# Entscheidungsbäume, Verhaltensbäume

# Agenten

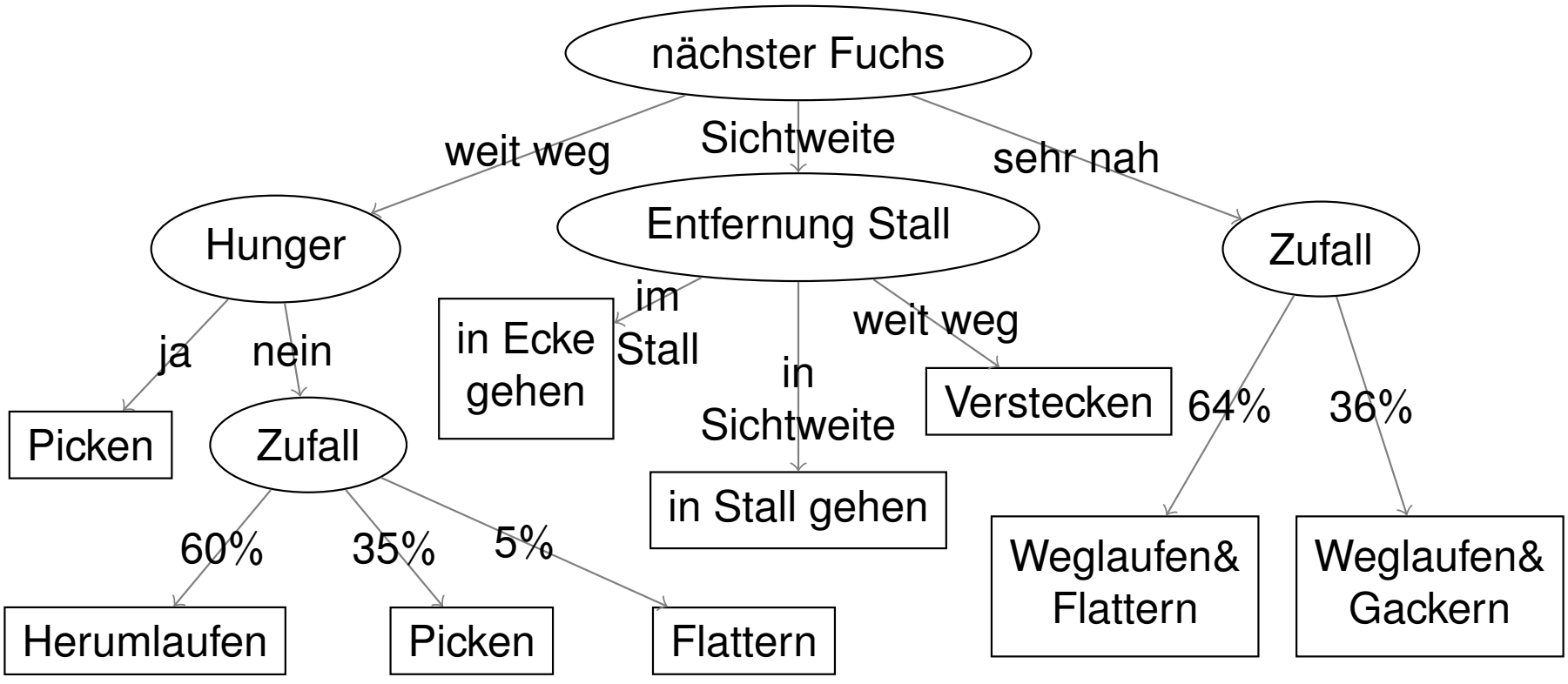
Computergesteuerte Akteure in Simulationen (und in der Robotik, sowie in Spielen) werden oft Agenten genannt (in Spielen verbreiteter MOBs).

