation
- Ausgangsmodelle
- detailliertere Modelle
- Überprüfung
- Wartung

Software-Entwicklung

- Anforderungen, Spezifikation
- Entwurf
- Implementation
- Testen
- Wartung



Automatische Kodegenerierung

Entwicklung von Modellen

Software-Entwicklung

Anforderungen, Spezifikation

Ausgangsmodelle

Anforderungen, Spezifikation

Implementation

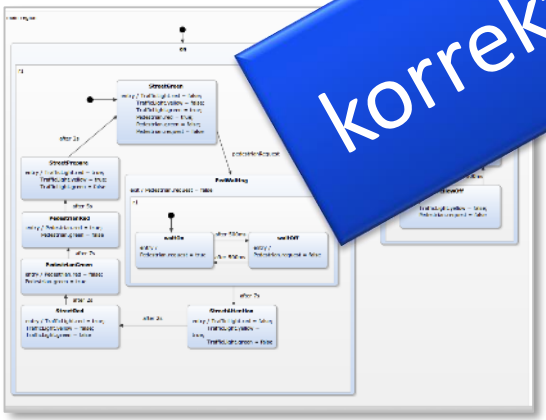
Überprüfung

Testen

Wartung

Wartung

korrektes Modell, korrekterer Kode



GRUNDBEGRIFFE

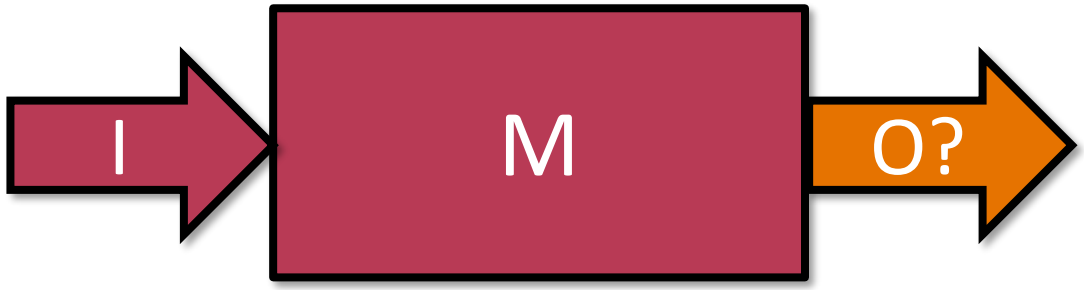
Modelle und Aufgaben

- Synthese:

  Welches Modell entspricht der Spezifikation?

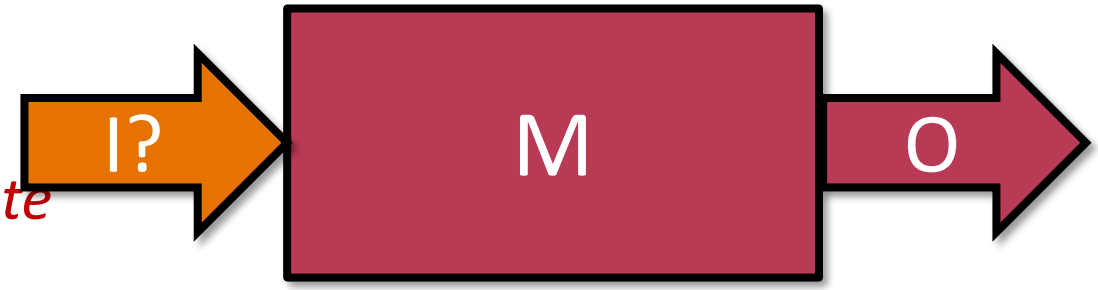
- Analyse:

  Wie ist das Verhalten des Modelles?



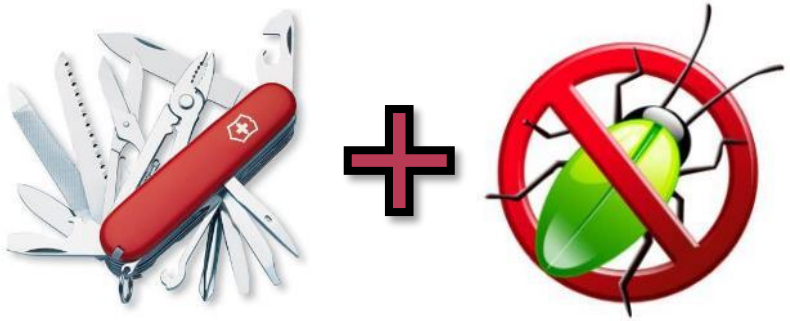
- Steuerung:

  Wie ist der gewünschte Zustand erreichbar?



Korrektheit

- Die Korrektheit ist die Fähigkeit des Modells oder Kodes, 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
  - Universelle Anforderungen
    - sollen immer wahr sein
- Erwartetes Verhalten (z.B. l
  - Welche Zustände sollen erre
  - Welche Funktionen soll
  - Existenzielle Anforderungen
    - sollen immer erreichbar sein

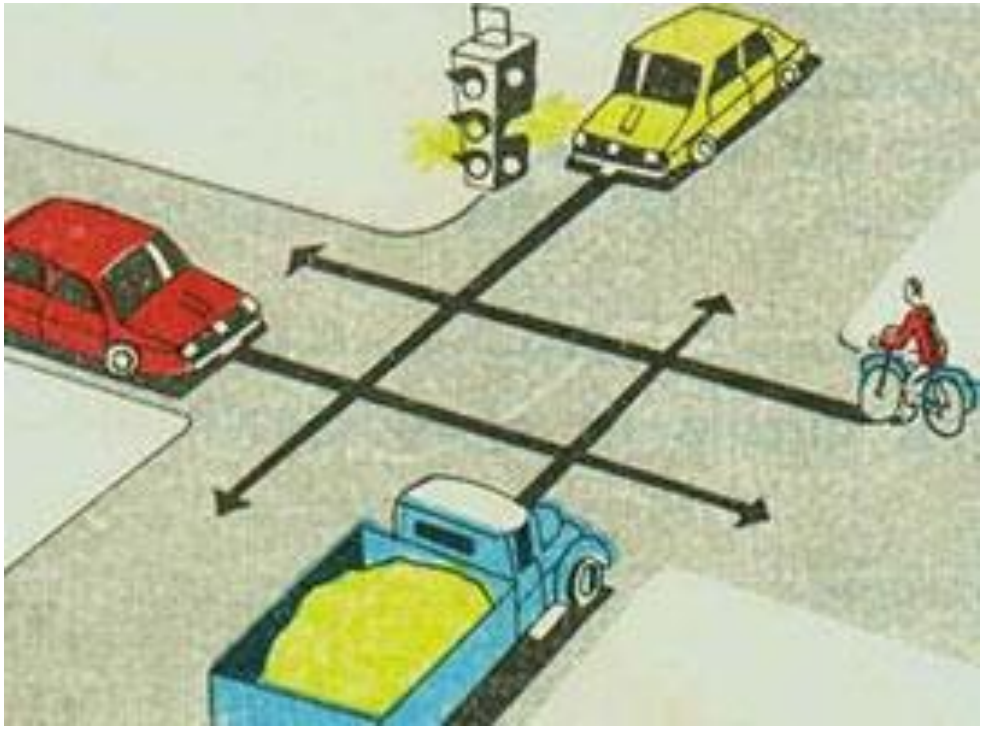
„Die Verkehrsampeln einer Kreuzung **dürfen nie** gleichzeitig grün sein.”

„Die Verkerhsampel **soll immer fähig sein** auf grün zu wechseln.”

Verklemmung (deadlock)

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

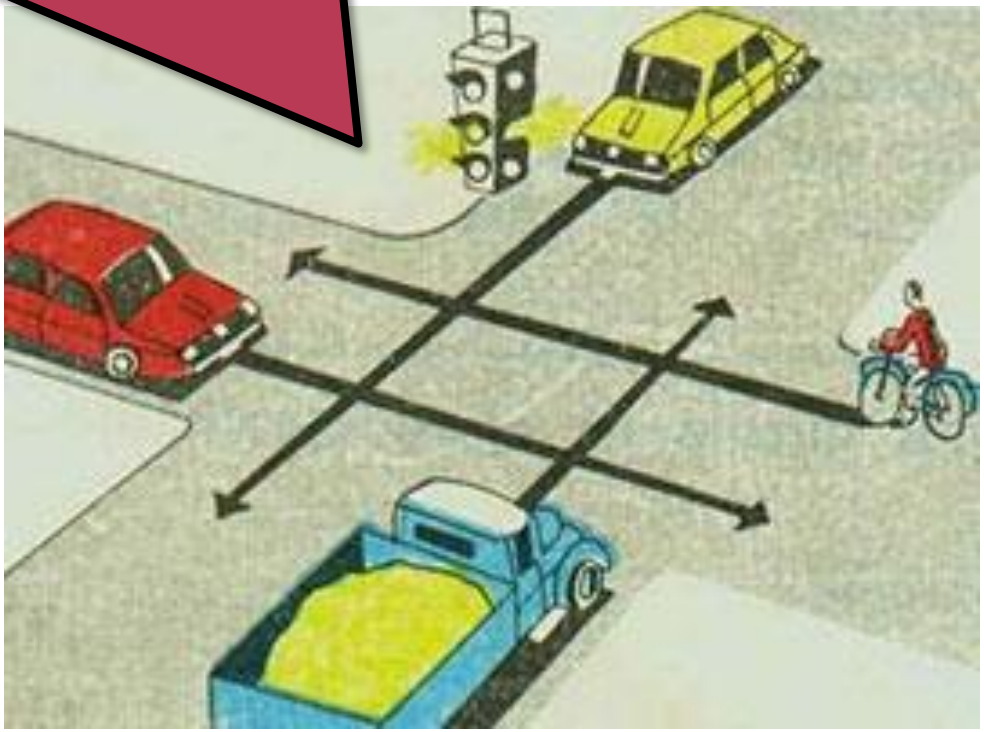
- z.B. aufeinander wartende Prozesse



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)

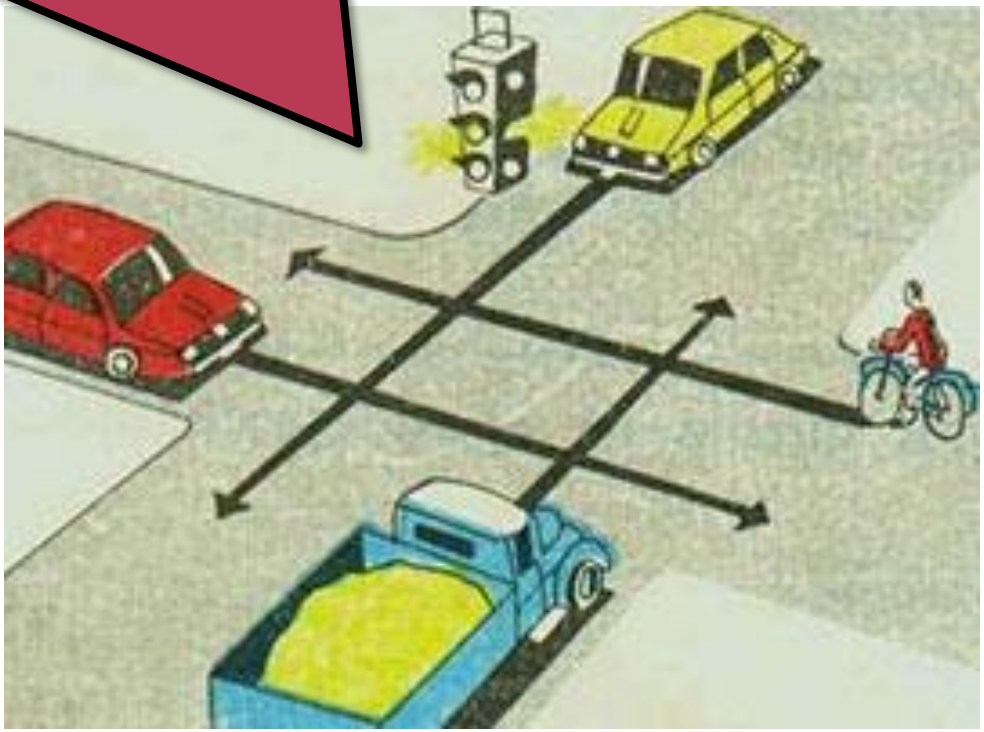
?



?

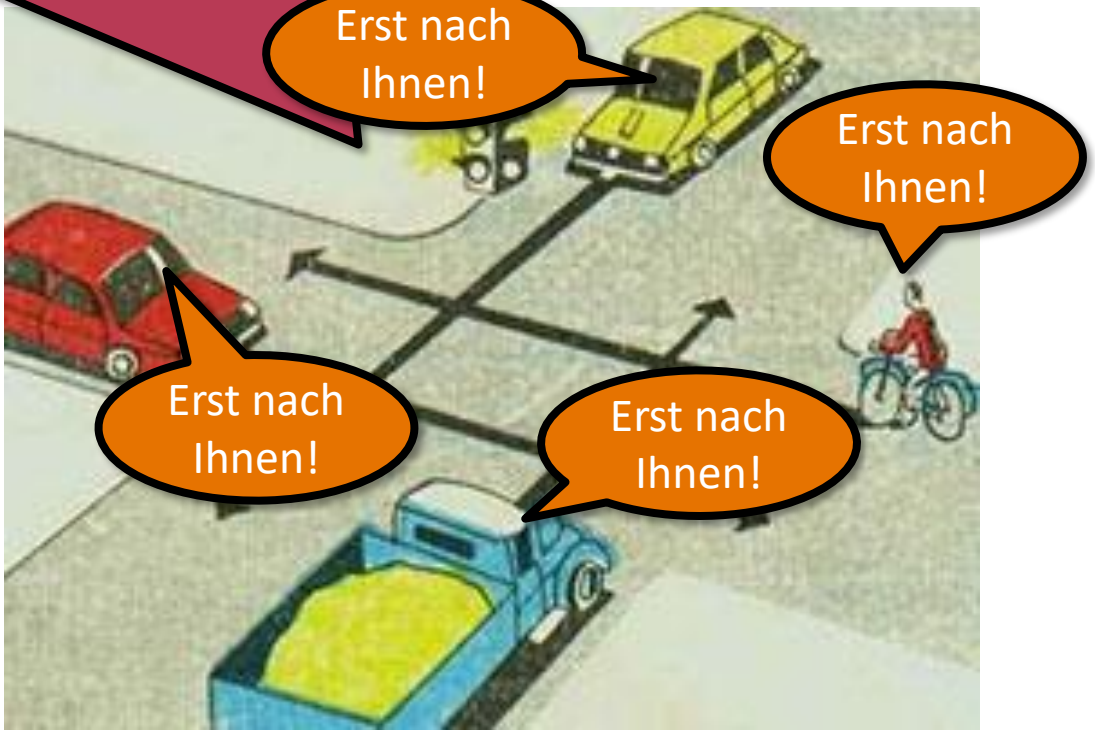
Auflösung einer Verklemmung

„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)



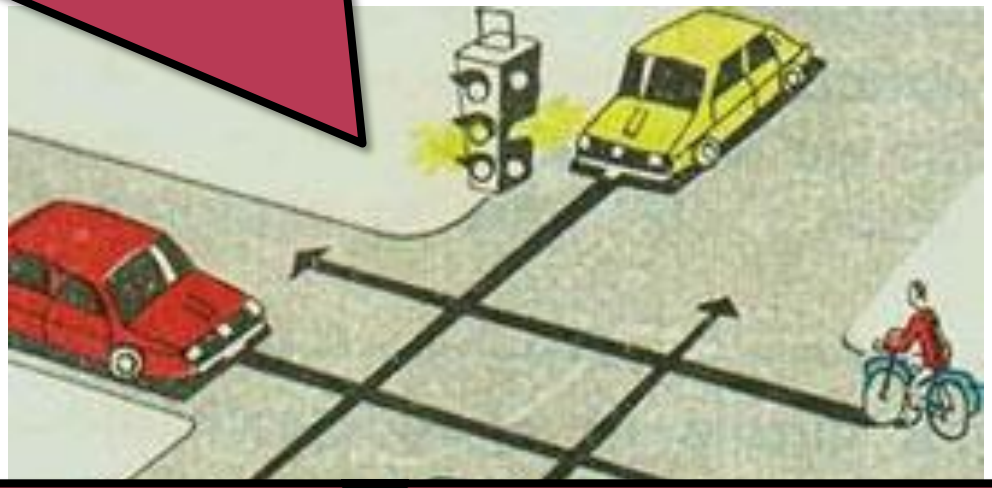
Auflösung einer Verklemmung

„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 irgend**einer** der Fahrer auf seinen Vorrang verzichten. Ansonsten werden sie laut StVO ewig stehenbleiben.” (gyakorikerdesek.hu)



Auflösung der Verklemmung der Auflösung:

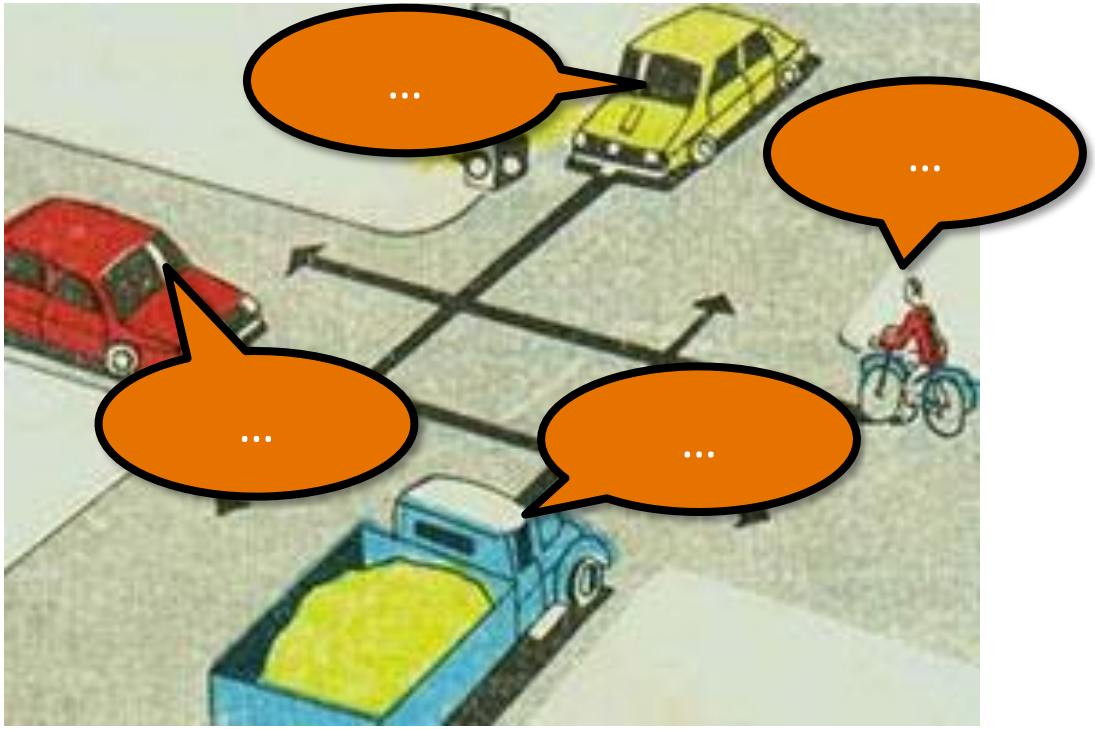
- Asymmetrische Algorithmen
- Algorithmen mit Zufallsprinzip
  - Siehe den Backoff-Verfahren bei Ethernet-Netzwerken

Endlosschleife (livelock)

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

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

?