Verschiedene Algorithmen in Verwendung, um Agenten zu steuern (oft Kombinationen)

- global planende Systeme (Beispiel: Wegsuche per Breitensuche aus dem Hauptprojekte)
- Entscheidungslisten (Beispiel: MOBs aus Vorprojekt)
- Zustandsübergangssysteme, Bedürfnishierarchie, Planungswarteschlange, . . . (In diesem Praktikum nicht verwendete Systeme)
- Entscheidungsbäume, Verhaltensbäume (Beispiel: Gesteuerte Flugzeuge aus Hauptprojekt)

# Entscheidungsbäume

Baum mit Enscheidungsfragen an inneren Knoten, unterschiedlichen Kindern für die Antworten und Verhaltensweisen an den Blättern



# Entscheidungsbäume, Umsetzung

Zwei Arten von Knoten:

- Entscheidungsknoten:
  Aus dem Zustand der Welt wird ein Kind gewählt
  Im Praktikum:
  - Nur zwei Kinder
  - Methode vom Typ `boolean name(Weltzustand w)` gibt an, ob linkes oder rechtes Kind
  - In JSON spezifiziert, welche Methode an diesem Knoten aufgerufen werden soll

- Aktionsknoten:
  Umzusetzende Aktion, reiner Seiteneffekt: Methode vom Typ `void name(Weltzustand w)`
  In JSON spezifiziert, welche Methode

# Entscheidungsbäume, Eigenschaften

Kleine Entscheidungsbäume können bereits komplexes Verhalten modellieren

Komplexere Graphen als Baum nicht notwendig: Aktionen können wieder Entscheidungsbäume sein

Nachteile: System rein reaktiv, innerer Zustand nicht direkt in der Modellierung (nur durch Aktionen, die inneren Zustand setzen machbar, schlecht für Übersichtlichkeit)

# Verhaltensbäume

Seit etwa 2000 bekannt, popularisiert durch Halo 2, Spore und Bioshock, sowie die Unreal Engine.
Verbreitet in Computerspiele- und Robotikindustrie
Erweiterung von Entscheidungsbäumen: Gibt Kombinatoren für Verhalten, sowie zeitliche Komponenten

- Es gibt **Aktionsknoten** und **Entscheidungsknoten** wie in Entscheidungsbäumen
- Jeder Aktionsknoten gibt zusätzlich an, ob die Aktion ein Erfolg oder ein Fehlschlag war
- Weitere Sorte innerer Knoten: **Dekoratoren**
  Hat genau ein Kind, bearbeitet das Ergebnis des Kinds
- Weitere Sorte innerer Knoten: **Kombinatorknoten**
  Hat Liste von Kinder, kombiniert ihre Verhaltensweisen in der Ausführung

Verhaltensbäume nicht vollständig einheitlich in Verwendung, gibt verschiedene Erweiterungen und Einschränkungen

# Verhaltensbäume, Verwendungsgründe

Alles, was man in Verhaltensbäumen schreiben kann, kann man auch in reinem Code schreiben; durch die angehängten Funktionsaufrufe können Verhaltensbäume genau so viel, wie normaler Code

Verhaltensbäume stellen sehr gut abgetrennte Einheiten dar, bessere Wartbarkeit

Erfahrung zeigt: Planungsverhalten, intelligent wirkendes Verhalten in Verhaltensbäumen leichter für Menschen umsetzbar

# Verhaltensbäume ausführen

Ergebnis eines Verhaltensbaums zu berechnen nennt man von Ausführen.

Ergebnis eines Verhaltensbaums oder Teilbaums ist eines von

- **„Erfolg“**
- **„Fehlschlag“**
- **„in Bearbeitung“**
  In diesem Fall werden zusätzliche Werte zurückgegeben, der sogenannte Ausführungszustand

Ist das Ergebnis „Erfolg“ oder „Fehlschlag“ ist, spricht man davon, dass dieser Teilbaum fertig ist
Ist das Ergebnis „in Bearbeitung“, so kann die Berechnung später fortgesetzt werden

# Verhaltensbäume, Aktionen

Aktionen haben Codereferenz, diese kann alle 3 Rückgabewerte („Erfolg“, „Fehlschlag“, „in Bearbeitung“) erzeugen, kann ihren internen Ausführungszustand fortsetzen

Gibt ein Kind „in Bearbeitung“ zurück, so passiert einheitlich das Folgende in fast allen Knoten (außer Parallel, spätere Folie)

- Knoten selbst gibt ebenfalls „in Bearbeitung“ zurück
- Ausführungszustand ist innerer Zustand des Knotens, sowie Information, welches Kind gerade ausgeführt wird, sowie Ausführungszustand des Kindes in Ausführung

# Zustandsnotation und Beispiele

Pseudonotation dieser Foliensatz (keine echte Java-Syntax)

- `'foo'` steht für einen Hook, also einen Funktionsaufruf, dessen Wert vom Weltzustand abhängt und dessen Ausführung sich auf den Weltzustand auswirken kann
- `Erfolg()`, `Fehlschlag()` für das entsprechende Berechnungsergebnis oder einen Knoten, der sofort dieses Ergebnis liefern würde
- `Bearb...(innerer Zustand)` in Bearbeitung
- andere Objektnamen für andere nicht gestartete Knoten

Damit steht beispielsweise

- `BearbDelay(3,Erfolg())` für einen Zustand, der noch 3 Sekunden wartet, bis er Erfolg zurückgibt
- `BearbDelay(3,`
  `Decide('foo',Fehlschlag(),Erfolg()))` für einen Zustand, der 3 Sekunden wartet und wenn `'foo'` dann wird „Fehlschlag“ zurückgegeben, ansonsten „Erfolg“

# Verhaltensbäume, Dekoratoren

Jeder Dekorator hat genau ein Kind

- Inverter: Kind wird ausgeführt, Ergebnis invertiert: Bei Erfolg Fehlschlag zurückgeben, bei Fehlschlag Erfolg
- Warteknoten: Ausführung wird an der Stelle für eine feste Zeitdauer angehalten, danach wird das Kind ausgeführt
  Um das zu tun, gibt der Warteknoten „in Bearbeitung“ zurück; wird er nach der Wartezeit fortgesetzt, wird das Kind ausgeführt
- Feste Schleife: Kinderknoten wird mehrmals ausgeführt, festgelegte Anzahl
- While-Schleife: Kinderknoten wird solange wiederholt, bis er Fehlschlag zurückgibt; gibt Fehlschlag zurück, wenn Kind nie erfolgreich war
- Retry-Schleife: Kinderknoten wird solange wiederholt, bis er Erfolg zurückgibt; dann Erfolg zurückgeben
- Debug: Kind wird normal ausgeführt, aber es wird eine Debugnachricht ausgegeben, wenn die Bearbeitung anfängt und wenn sie zuende ist

# Verhaltensbäume, Dekoratoren, Details

- Inverter: Kind gibt zurück
  - Erfolg: Gib Fehlschlag zurück
  - Fehlschlag: Gib Erfolg zurück
  - Bearbeitungszustand $b$: Gib invert($b$) zurück
    Bekommt Inverter den Zustand invert($b$), so wird dem Kind $b$ gegeben

- Feste Schleife: Benötigt interne Schleifenvariable $i$, beim ersten Aufruf $i \leftarrow 0$
  Kind gibt zurück
  - Erfolg: $i$ um 1 erhöhen, falls $i \geq$Durchlaufzahl: Gib Erfolg zurück, sonst Kind erneut aufrufen
  - Fehlschlag: $i$ um 1 erhöhen, falls $i \geq$Durchlaufzahl: Gib Fehlschlag zurück, sonst Kind erneut aufrufen
  - Bearbeitungszustand $b$: Gib feste_Schleife($i$,$b$) zurück