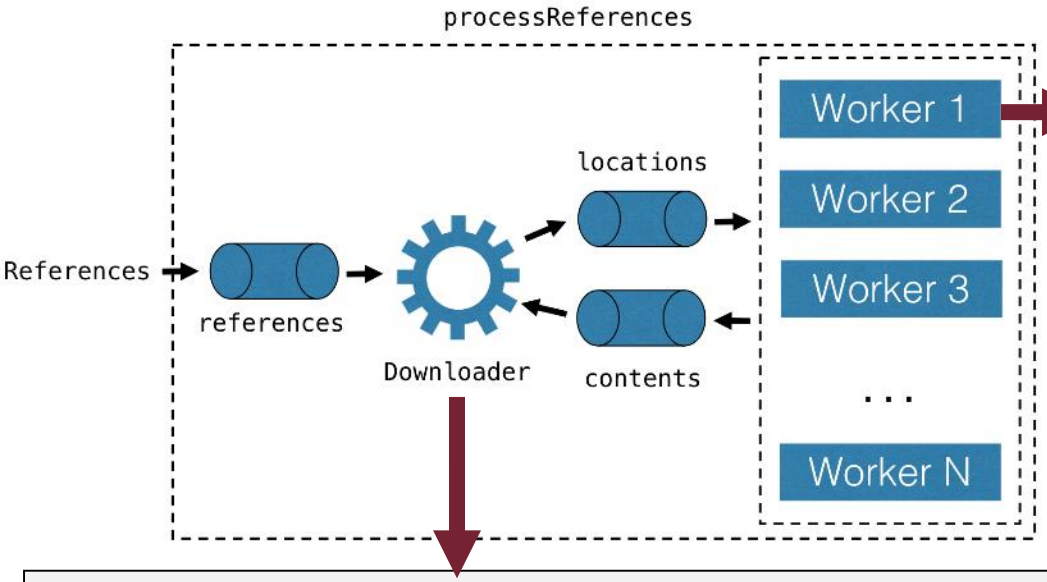
?

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 warten auf die andere R.

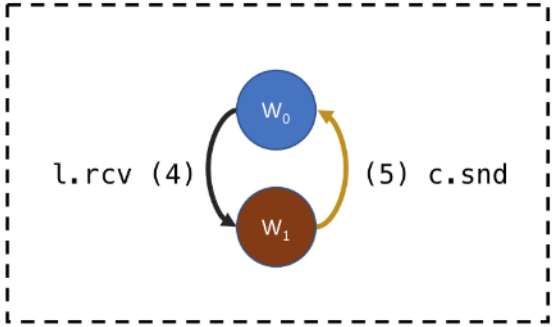
Bsp.: Kommunizierende SW-Komponente



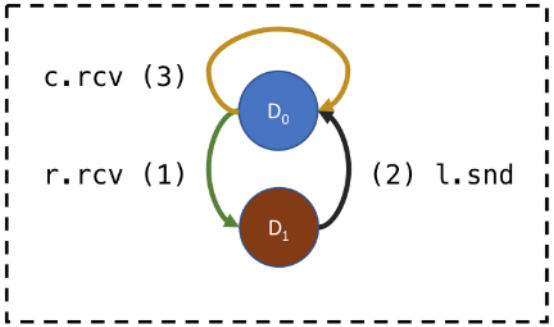

```
for (loc in locations) {          // (4)
  val content = downloadContent(loc)
  contents.send(                  // (5)
    LocContent(loc, content)
  )
}
```

```
while (true) {
  select<Unit> {
    references.onReceive {                    // (1)
      ref ->
        val loc = ref.resolveLocation()
        //...
        locations.send(loc)                   // (2)
      }
    contents.onReceive {                      // (3)
      (loc, content) -> //...
    }
  }
}
```

Worker

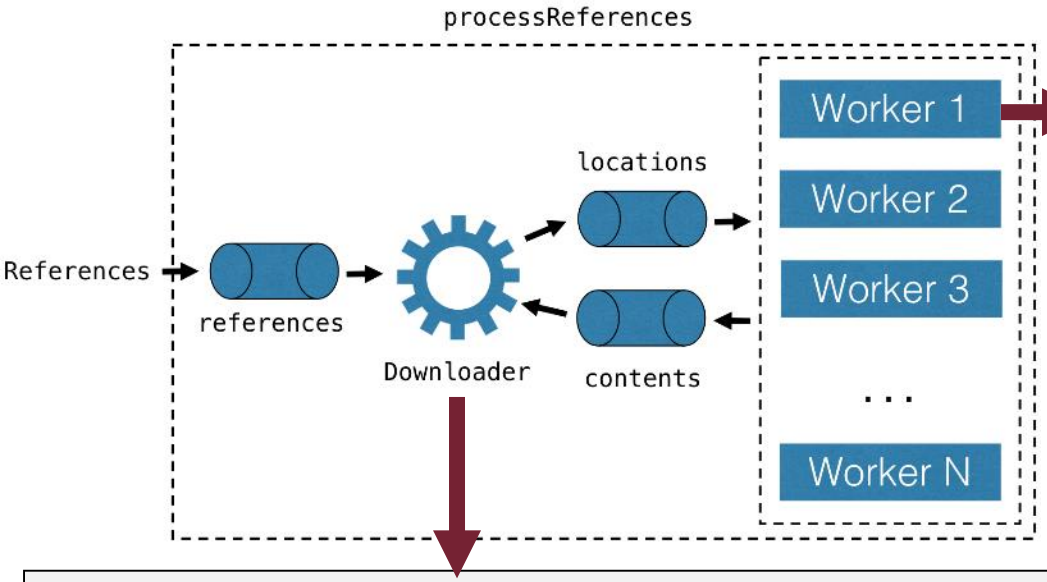


Downloader



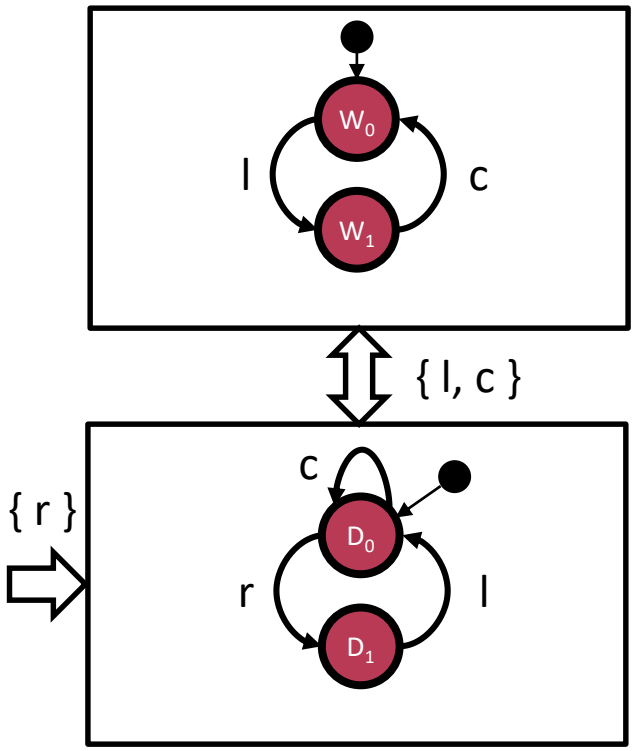
Forrás: https://medium.com/@elizarov/deadlocks-in-non-hierarchical-csp-e5910d137cc

Bsp.: Kommunizierende SW-Komponente



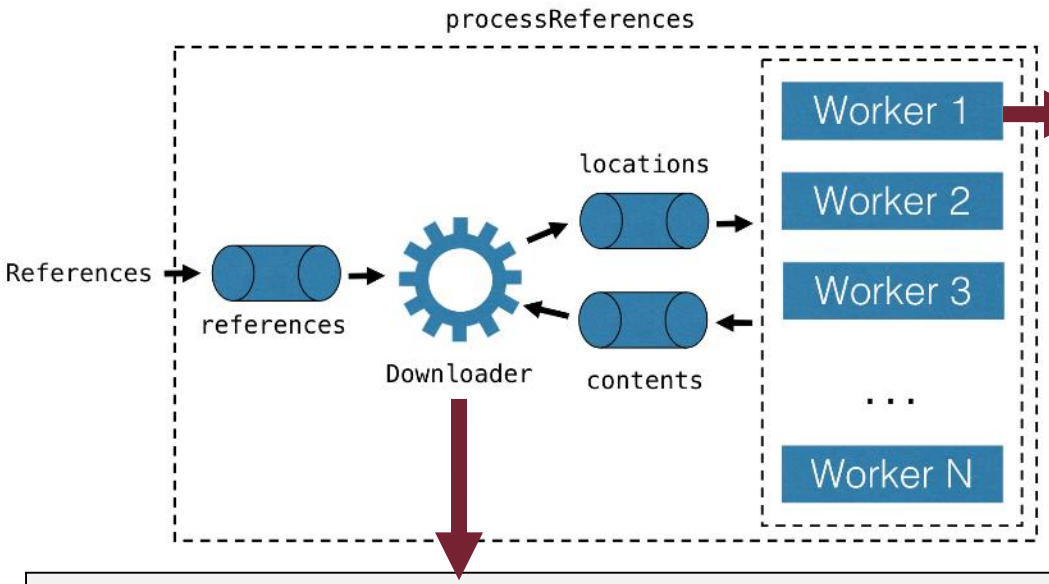

```
for (loc in locations) {          // (4)
  val content = downloadContent(loc)
  contents.send(                  // (5)
    LocContent(loc, content)
  )
}
```

```
while (true) {
  select<Unit> {
    references.onReceive {                    // (1)
      ref ->
        val loc = ref.resolveLocation()
        //...
        locations.send(loc)                   // (2)
      }
    contents.onReceive {                      // (3)
      (loc, content) -> //...
    }
  }
}
```



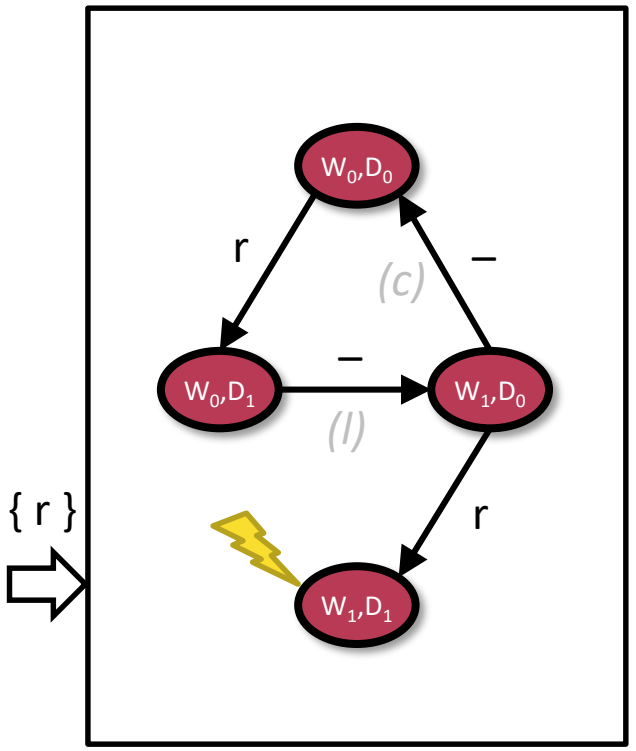
Forrás: https://medium.com/@elizarov/deadlocks-in-non-hierarchical-csp-e5910d137cc

Bsp.: Kommunizierende SW-Komponente



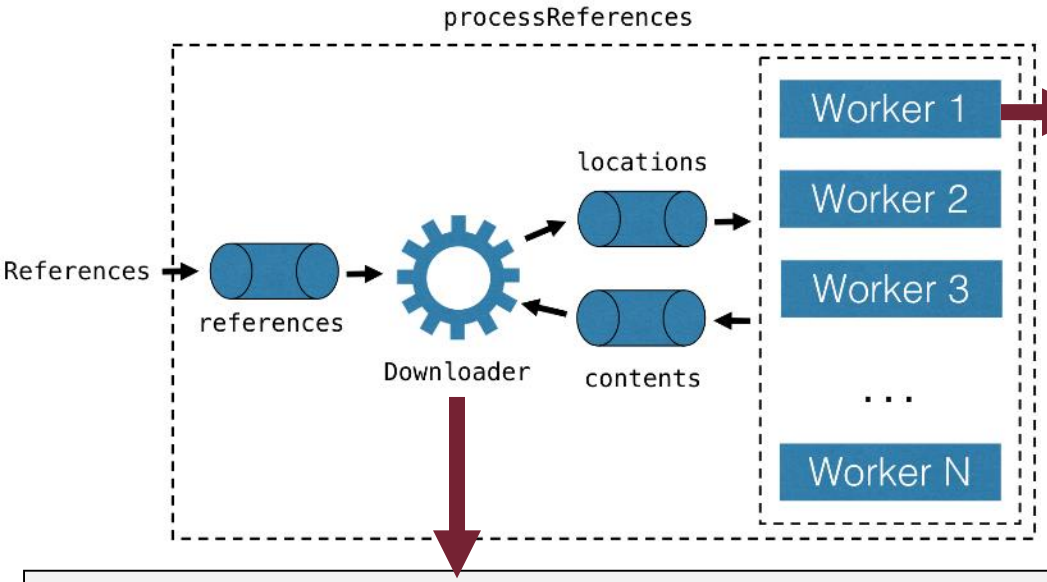

```
for (loc in locations) {          // (4)
  val content = downloadContent(loc)
  contents.send(                  // (5)
    LocContent(loc, content)
  )
}
```

```
while (true) {
  select<Unit> {
    references.onReceive {                    // (1)
      ref ->
        val loc = ref.resolveLocation()
        //...
        locations.send(loc)                   // (2)
      }
    contents.onReceive {                      // (3)
      (loc, content) -> //...
    }
  }
}
```



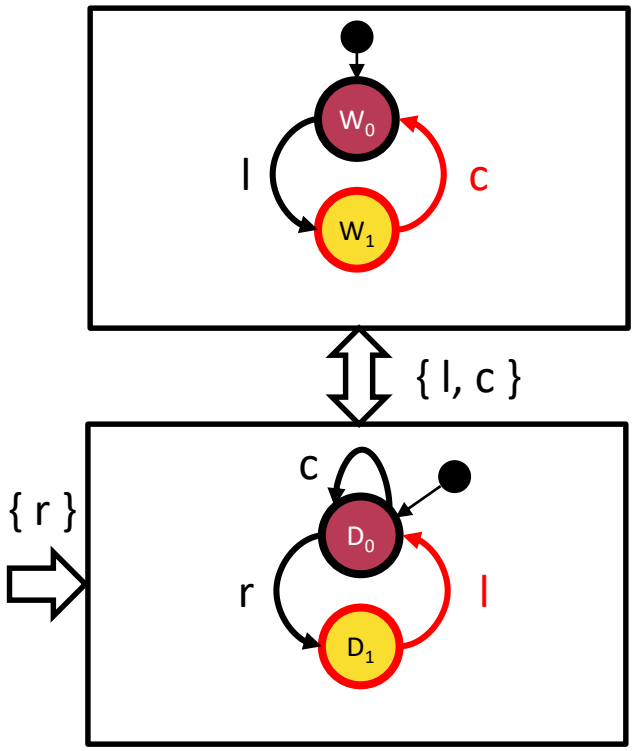
Forrás: https://medium.com/@elizarov/deadlocks-in-non-hierarchical-csp-e5910d137cc

Bsp.: Kommunizierende SW-Komponente



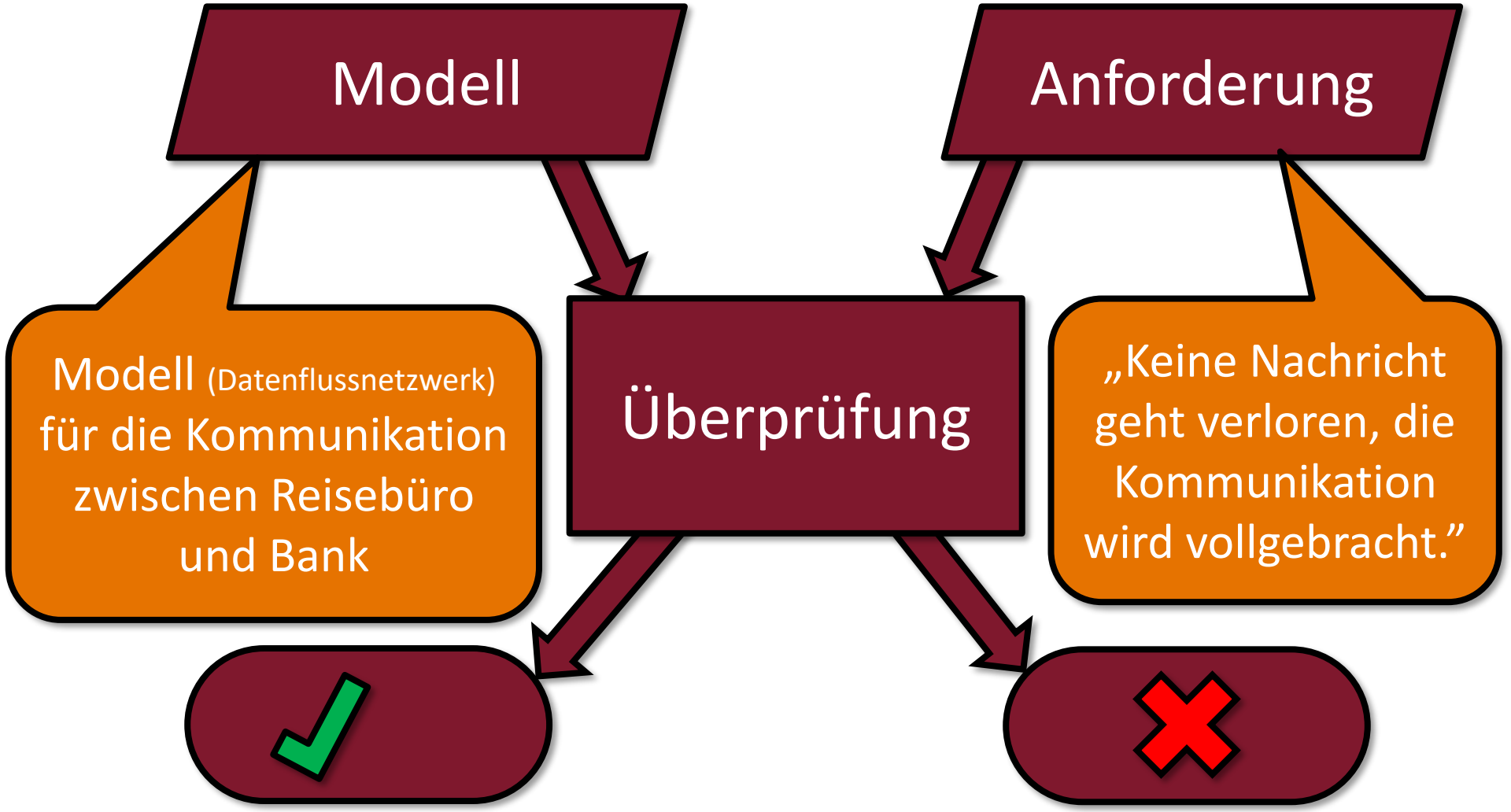

```
for (loc in locations) {          // (4)
  val content = downloadContent(loc)
  contents.send(                  // (5)
    LocContent(loc, content)
  )
}
```

```
while (true) {
  select<Unit> {
    references.onReceive {                    // (1)
      ref ->
        val loc = ref.resolveLocation()
        //...
        locations.send(loc)                   // (2)
      }
    contents.onReceive {                      // (3)
      (loc, content) -> //...
    }
  }
}
```