Andere Dekoratoren analog

# Verhaltensbäume, Kombinatoren

Jeder Kombinator hat eine Liste an Kindern; kann fest oder randomisiert sein

- Sequenz:
  Alle Kinder werden der Reihe nach ausgeführt, es sei denn, ein Kind gibt Fehlschlag zurück, dann Fehlschlag zurückgeben.
  Waren alle Kinder erfolgreich, dann Erfolg zurückgeben
- Selektor (auch Fallback genannt):
  Wie Sequenz, nur Rolle von Erfolg und Fehlschlag vertauscht.
- Parallel:
  Alle Kinder werden „gleichzeitig“ ausgeführt, d.h. die Kinder werden in Reihenfolge ausgeführt, gibt aber ein Kind „in Bearbeitung“ zurück, so werden die anderen Kinder auch ausgeführt.
  Sobald ein Kind „Erfolg“ oder „Fehlschlag“ zurückgibt, wird das vom Parallel-Knoten auch zurückgegeben

# Verhaltensbäume, Sequenz und Selektor

*Sequenz:* Führt Kinder nacheinander aus, solange sie erfolgreich waren

*Selektor:* Führt Kinder nacheinander aus, bis eines erfolgreich war

Im Falle von „in Bearbeitung“ wird gespeichert, welches Kind „in Bearbeitung“ ist und welchen Zustand dieses Kind hat

Beispiel:

- Verhaltensbaum `Sequenz('foo','bar','buz','blah')`
- `'foo'` gibt „in Bearbeitung“ zurück, damit Zustand `BearbSequenz(0,BearbFoo(...),'foo','bar','buz','blah')`
- Bei nächster Berechnung gibt `'foo'` „Erfolg“ zurück
- Darum wird direkt `'bar'` gestartet, gibt auch „Erfolg“ zurück
- Darum wird direkt `'buz'` gestartet, gibt „in Bearbeitung“ zurück
  Damit ensteht Zustand `BearbSequenz(3,BearbBuz(...),'foo','bar','buz','blah')`
- Bei nächster Berechnung gibt `'buz'` „Fehlschlag“ zurück, darum gibt Sequenz im ganzen auch „Fehlschlag“ zurück