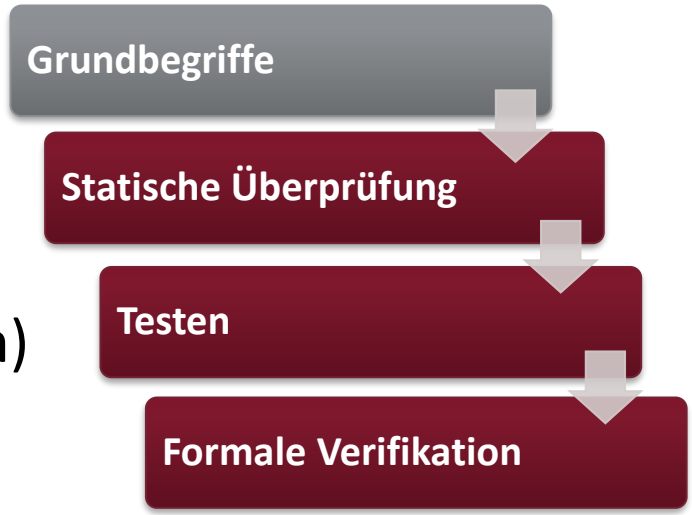
Forrás: https://medium.com/@elizarov/deadlocks-in-non-hierarchical-csp-e5910d137cc

Überprüfung von Modellen



Arten der Untersuchungen

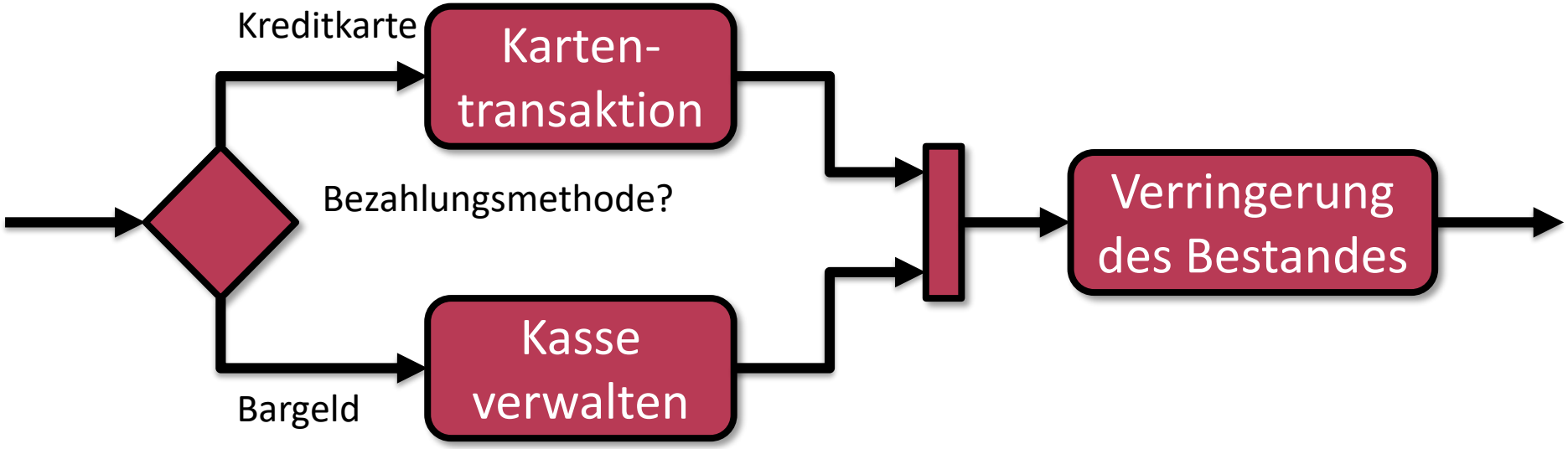
- Nach dem Zweck:
  - Verifikation: **Ist das System richtig gebaut?**
    - Ist die Implementation entsprechend der Spezifikation?
  - Validation: **Ist das richtige System gebaut?**
    - Erfüllt das System die Benutzeranforderungen?
- Nach der Methode:
  - Statische Überprüfung
  - Dynamische Überprüfung
    - stichprobenartig (Testen, Simulation)
    - ausführlich/vollständig (Überprüfung der Modelle)



STATISCHE ÜBERPRÜFUNG

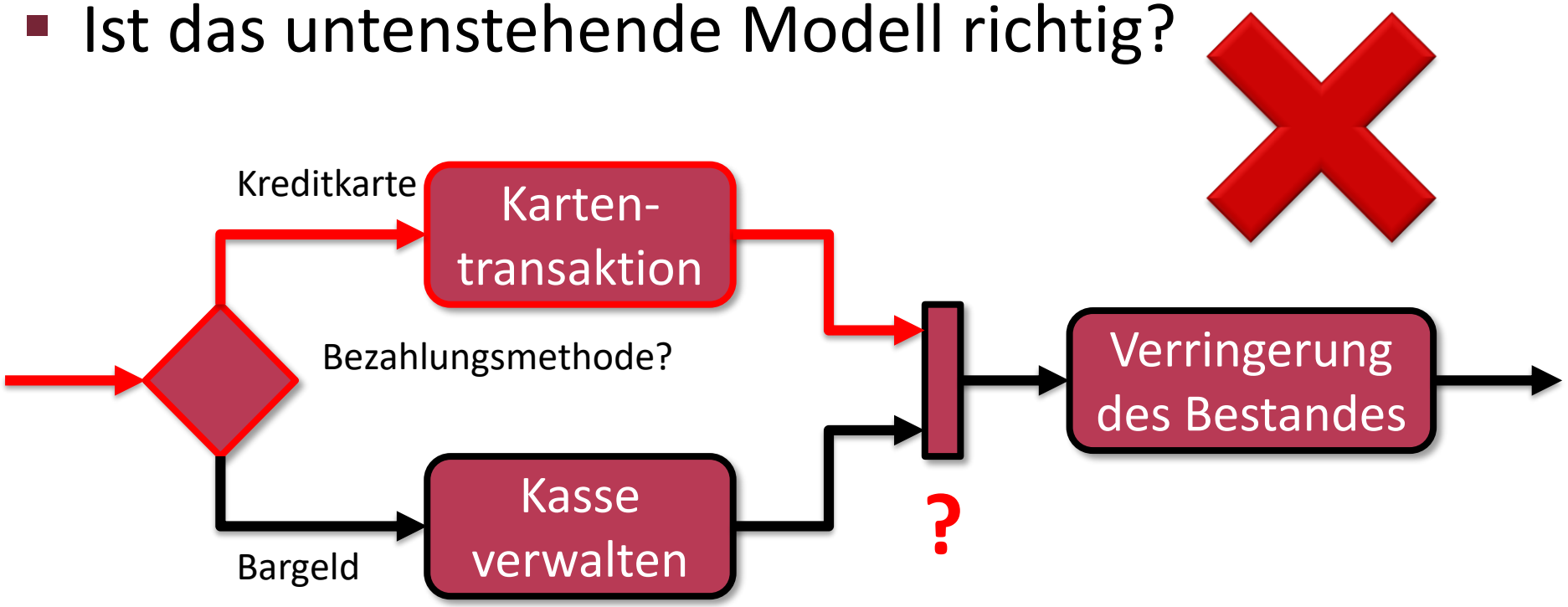
Decision und Join

- Ist das untenstehende Modell richtig?



Decision und Join

- Ist das untenstehende Modell richtig?

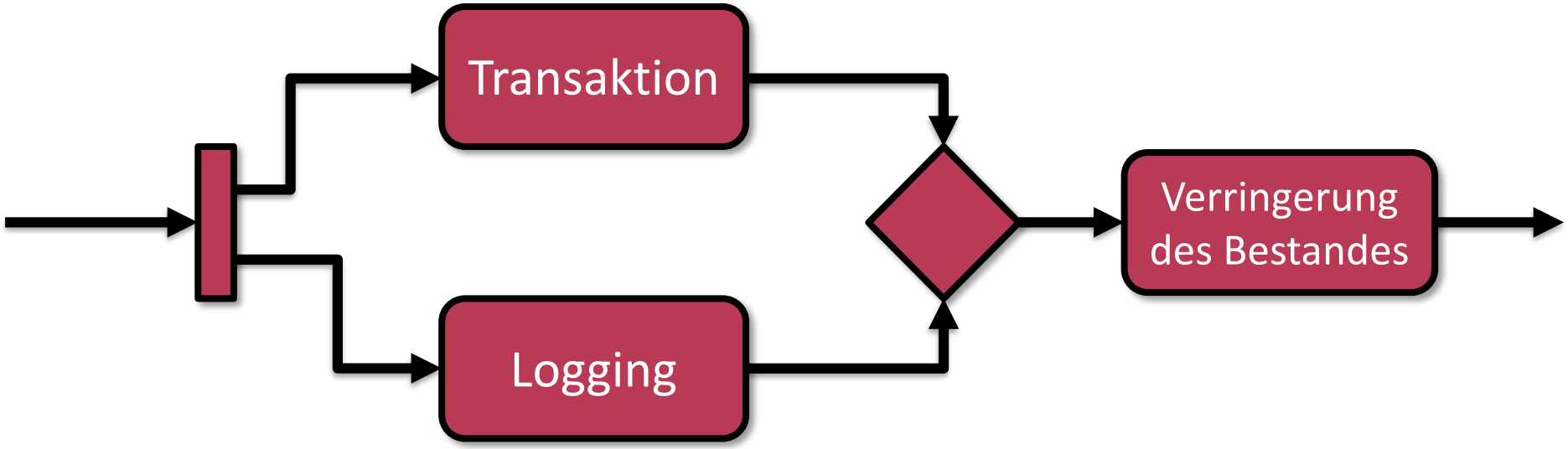


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

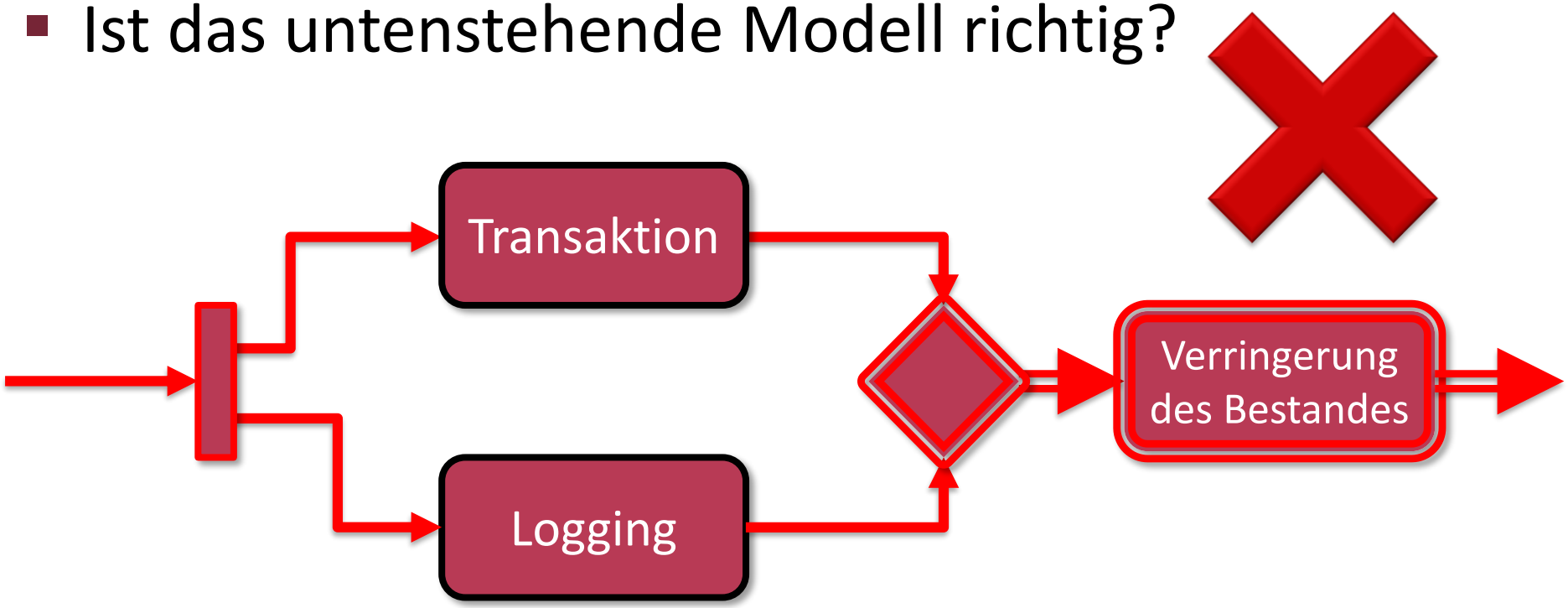
Fork und Merge

- Ist das untenstehende Modell richtig?



Fork und Merge

- Ist das untenstehende Modell richtig?

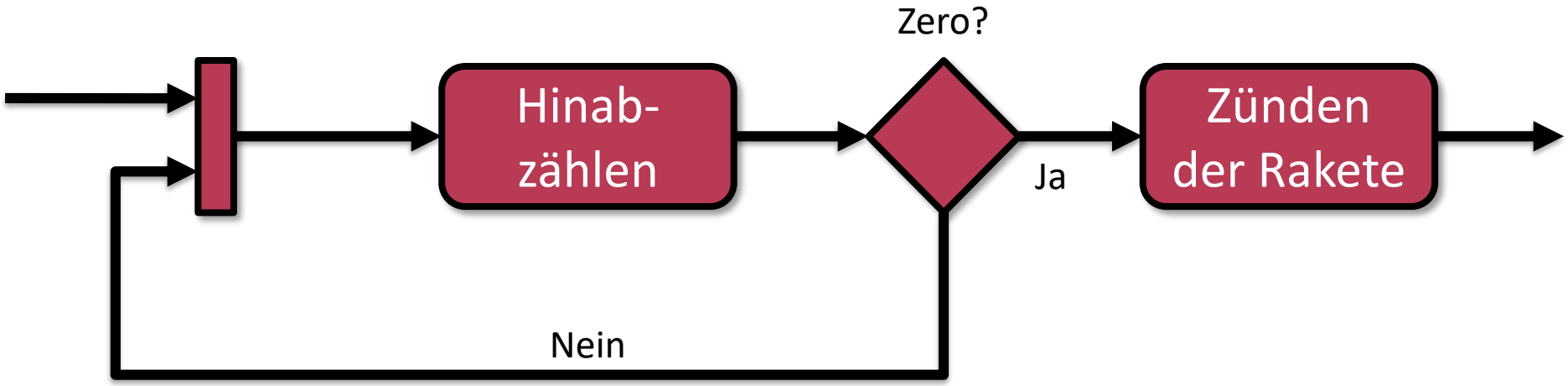


- Merge: Eine Fortsetzung ist möglich, sobald an einem Eingang ein Token angekommen ist
  - Keine Synchronisation der Faden

→ **DOPPELTE „VERRINGERUNG DES BESTANDES”**

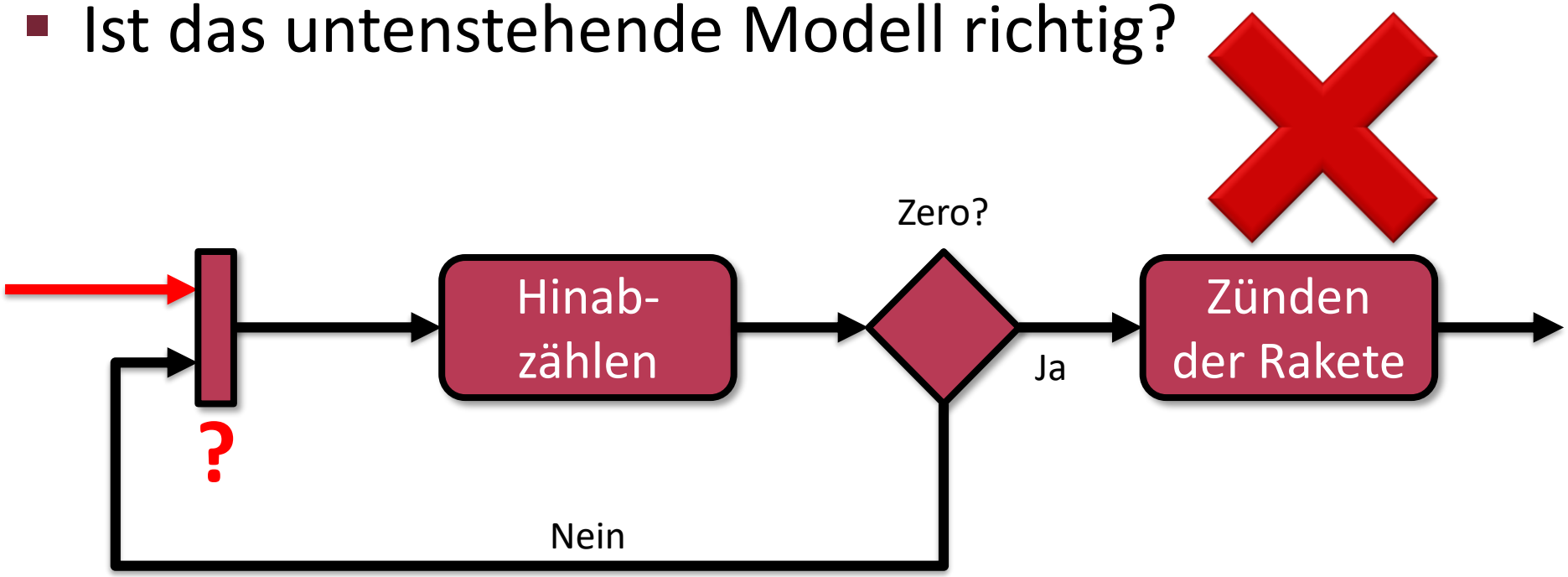
Schleife 1.

- Ist das untenstehende Modell richtig?



Schleife 1.

- Ist das untenstehende Modell richtig?

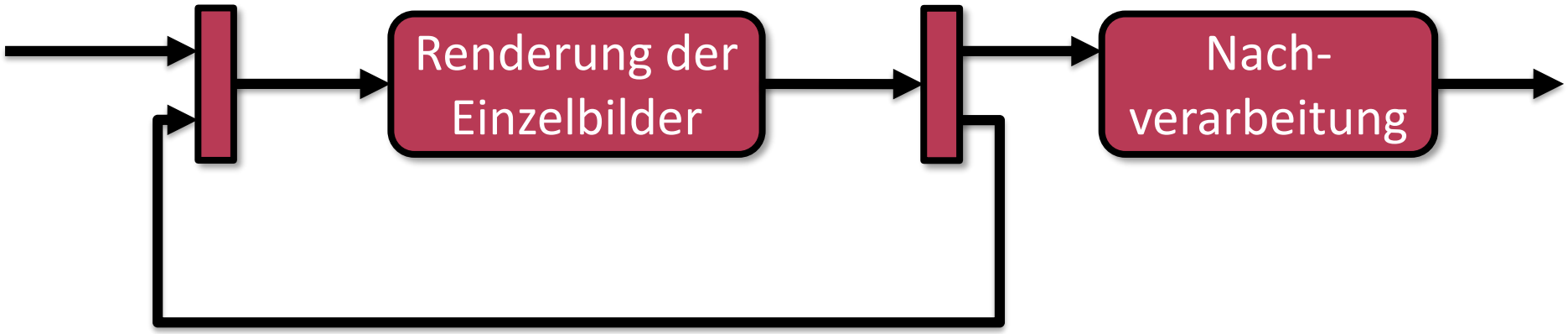


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

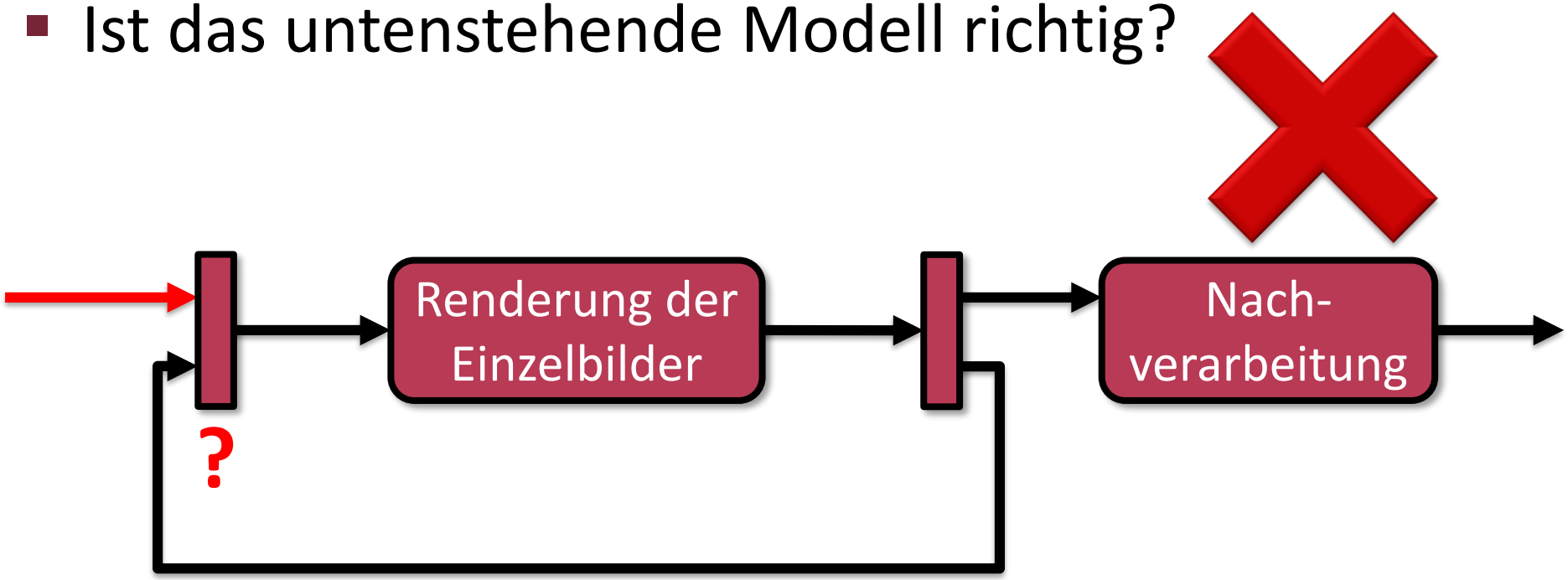
Schleife 2.

- Ist das untenstehende Modell richtig?



Schleife 2.

- Ist das untenstehende Modell richtig?

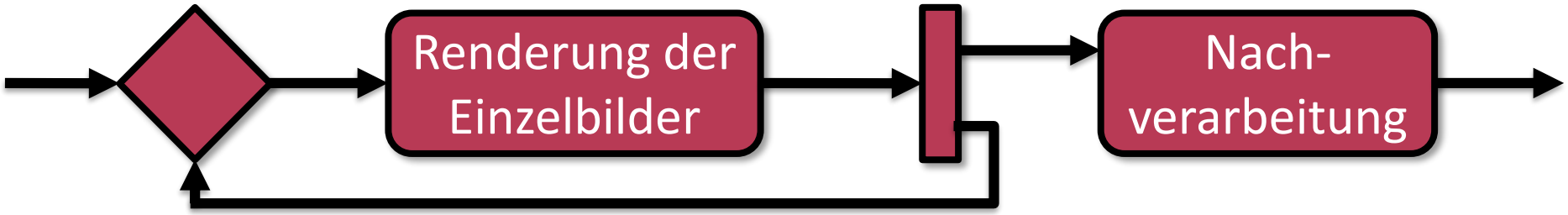


- Join: Eine Fortsetzung ist nur möglich, wenn an allen Eingängen ein Token angekommen ist

→ **VERKLEMMUNG (DEADLOCK)**

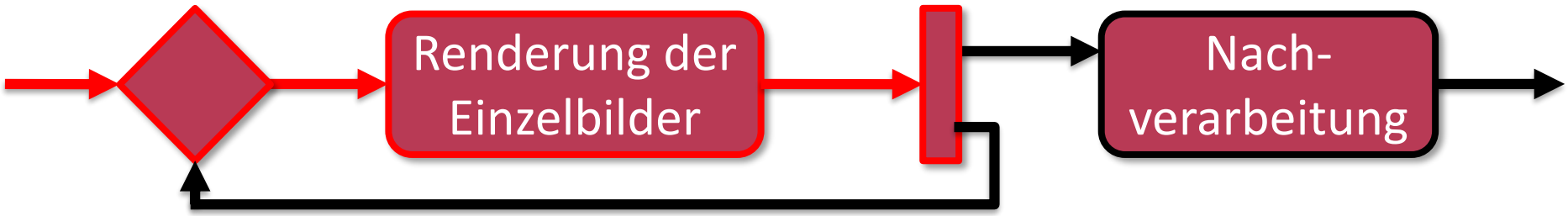
Schleife 3.

- Ist das untenstehende Modell richtig?



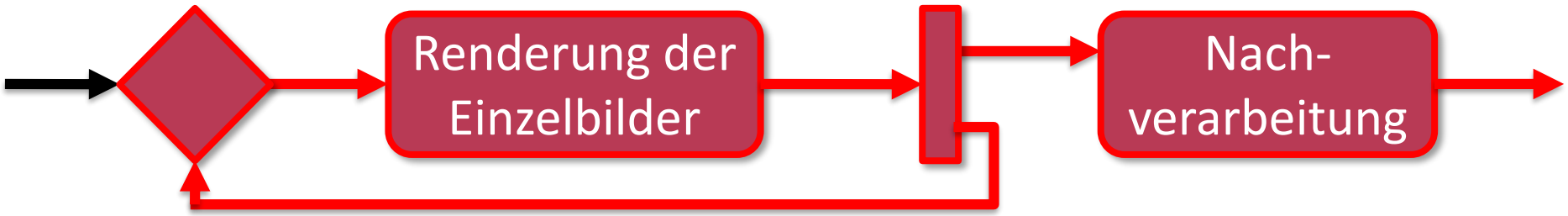
Schleife 3.

- Ist das untenstehende Modell richtig?



Schleife 3.

- Ist das untenstehende Modell richtig?

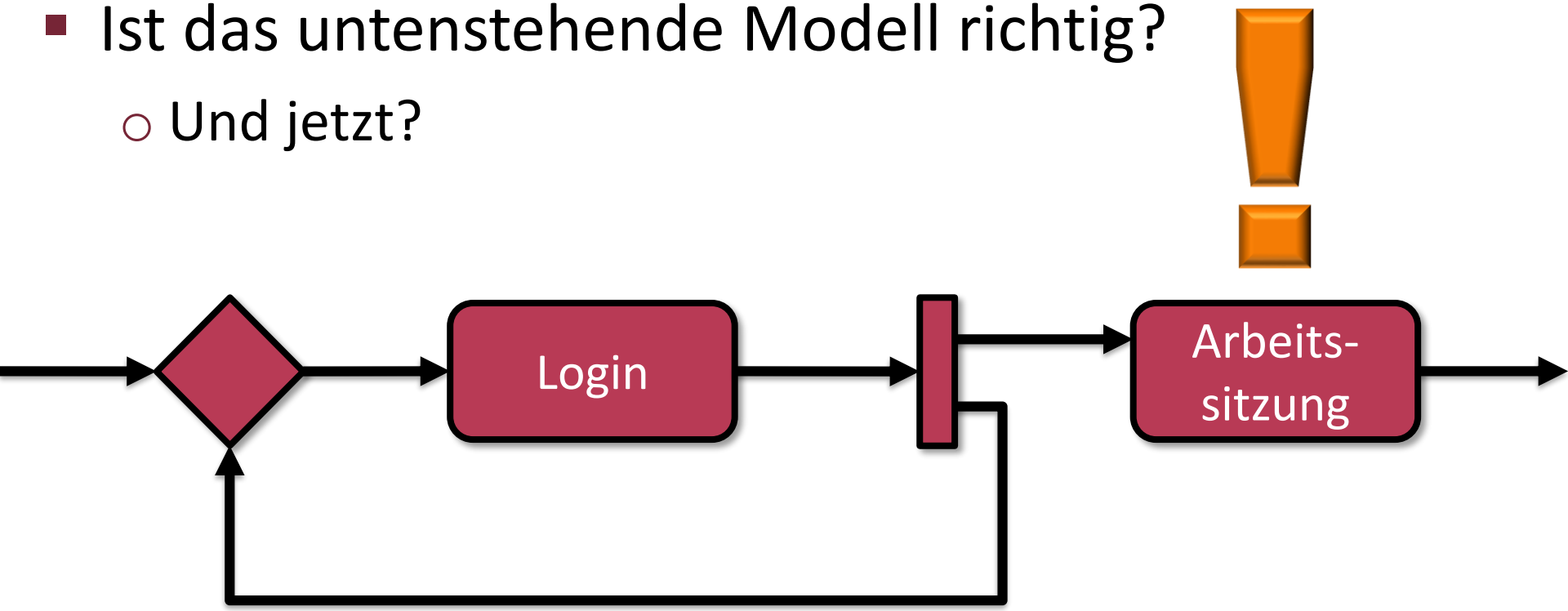


- Ein neues Bild in jeder Iteration
  - Nachverarbeitung jedes Bildes in neuem Faden (Wie viel mal?)

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

Ciklus 3.

- Ist das untenstehende Modell richtig?
  - Und jetzt?

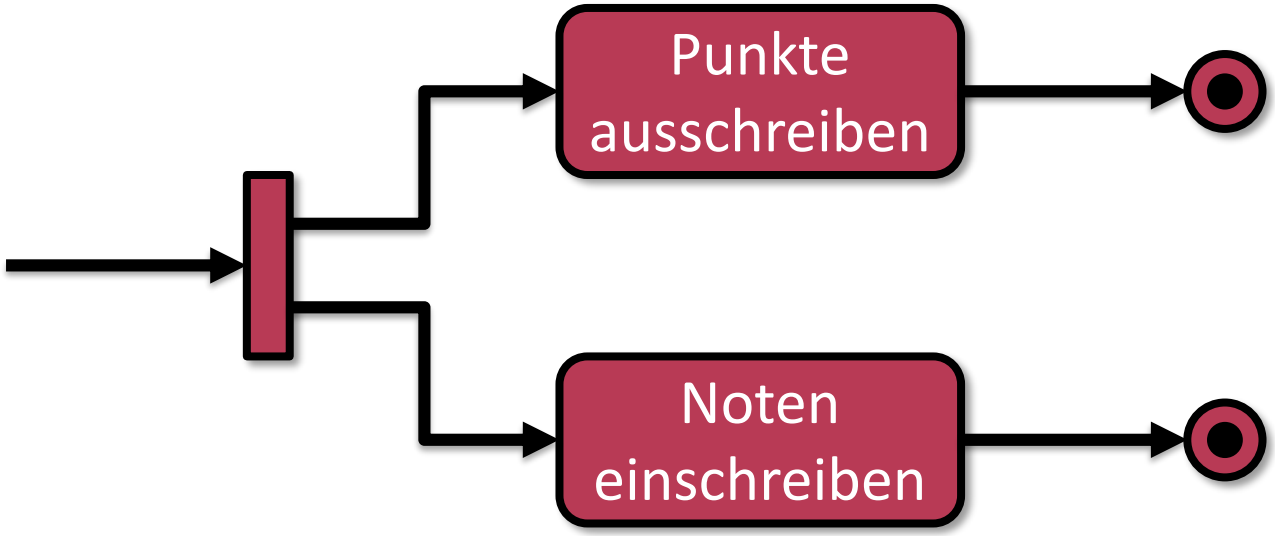


- Nach jedem Login ein weiters Login …
  - …und eine Arbeitssitzung (working session) …?

→ **Die fehlerhafte Implementierung „produziert” Threads**

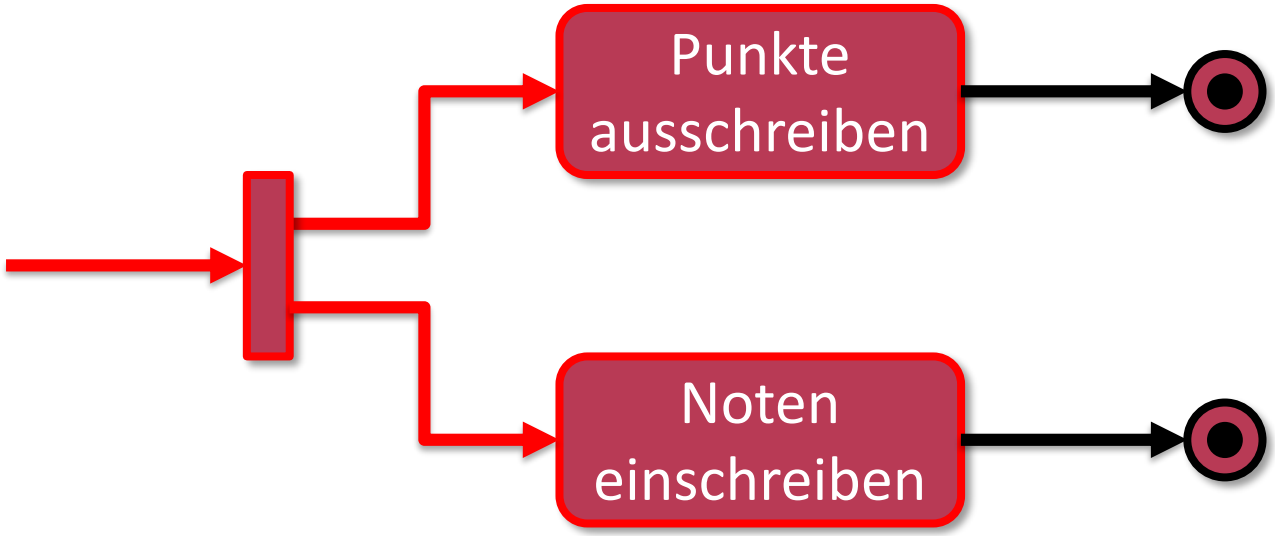
Endknotenpunkte

- Ist das untenstehende Modell richtig?



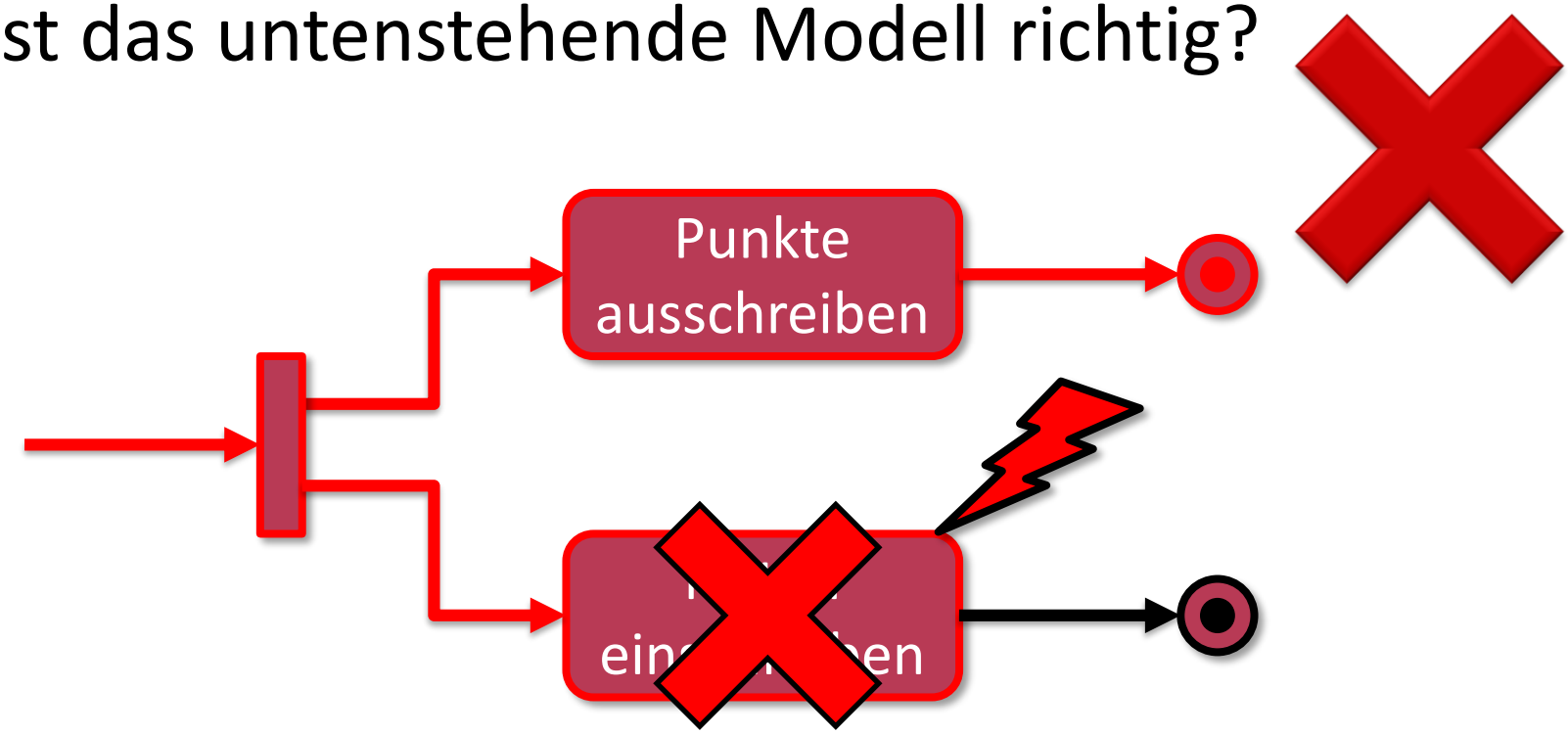
Endknotenpunkte

- Ist das untenstehende Modell richtig?



Endknotenpunkte

- Ist das untenstehende Modell richtig?