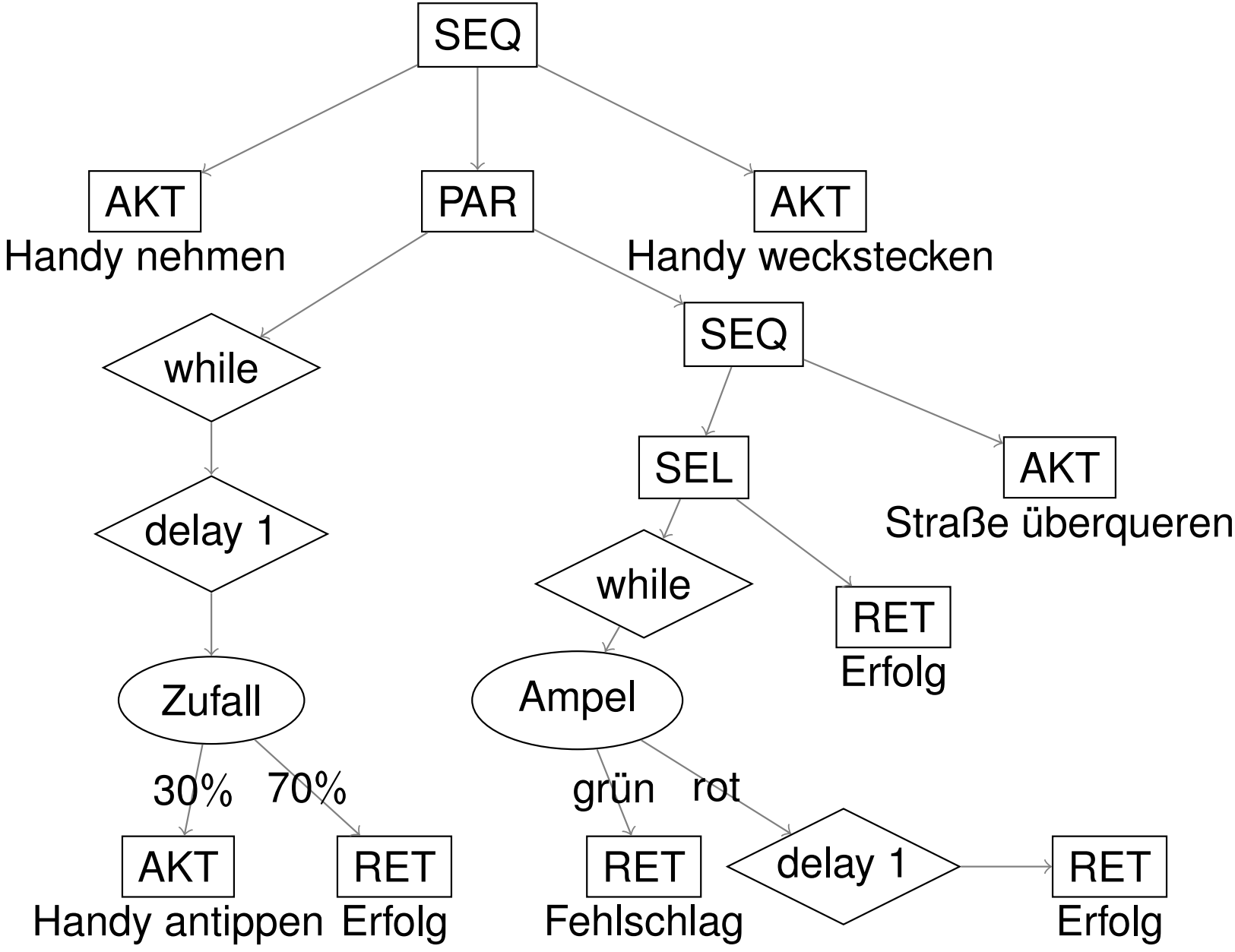
# Verhaltensbäume, Parallelkombinator

Nur wenn alle Kinder „in Bearbeitung“ zurückgeben, gibt Parallel-Kombinator auch „in Bearbeitung“ zurück
Rückgabe: parallel($b_0, b_1, b_1, \ldots$)

Parallelkombinatoren mit Bedacht einsetzen: Können schnell unübersichtlich werden

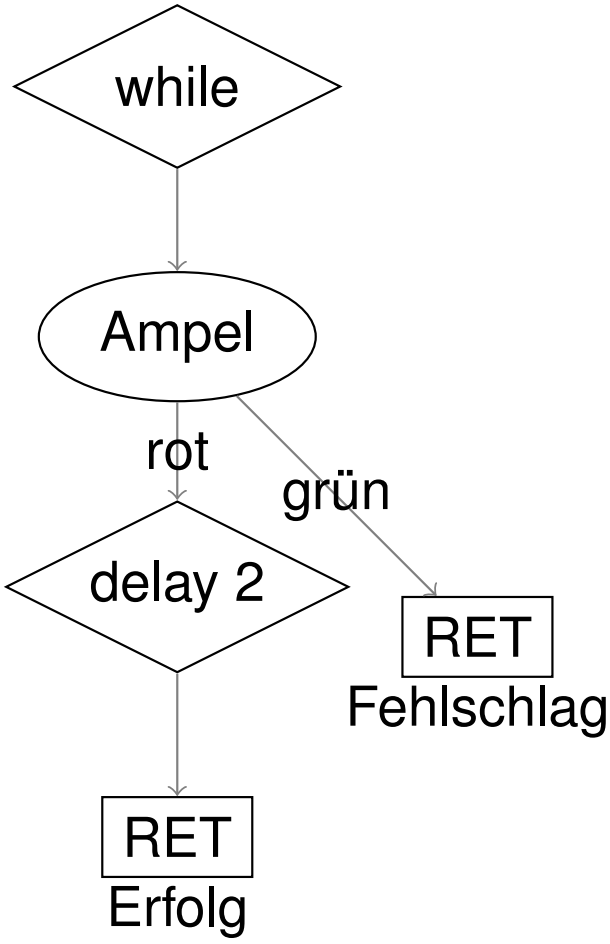
Parallelkombinatoren können Sicherheit geben: Paralleler Check, ob anderes Objekt zu nahe kommt, damit Überleitung in anderes Verhalten