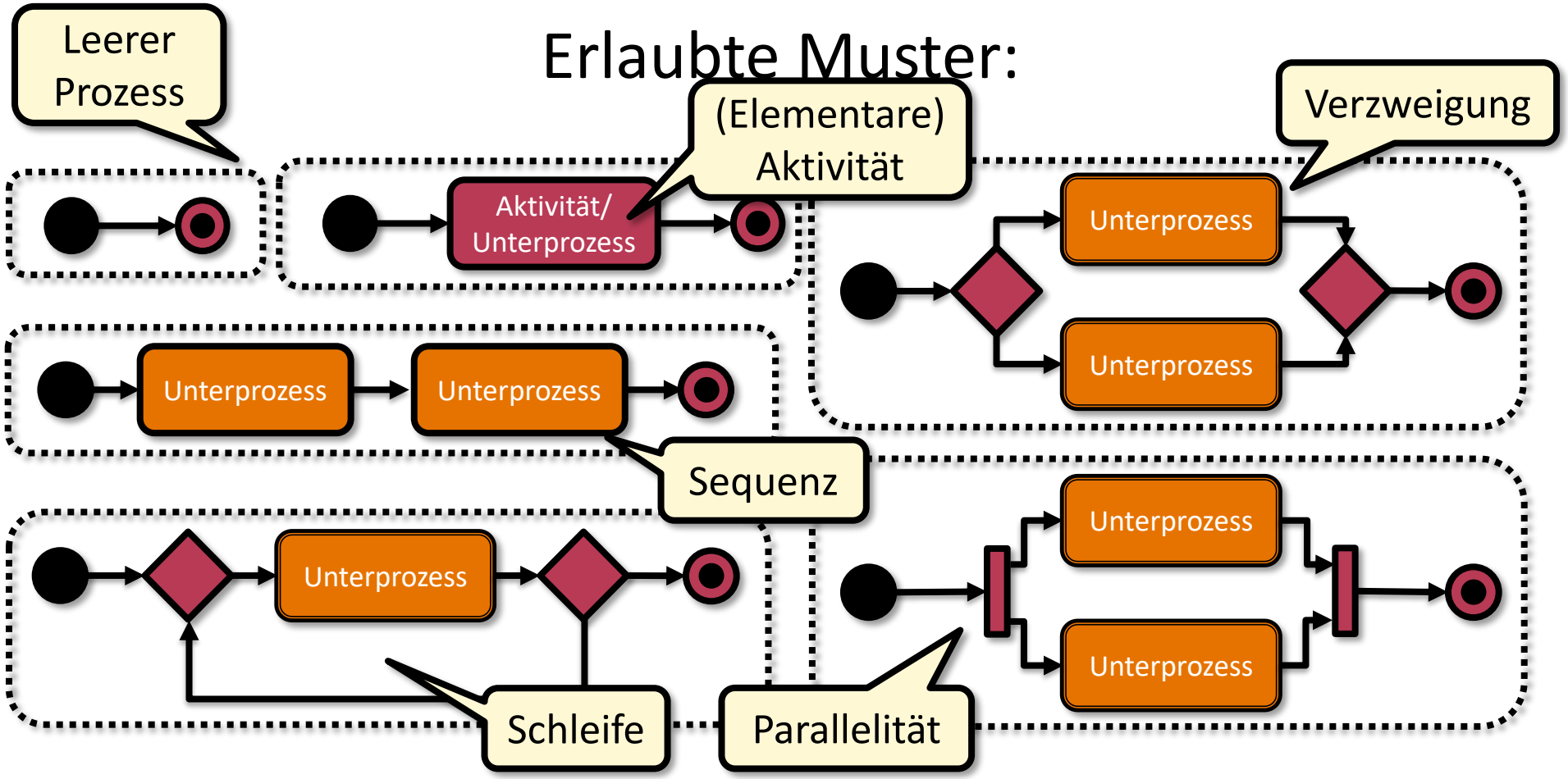


- Der terminierende Knoten stoppt den **ganzen** Prozess

→ **DIE ANDERE AKTIVITÄT WIRD NIE ABLAUFEN**

Wohlstrukturierte Prozessmodelle

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



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 Kode begleitet werden
    - „Ausführung im Kopf”

```
int mul(int a, int b) {
    return a * b;
}
```

- **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

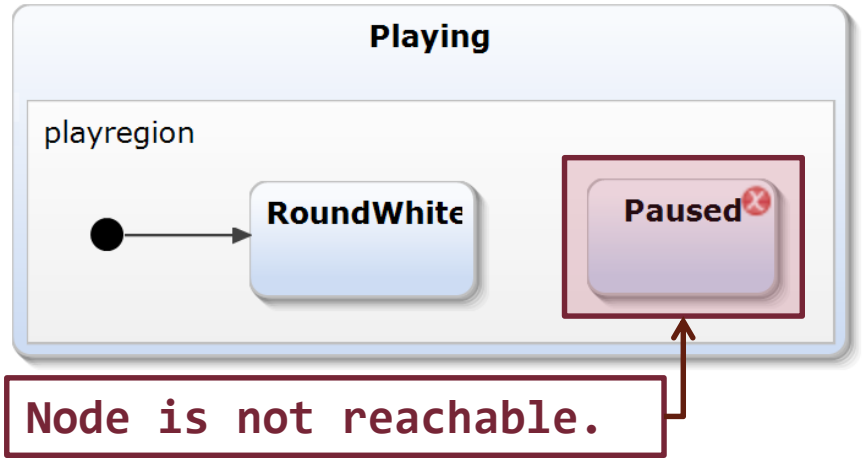
- Syntaktische Überprüfung: die Modellierungsumgebungen verbinden die logisch zusammengehörenden Elemente

- Syntaxgesteuerte Editoren
  - Fehler während Bearbeitung→ `Couldn't resolve reference`
  - Moderne Umgebungen: Vorschlagen der Kandidaten
- Kode+Diagram gemeinsam

- 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 Wertzuornungen, etc.

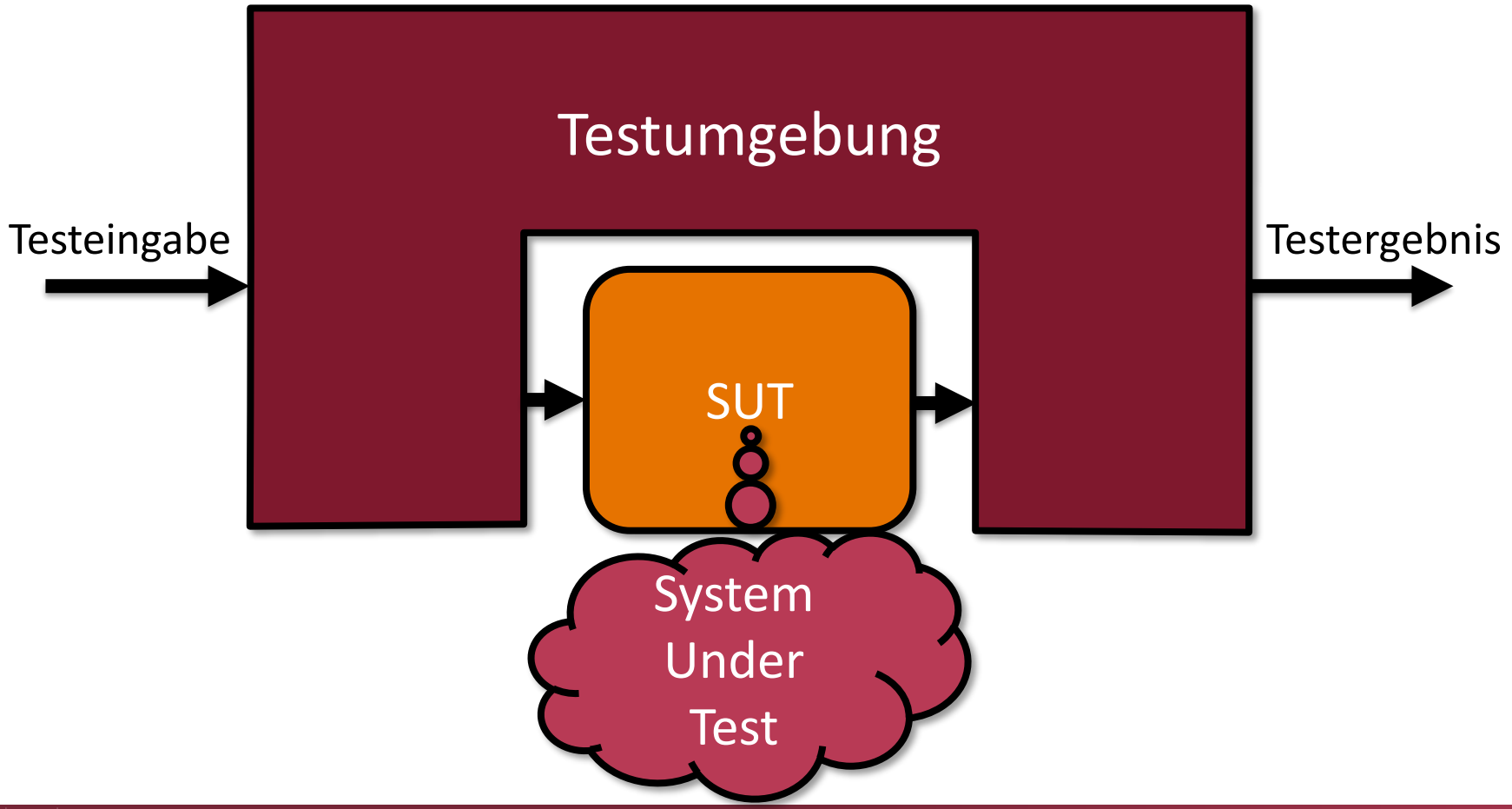
Statische Überprüfung: Unterstützung von Entwurfsrichtlinien

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

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

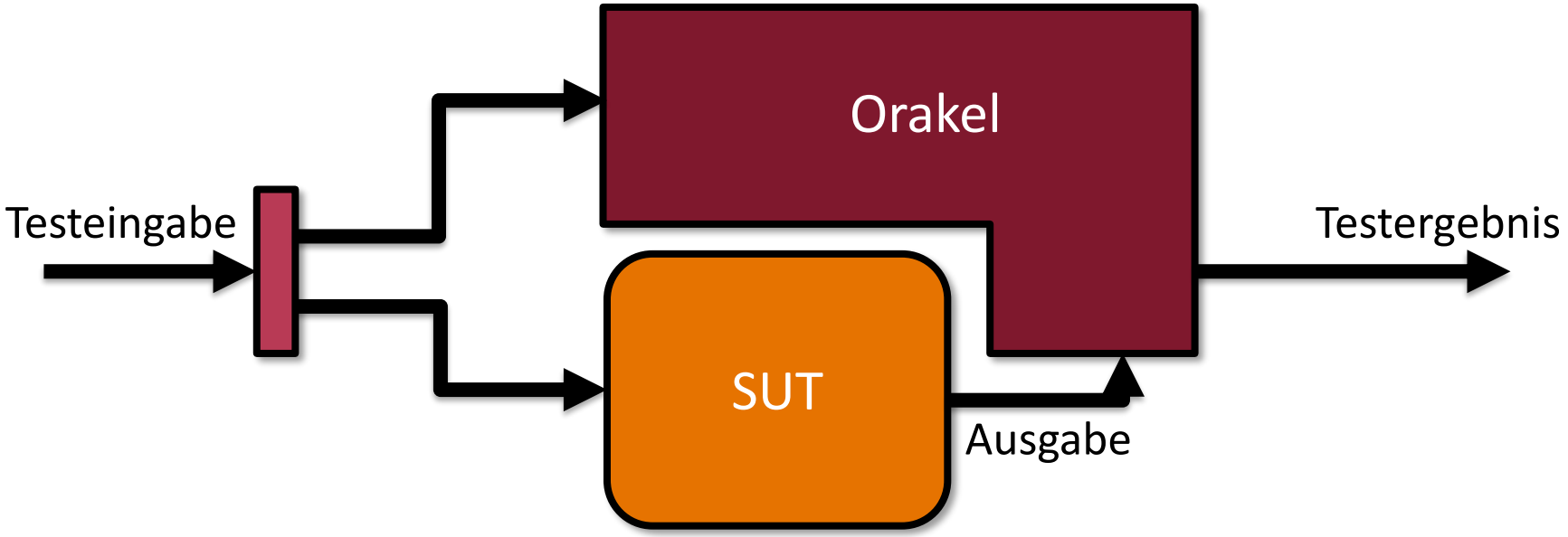
TESTEN

Testen von Modellen I.



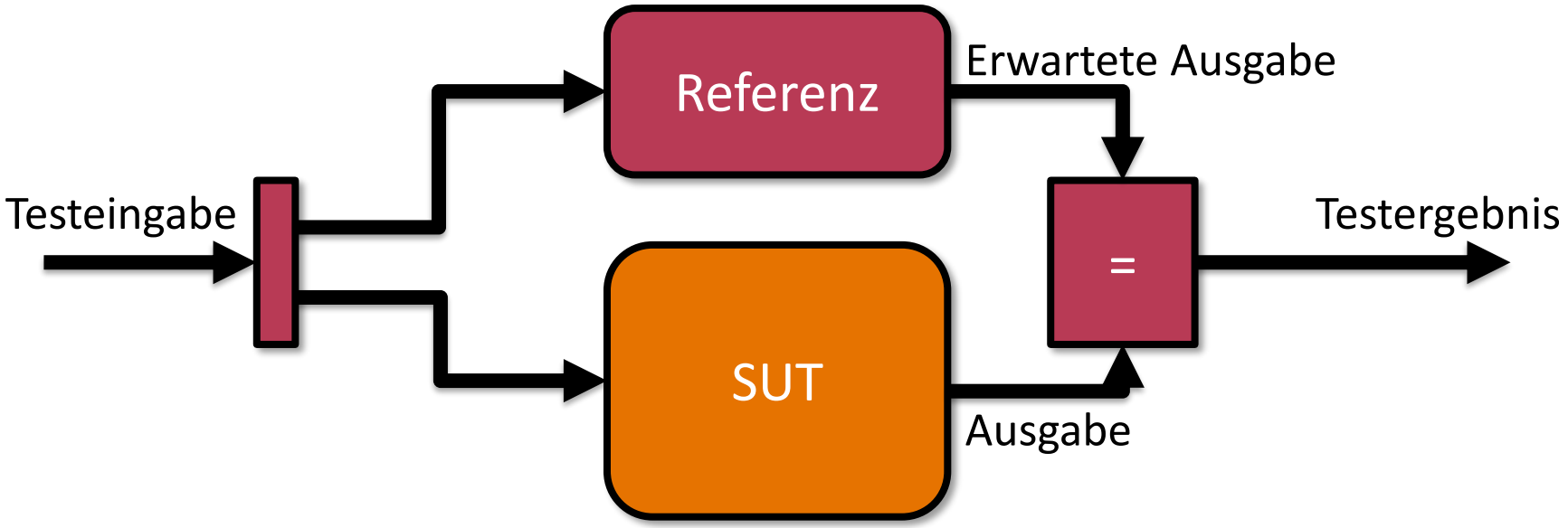
Testen von Modellen II.

- **Das Orakel:**
  Erzeugung und Vergleichung der erwarteten Ausgaben

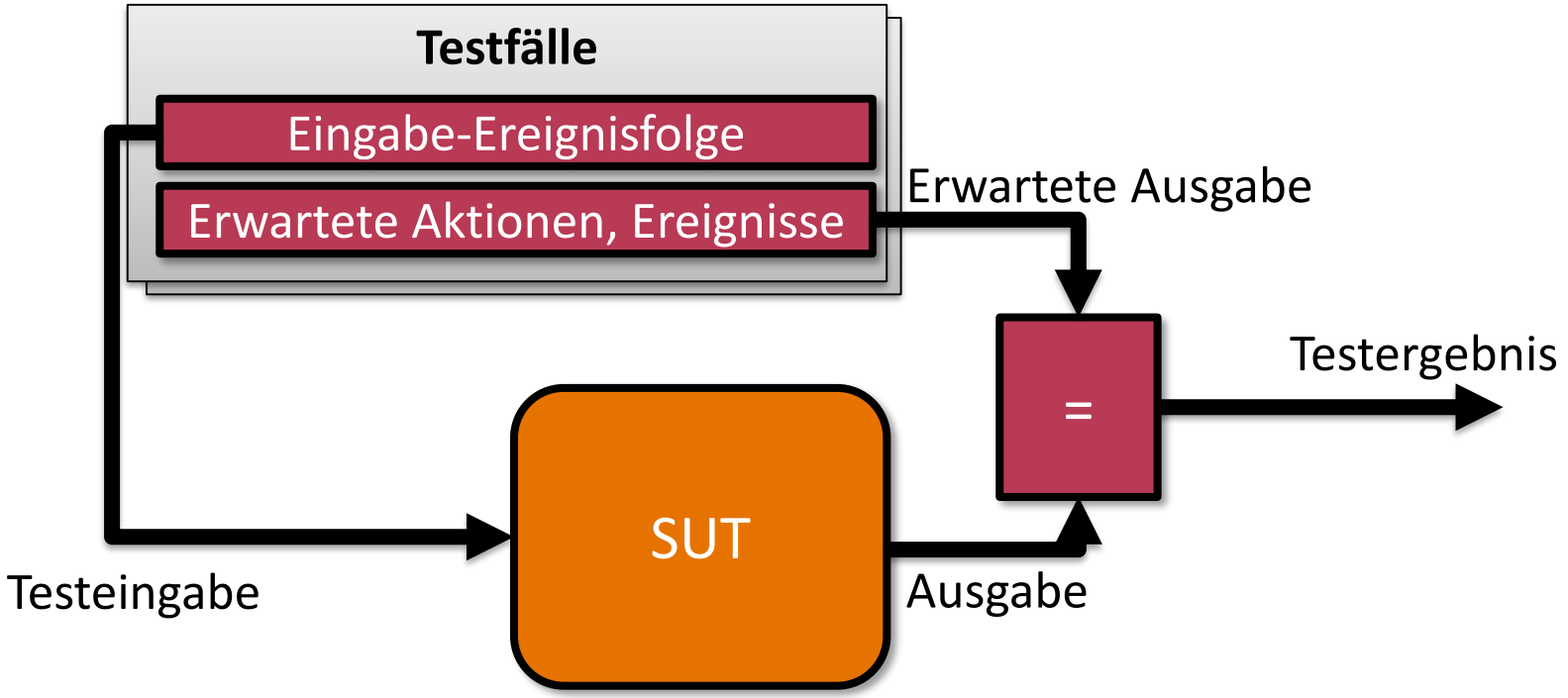


Testen von Modellen III.