# Beispiel Verhaltensbaum: Ampelüberquerung



# Beispiel Ausführung Verhaltensbaum

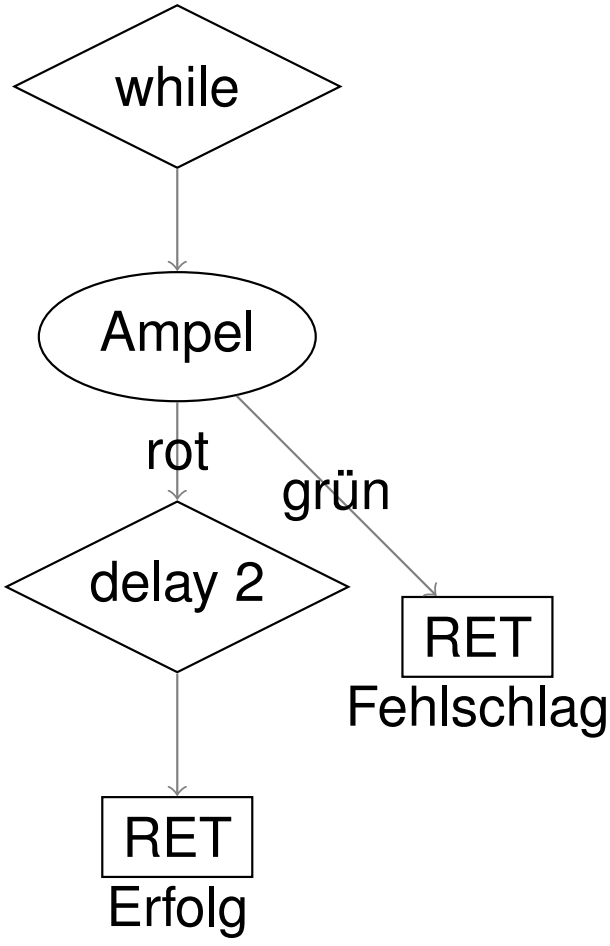
Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343