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



Testen von Modellen: Yakindu Zustandsmaschine



Testen von Modellen: Yakindu Zustandsmaschine

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

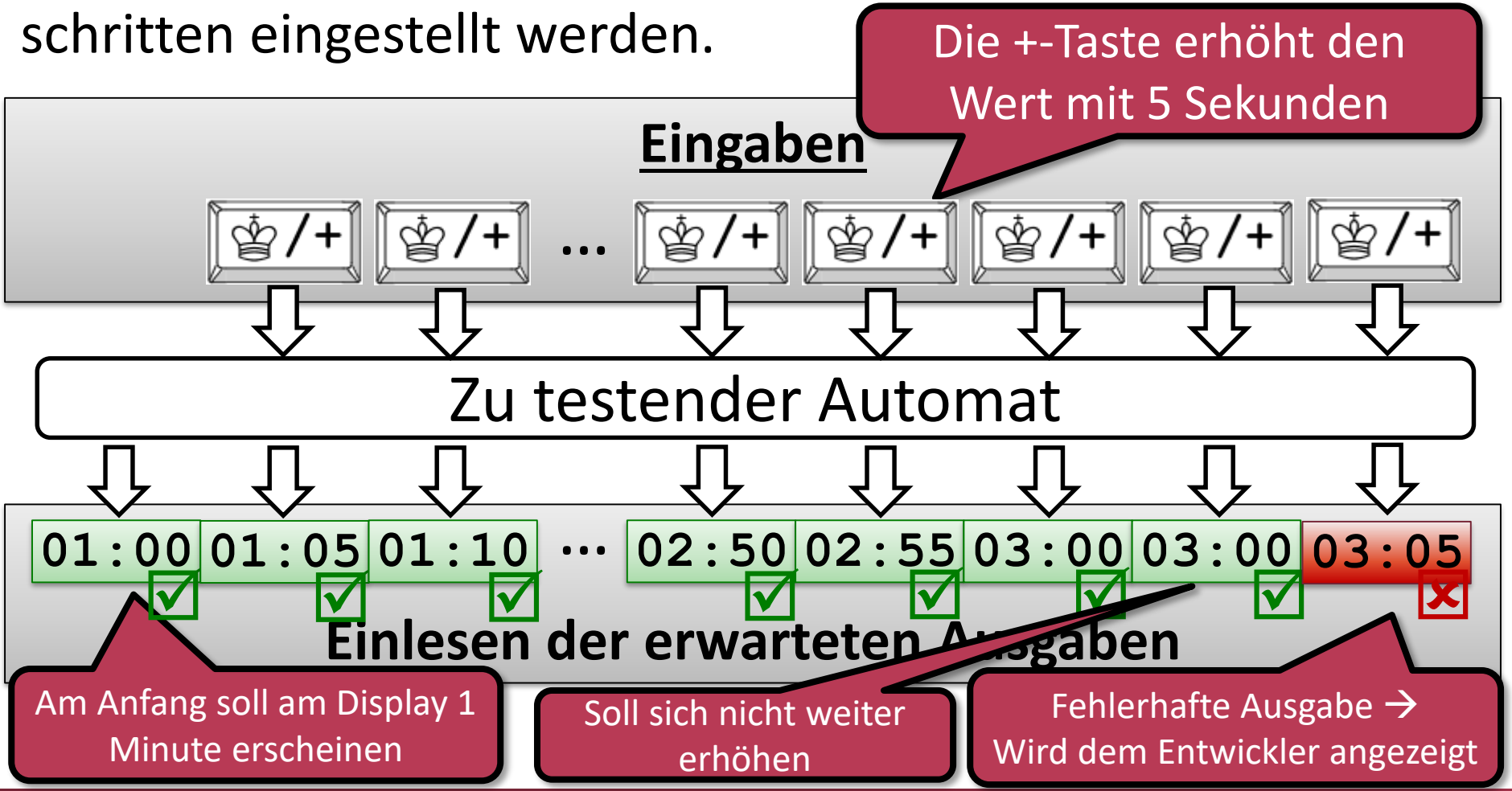
Eingaben

Zu testender Automat

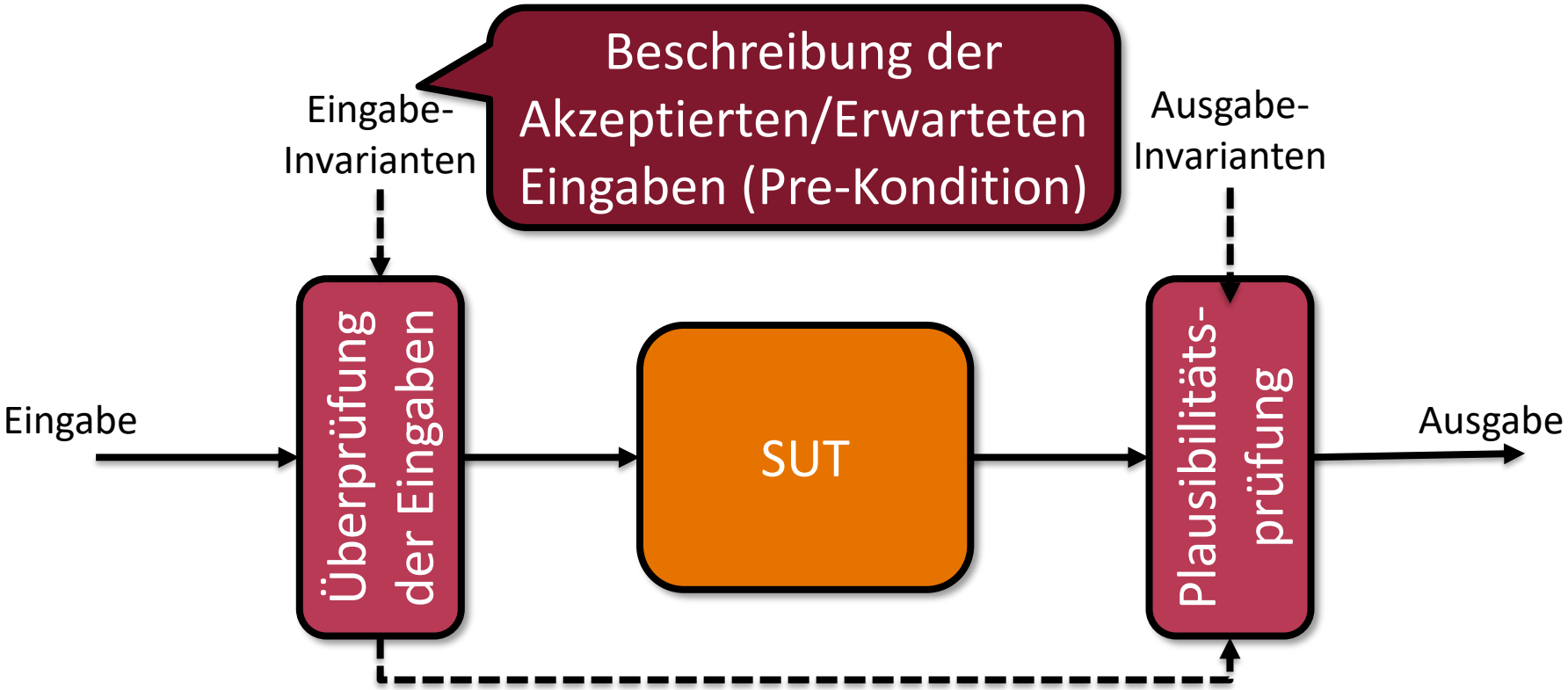
Einlesen der erwarteten Ausgaben

Testen von Modellen: Yakindu Zustandsmaschine

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

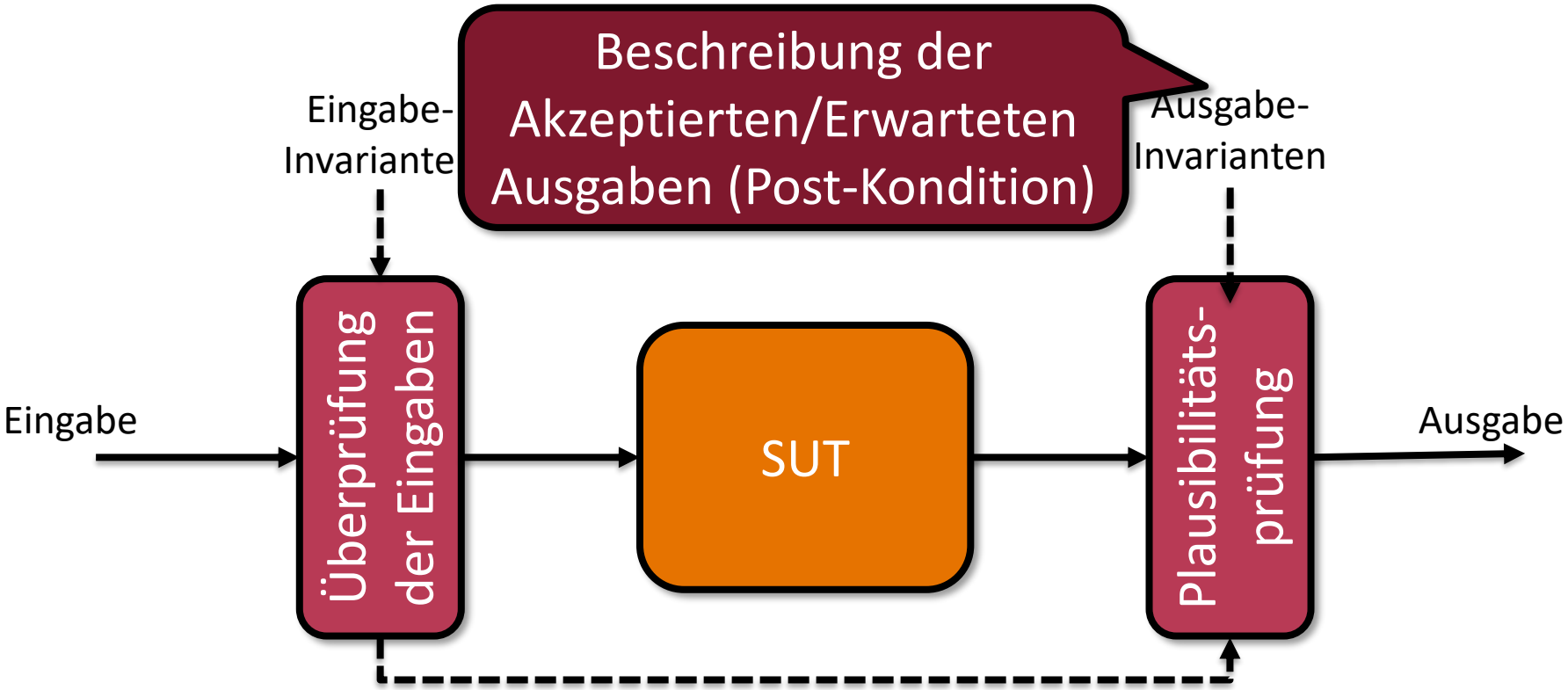


Selbsttest (monitor)



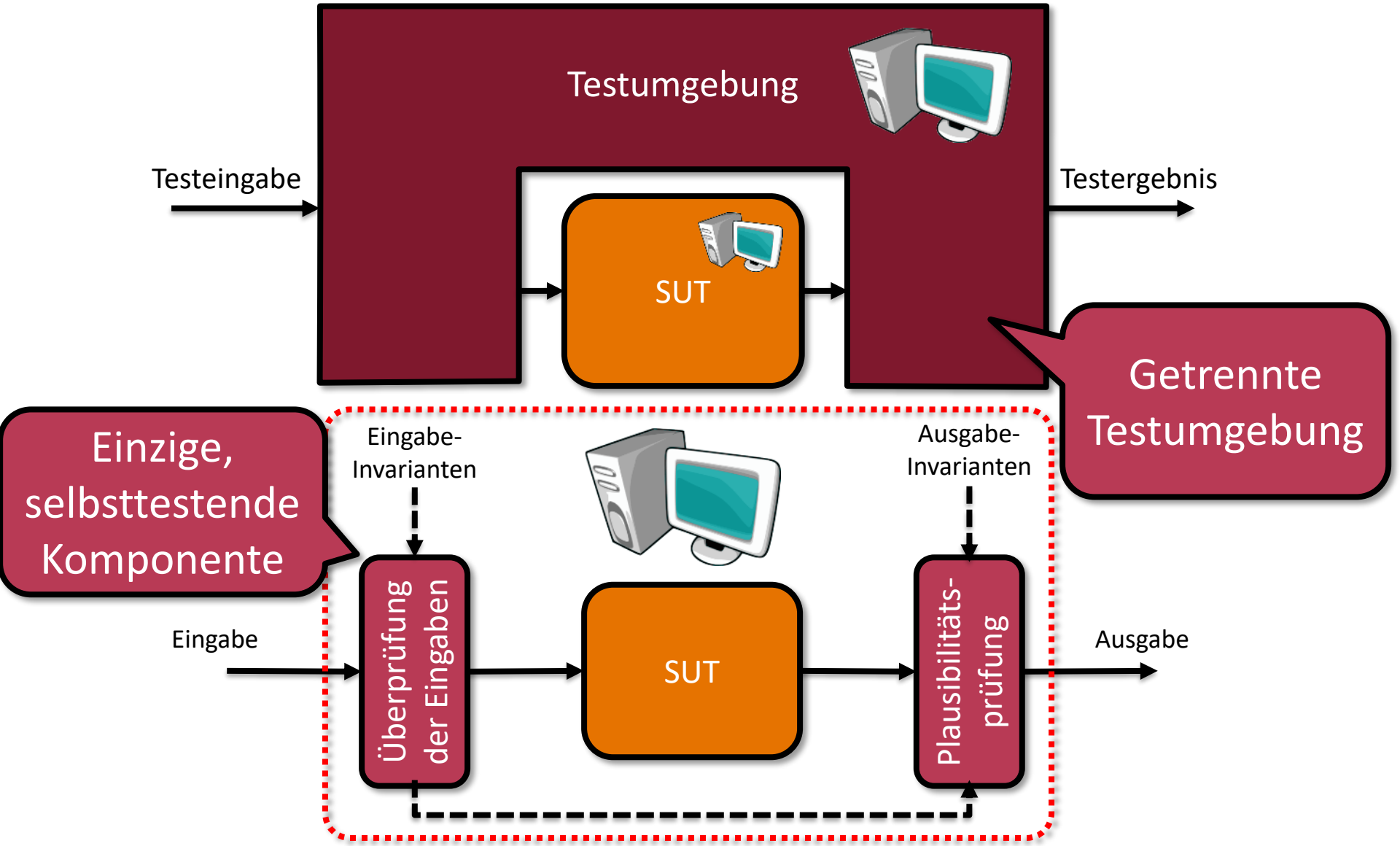
- **Invariante Eigenschaft:**
  sie soll fortlaufend wahr sein

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.

Assert (Behauptung):
Fehlerzustand, auf dessen Behandlung der Kode **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);
}
```

Testen von Modellen

- Ausführen des Modells: Simulation
  - Auf gegebene Eingaben überprüftes 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):

$$\frac{\textbf{Anzahl der berührten Zustände}}{\textbf{Anzahl aller Zustände}}$$

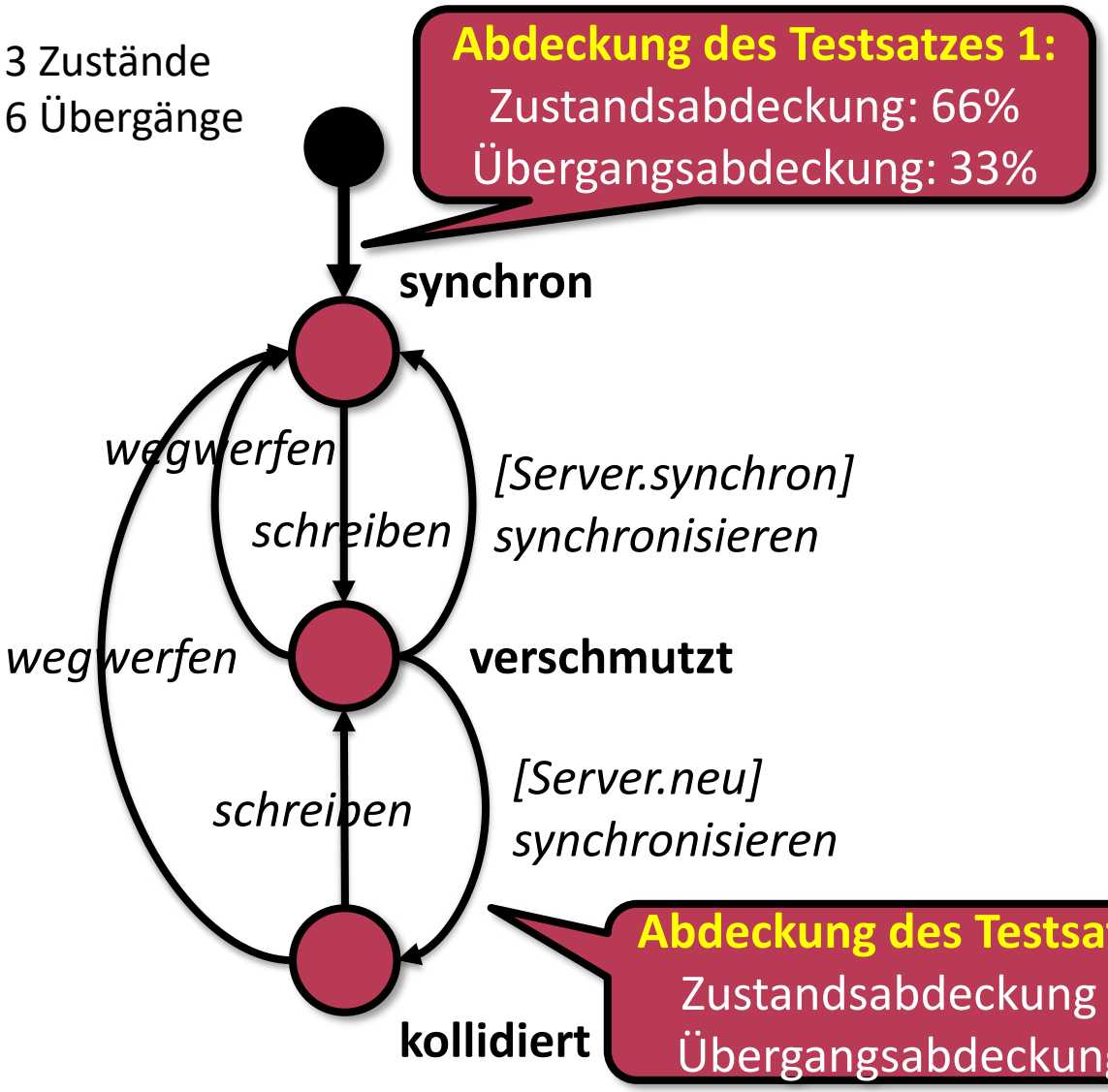
  - Übergangsabdeckung (in Zustandmaschinen):

$$\frac{\textbf{Anzahl der berührten Übergänge}}{\textbf{Anzahl aller Übergänge}}$$

  - Aktivitätsabdeckung (in Kontrollflüssen):

$$\frac{\textbf{Anzahl der berührten Aktivitäten}}{\textbf{Anzahl aller Aktivitäten}}$$

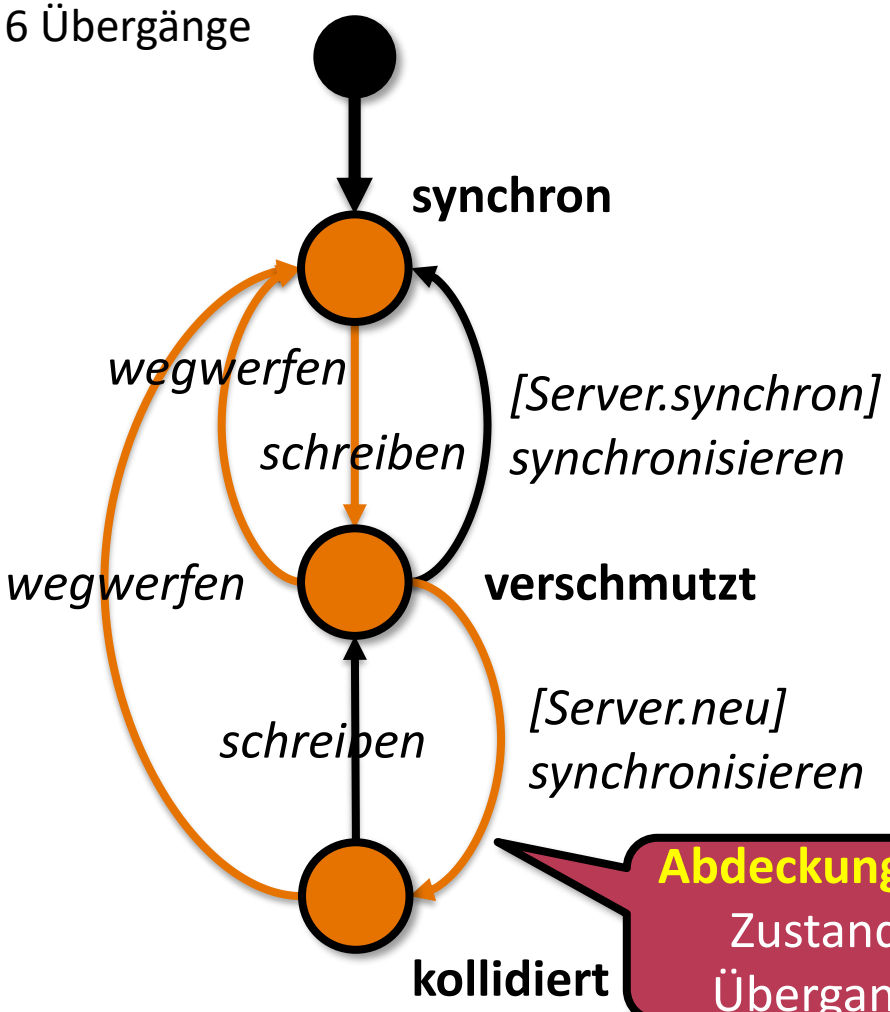
Beispiel: Wolkenbasierte Datenspeicherung



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:
   a) schreiben
   b) synchronisieren [Server = neu]
   c) schreiben
   d) synchronisieren [Server = sync.]

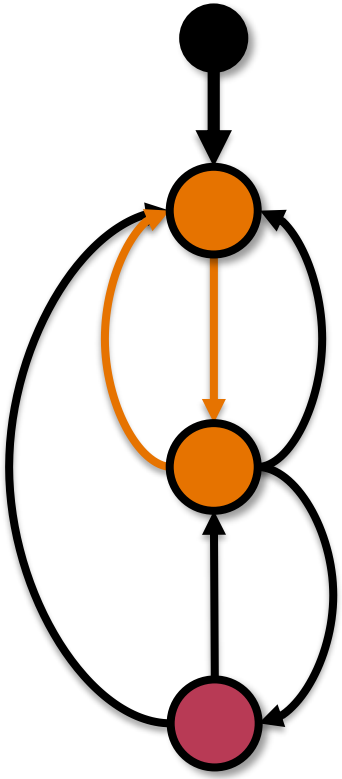
Abdeckung des Testsatzes 1+2+3:
Zustandsabdeckung : 100%
Übergangsabdeckung : 100%

Abdeckung

Nach dem 1. Testfall:

Zustandsabdeckung: 2/3=66%

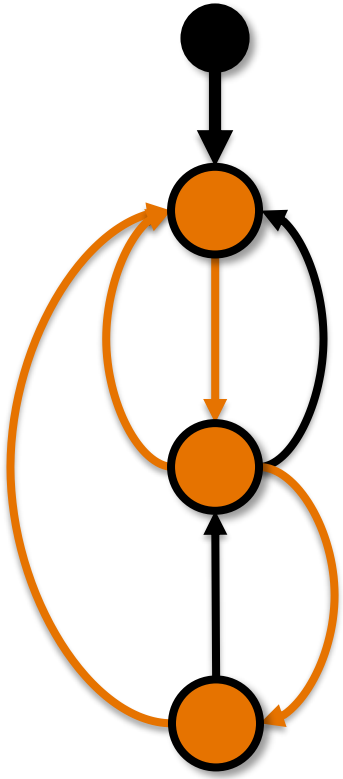
Übergangsabdeckung: 2/6=33%



Nach dem 2. Testfall:

Zustandsabdeckung : 3/3=100%

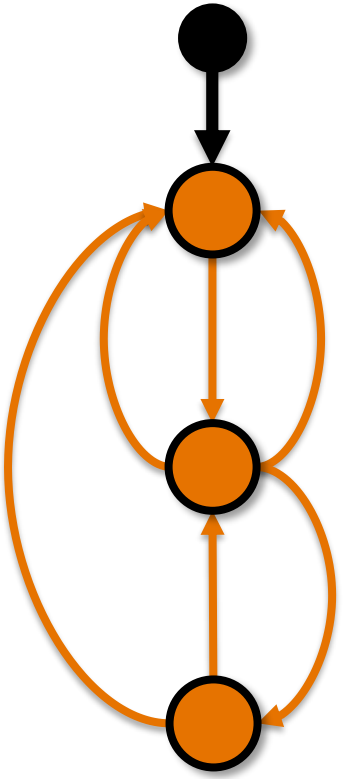
Übergangsabdeckung : 4/6=66%



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 (
  - Abdeckende Testeingaben (als Eingabe)
  - Vom Modell produzierte Ausgaben (als erwartete Ausgaben)
- **Monitoring:** Simulieren des M
während Laufen der Software
  - Gleiche Eingaben für das Modell und das Programm
  - Vergleichung der Ausgaben → **Fehlererkennung**
- **Untersuchung der Aufzeichnu**
  - Monitoring mit den aufgezeichneten n

Vor der Ausführung

Während der Ausführung

Nach der Ausführung

Testdokumentation

- Die Testfälle und –Ergebnisse sollen dokumentiert werden!

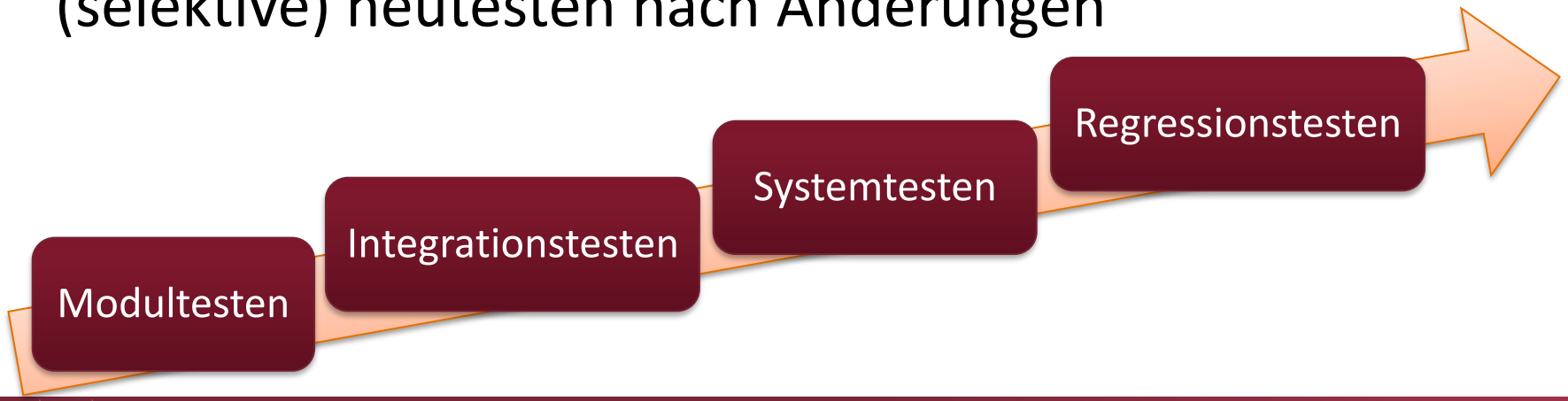
Testspezifikation
- Was wird getestet?
- Anhand welcher Anforderungen?
- Mit welchen Eingaben?
- Welche Ausgaben sind erwartet?

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



FORMALE VERIFIKATION

Formale Verifikation

- **Formale Verifikation:** Beweis der Korrektheit von Modellen/Programmen mit matematischen 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

Modellprüfung

- **Modellprüfung:** Erschöpfende (vollständige) Untersuchung aller potentiellen Verhalten eines Modells anhand gegebener Anforderungen
  - Suchen nach Fehlverhalten → **Gegenbeispiel**
- **Zustandsraumgeneration:** Aufdeckung aller möglichen Verhalten
  - In der Praxis: Berechnung gemischter Produkte
- **Anforderungen:** erwartete Eigenschaften des Zustandsgraphen
  - In der Praxis: temporal-logische Ausdrücke

Zustandsraumgeneration

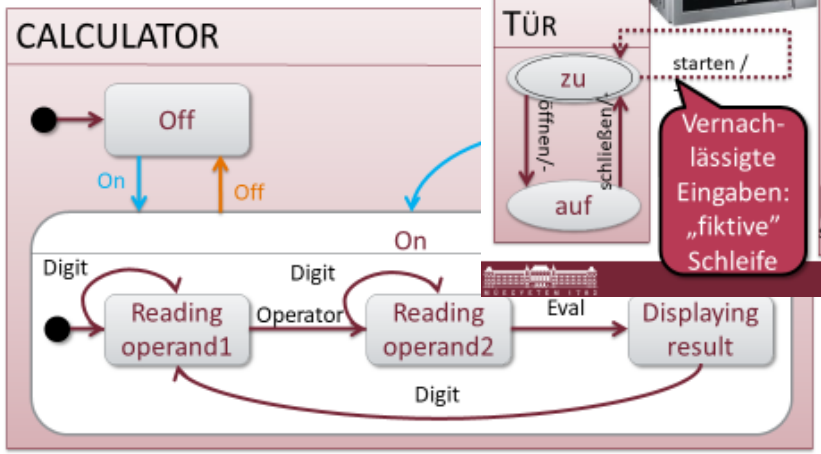
- Generierung einer einzigen *einfachen Zustandsmaschine* von komplexeren Modellen

Variablen + Wächterkriterien

- Zyklischer Zähler
  - $S = \{0,1,\dots,i\}$

- mit Wächterkriterien :

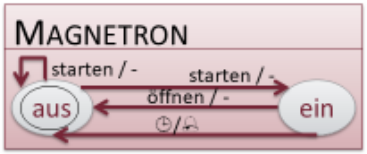
Statechart-Beispiel: Zus

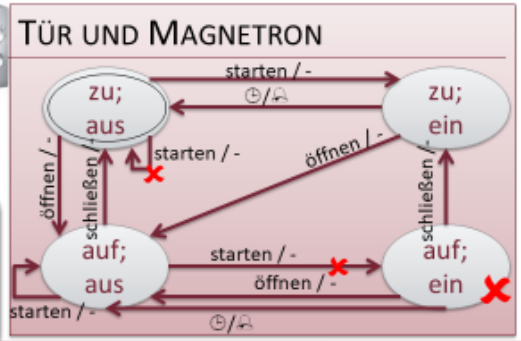


- Das gemeinsame Verhalten wird abgetrennt

Beispiel: Gemischtes Produkt

- Verfeinerung notwendig
  - Überg. können wegfallen
  - Zustände können auch unerreichbar werden





Anforderungen: Temporale Logik

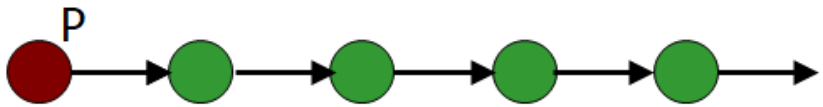
- **Zustandsmenge** beschreiben:
  durch Prädikatenlogik
  - z.B.: „$\Delta T < 0$” oder „Temperatur > 5” (P, Q Predikate)
- Eine **Sequenz von Zuständen** beschreiben:
  durch (lineare) temporale Logik

Anforderungen: Temporale Logik

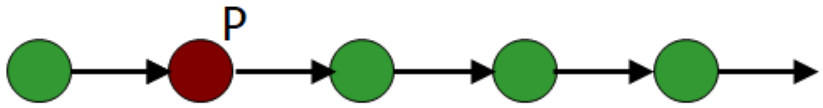
- Eine **Sequenz von Zuständen** beschreiben:
  durch (lineare) temporale Logik

Wann soll P (und Q) wahr sein?

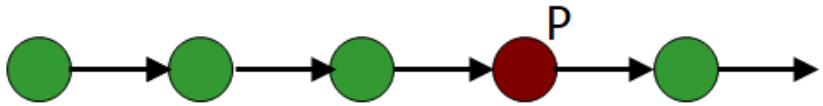
- im aktuellen Zustand — P



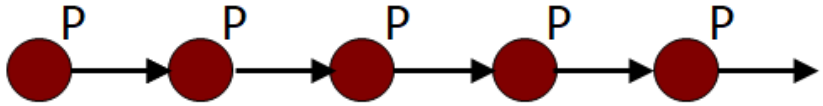
- im nächsten Zustand — X P



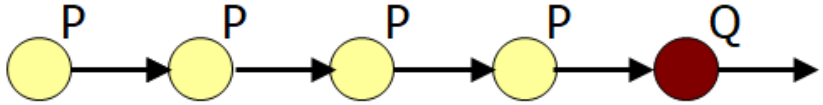
- in einem zukünftigen Zustand — F P



- in jedem Zustand — G P



- P immer *wahr* bis Q *wahr* wird — P U Q



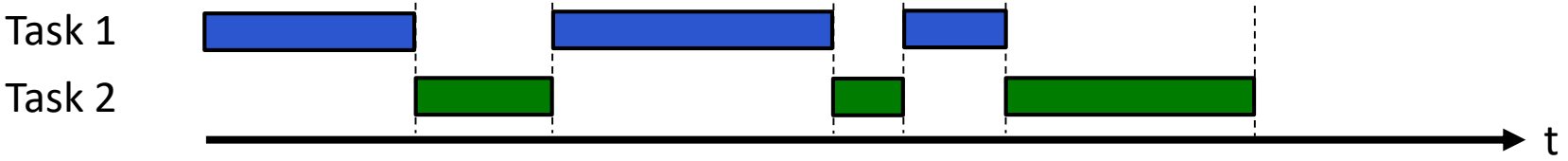
SOFTWARE VERIFIKATION

Ziel der Software Verifikation

- **Modellierung von Programkode durch Zustandsmaschinen**
  → Formale Verifikation von Software

- Anforderungen:
  - Ausfürung der Anweisungen – atomare Aktivität
  - Auswertung der (einfachen) Bedingungen – atomare A.
  - **Alles andere ist unterbrechbar**
    - Während der Ausführung kann das OS jederzeit stoppen und zu einem anderen Task oder Thread wechseln.

Nicht-unterbrechbar



Modellierung von Programkode: CFA

- **Control Flow Automata**
  - (Uhrtick getriggerte) Zustandsmaschine
    - mit Variablen und Kode-specifischen Elementen
- **Knoten** (*locations*)
  - Je ein (Kontroll-)Knoten zwischen jeder zwei Anweisungen
    - Wie ein *breakpoint* beim debugging (*„Welche ist die nächste Zeile?”*)
  - Spezieller *„end”*-Knoten für das ProgrammEnde
  - Spezieller *„error”* -Knoten für die *assert* Anweisung
- **Kanten** (*transitions*)
  - Modelliert die Auswertung der Bedingungen (*if*, *while*)
  - Modelliert die Ausführung der Anweisungstransaktionen (Atomizität!)
  - Hier an einer Kante:
    - eine Bedingung **oder** eine einzige Anweisung

CFA-Generierung vom Programkode – Beispiel

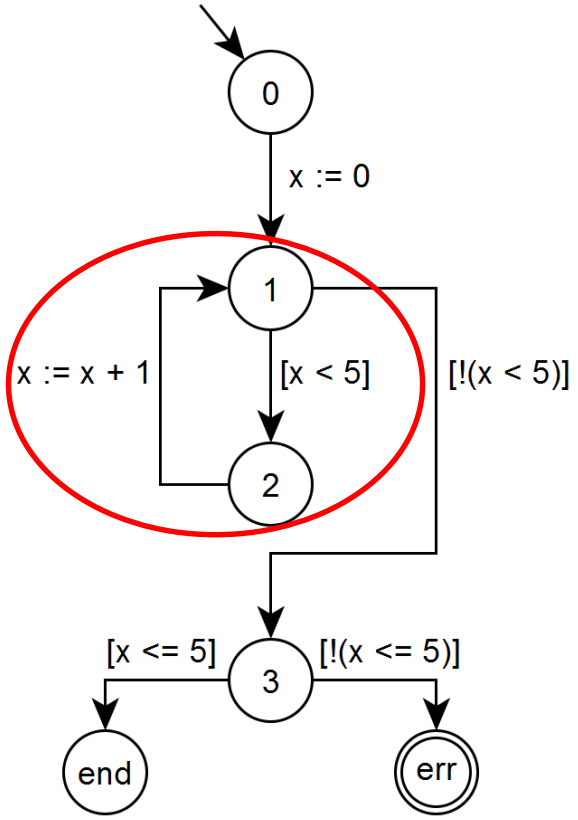
1. Gib eine *assert*-Anweisung zum Programm
2. Anfangsknoten – vor der ersten Kodezeile
3. Erste Anweisung
   - Ein neuer Knoten nach der Anweisung
     - hier: Anfang der nächsten Zeile
   - Eine Kante mit der Anweisung in der Aufschrift
4. While – Auswertung der Bedingung
   - When *wahr*, „einspringen”
   - When *falsch*, „überspringen”
5. Schleife – „zurückspringen” nach Ausführung
6. Assertion
   - When nicht erfüllt: Fehler – spezieller Knoten
   - When erfüllt : kein Fehler – nächste Anweisung
7. When keine Anweisung mehr – Endknoten

```
0: int x=0;
1: while (x<5) {
2:    x=x+1;
   }
3: assert (x<=5)
```

Trennung der Bedingungen und Anweisungen

- Warum könnte eine einzige Aufschrift eine **Bedingung** und auch eine **Anweisung** beinhalten?
  - Ausführung kann jederzeit gestoppt werden, auch zwischen
    - Auswertung der Bedingung und
    - Ausführung der Anweisung.
  - Inzwischen kann der Wert von *x* von einem anderen Thread modifiziert werden
    - (siehe Bsp. am Ende der Vorlesung)

```
0: int x=0;
1: while (x<5) {
2:    x=x+1;
   }
3: assert (x<=5)
```



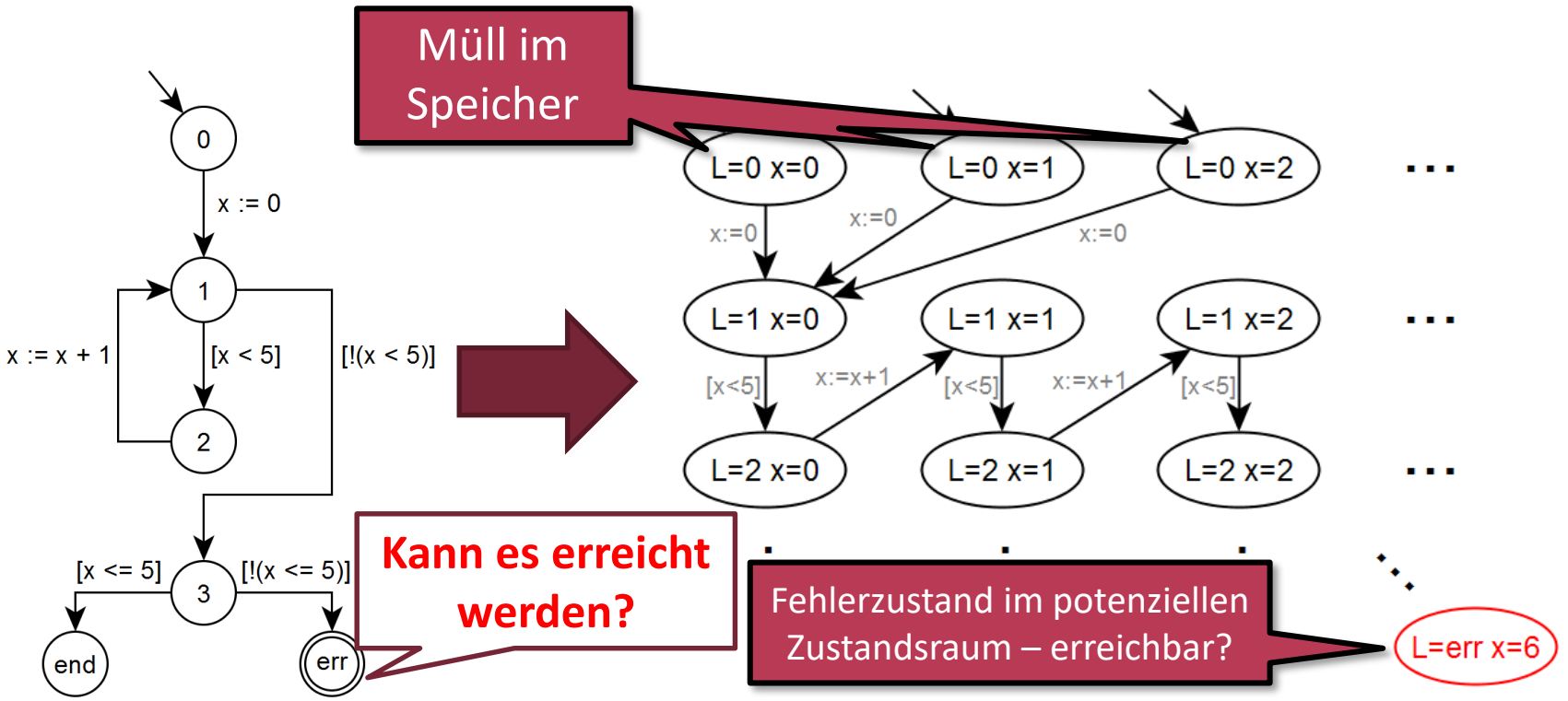
Software Verifikation – Zustandsraumgeneration

- **Zustände:** CFA-Knoten + Werte der Variablen (L, x_1, x_2, …, x_n)
- **Übergänge:** CFA-Kanten

CFA

Generierter Zustandsraum

(einfache Zustandsmaschine mit spontanen Übergängen)



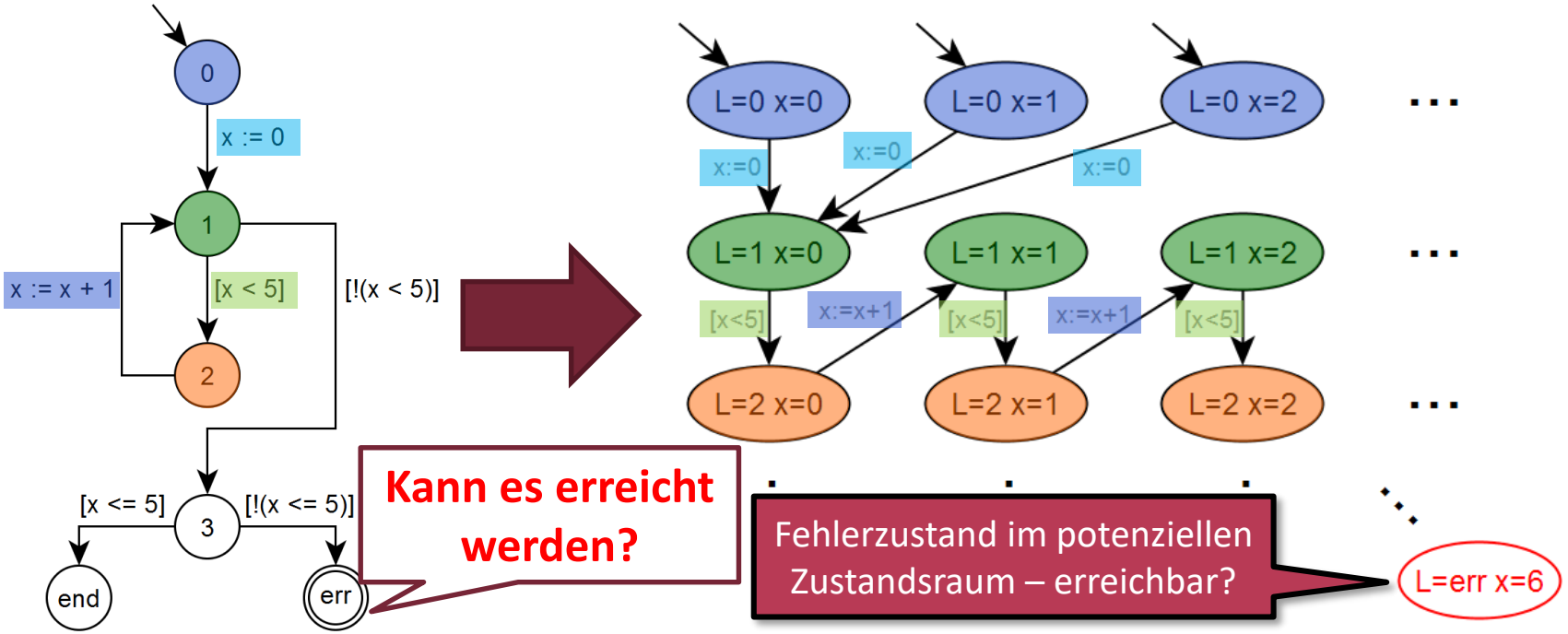
Software Verifikation – Zustandsraumgeneration

- **Zustände:** CFA-Knoten + Werte der Variablen (L, x_1, x_2, …, x_n)
- **Übergänge:** CFA-Kanten

CFA

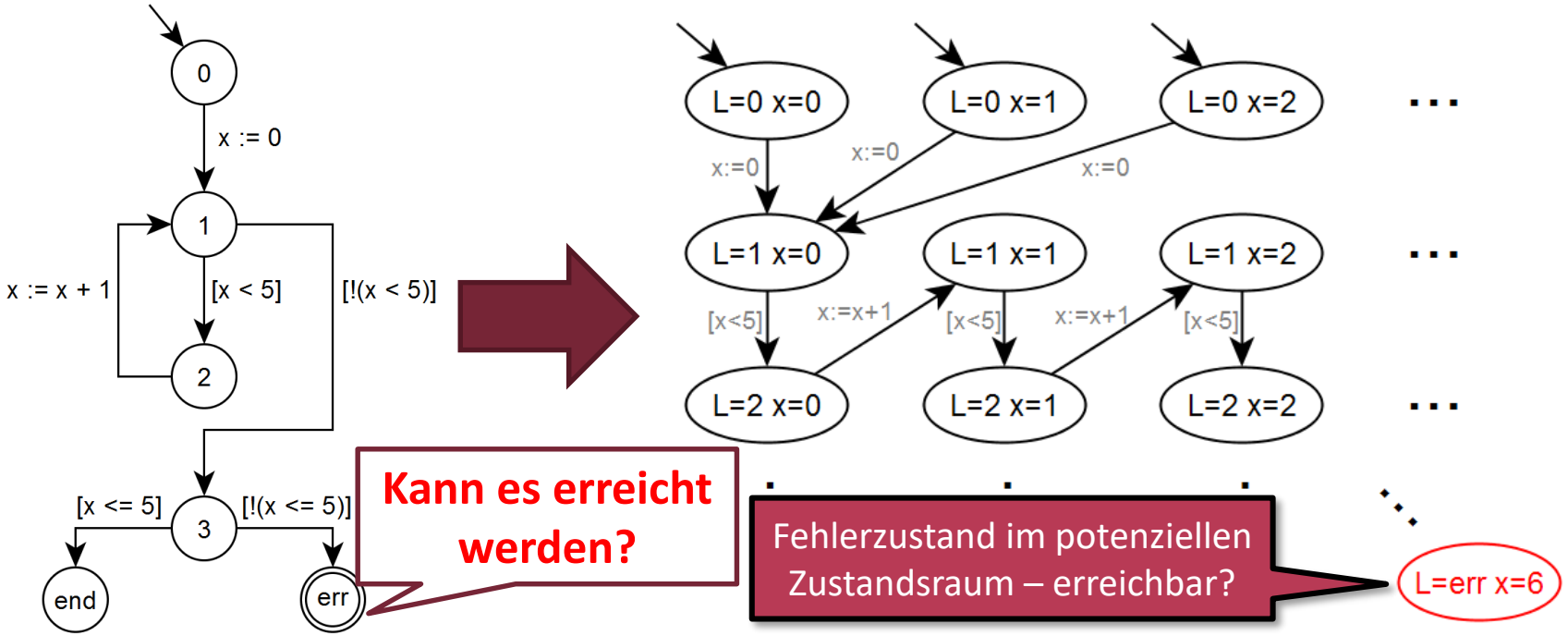
Generierter Zustandsraum

(einfache Zustandsmaschine mit spontanen Übergängen)



Software Verification – State Space Generation

- **Zustände:** CFA-Knoten + Werte der Variablen (L, x_1, x_2, …, x_n)
- **Übergänge:** CFA-Kanten
- Problem: **Zustandsraumexplosion** wegen der Variablen
  - z.B.: 10 Knoten, 2 Stück 32-bit Integers → $10 \cdot 2^{32} \cdot 2^{32}$ mögliche Zustände
- **Ziel: Verringerung** der Größe des Zustandsraumes **durch Abstraktion**



Software Verification – State Space Generation

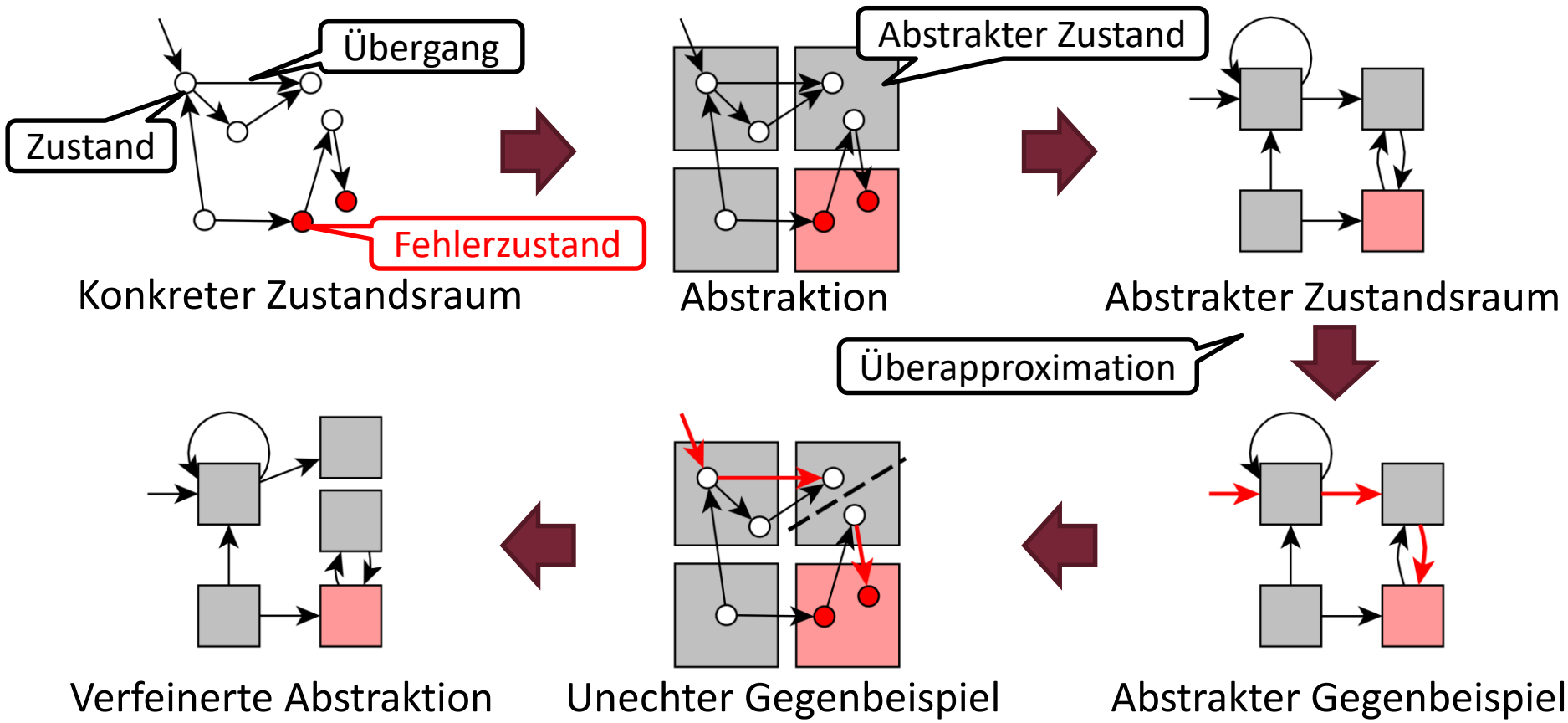
- **Zustände:** CFA-Knoten + Werte der Variablen ($L, x_1, x_2, \ldots, x_n$)
- **Übergänge:** CFA-Kanten
- Problem: **Zustandsraumexplosion** wegen der Variablen
  - z.B.: 10 Knoten, 2 Stück 32-bit Integers → $10 \cdot 2^{32} \cdot 2^{32}$ mögliche Zustände
- **Ziel: Verringerung** der Größe des Zustandsraumes **durch Abstraktion**

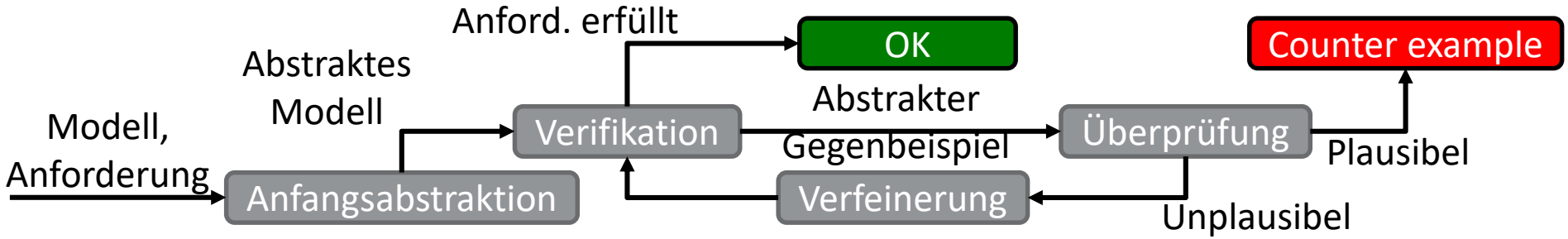
Eine allgemeine Lösung:

CounterExample-Guided Abstraction Refinement

CEGAR (Auf Gegenbeispielen basierende Abstraktionsverfeinerung)

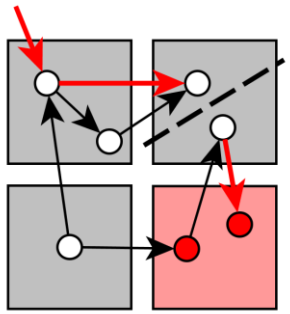
CEGAR – Übersicht





CEGAR – Verschiedene Arten der Abstraktion

- **Prädikatabstraktion**
  - **Anfangs:** nur die Knoten (keine Variablen)
  - **Verfeinerung:** in den zwei Teilen des Klemm-zustandes müssen die Werte der neuen Prädikaten verschieden sein
  - Anwendbar für größere Wertemengen



- **Variablen ausblenden** (Projektion)
  - **Anfangs:** nur die Knoten (keine Variablen)
  - **Verfeinerung:** in den zwei Teilen des Klemmzustandes müssen die Werte der neulich sichtbaren Variablen verschieden sein
  - Anwendbar, wenn es mehrere (irrelevante) Variablen gibt
- **Hybride Abstraktion**
  - Manche Variablen werden eingeblendet
  - Manche andere werden durch Prädikate abgedeckt

CEGAR – Zusammenfassung

- **CounterExample-Guided Abstraction Refinement**
- Anfangs, suche Fehler **in einer groben Abstraktion**
  - Wenn es keine gibt, dann gibt es keinen Gegenbeispiel
- **Simuliere** den Fehler auf dem konkreten System
  - Wenn der auch da möglich ist, dann ist der Fehler ein echter Gegenbeispiel
- Für **unechte Fehler**, **verfeinere**
  - … so, dass dieser Fehler nie mehr auftaucht

Abstraktes Modell = Kleineres Modell = Effizienz++

SOFTWARE VERIFIKATION BEISPIEL

Modellverifikation eines Algorithmus
für gegenseitigen Ausschluss

Algorithmus für gegenseitigen Ausschluss (Pseudokode)

- 2 Teilnehmer, 3 gemeinsame Variablen (H. Hyman, 1966)
  - **blocked0**: 1. Teilnehmer (P0) will eintreten
  - **blocked1**: 2. Teilnehmer (P1) will eintreten
  - **turn**: Wer darf als nächster eintreten? (0 für P0, 1 für P1)

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Ist dieser Algorithmus korrekt?

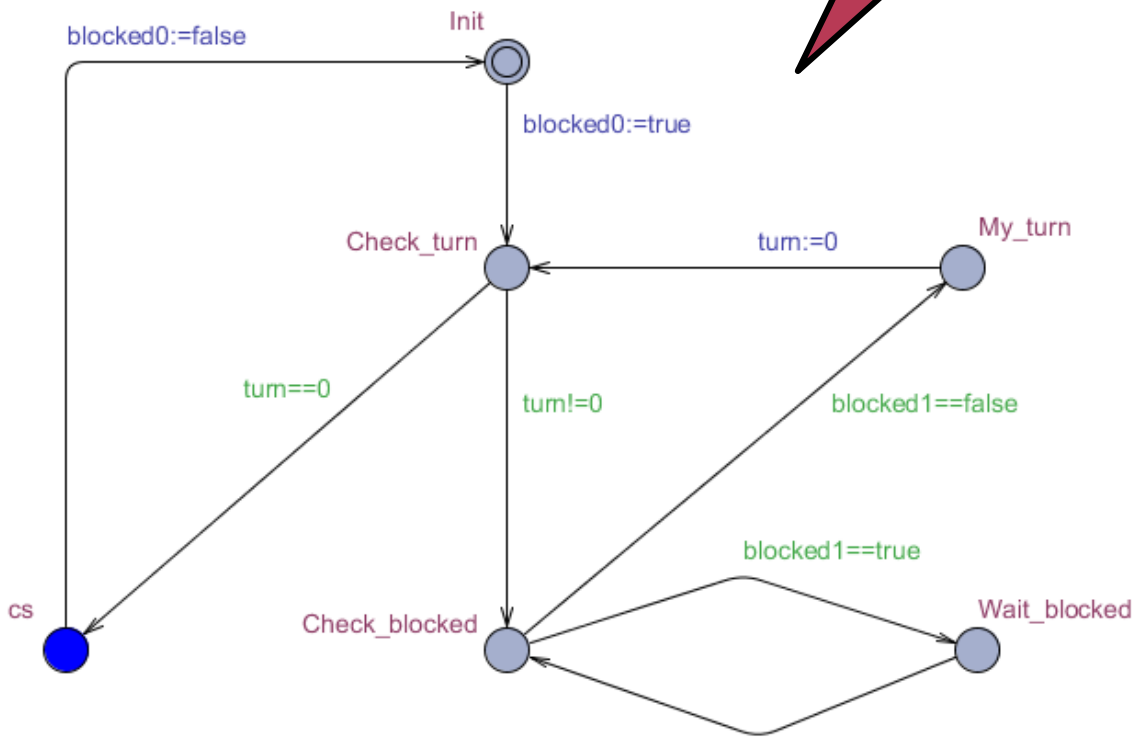
Modell des P0 Prozesses

Declarations:

```
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;
```

UPPAAL model checking tool

The P0 automat:



P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: false
- **turn**: 0

P0 und P1 beide starten.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: **true**
- **turn**: 0

P1 will eintreten, ...

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: true
- **turn**: 0

... aber dies ist nicht seine Runde.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: true
- **turn**: 0

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (0!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: true
- **turn**: 0

P1 sollte warten,
wenn auch P0 wollte eintreten, ...

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: false
- **blocked1**: true
- **turn**: 0

... P0 will aber noch nicht.
Deshalb P1 darf dies als seine Runde annehmen.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (false==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: **true**
- **blocked1**: true
- **turn**: 0

P0 will auch eintreten, ...

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (false==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 0

... und dies ist seine Runde.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (false==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 0

P0

```
while (true) {
   blocked0 = true;
   while (0!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (false==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 0

Deshalb P0 darf eintreten.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (false==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: **1**

P1 nimmt dies als seine Runde an, ...

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 1

... und dies wird dann auch seine Runde.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 1

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (1!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten

- **blocked0**: true
- **blocked1**: true
- **turn**: 1

Deshalb darf P1 auch eintreten.

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
    while (blocked1==true) {
        skip;
    }
    turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
    while (blocked0==true) {
        skip;
    }
    turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```

Fehlerhaftes Verhalten