# Beispiel Ausführung Verhaltensbaum

Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343

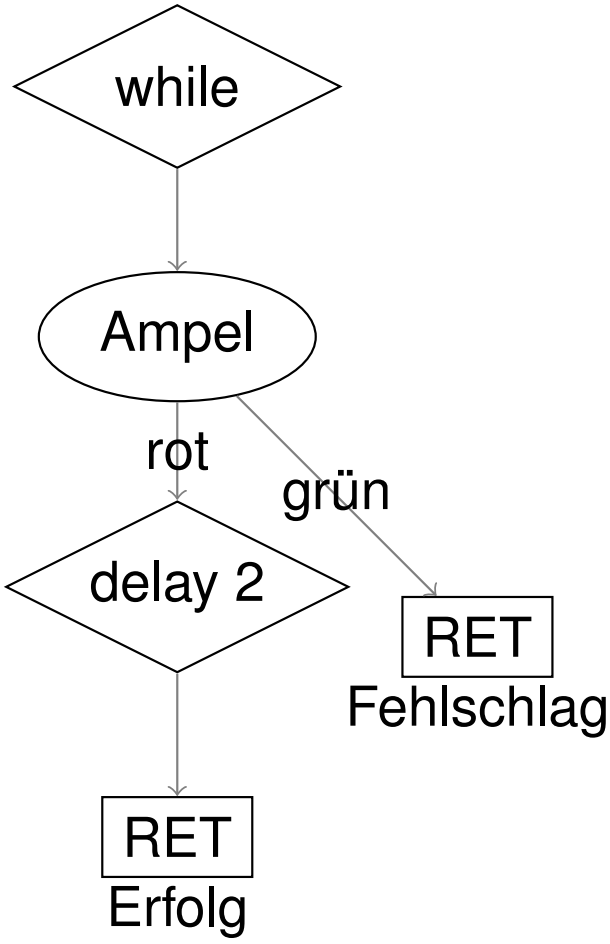
t=343 „while“



# Beispiel Ausführung Verhaltensbaum

Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343

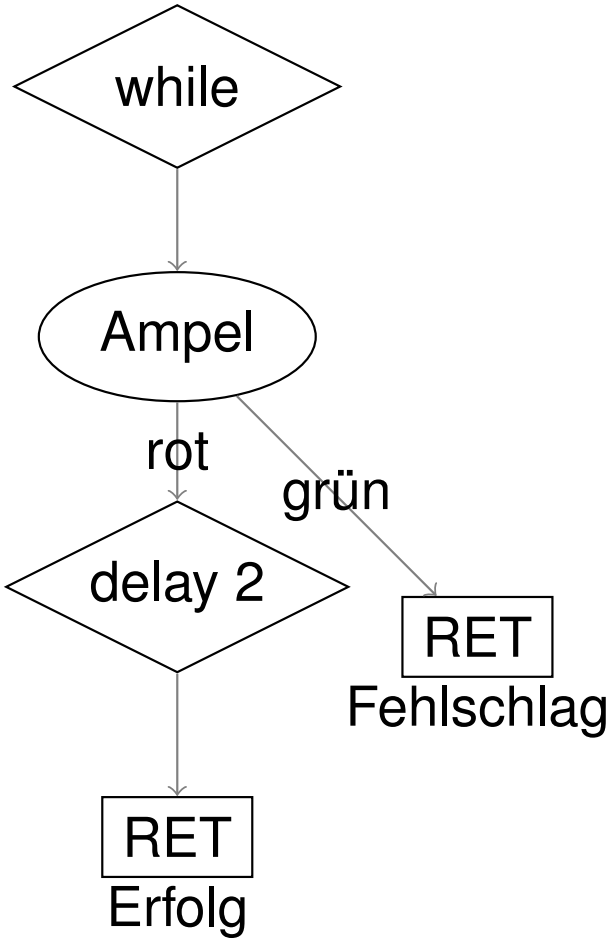
t=343 „while“ ruft Kind „Ampel“ auf



# Beispiel Ausführung Verhaltensbaum

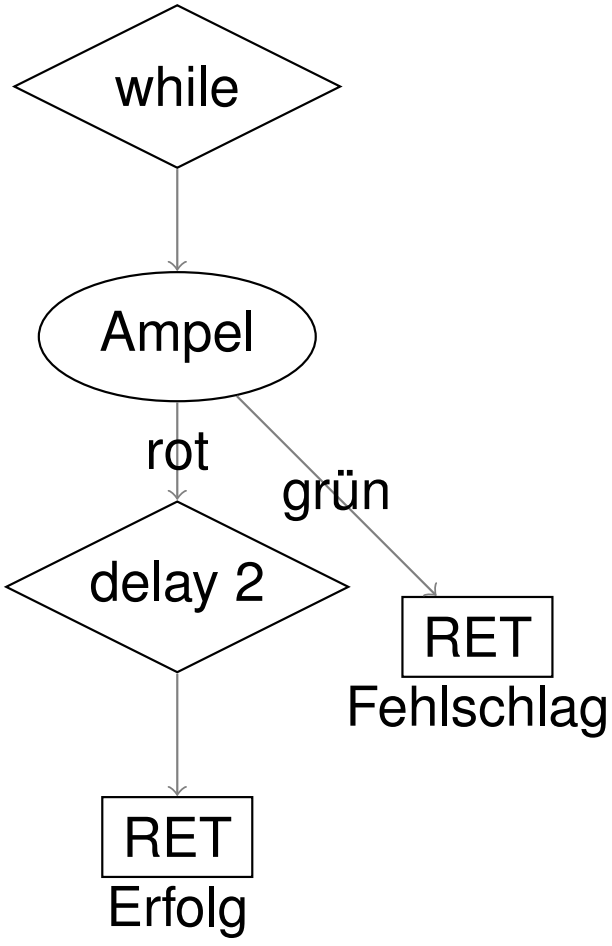
## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343

t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf



# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343

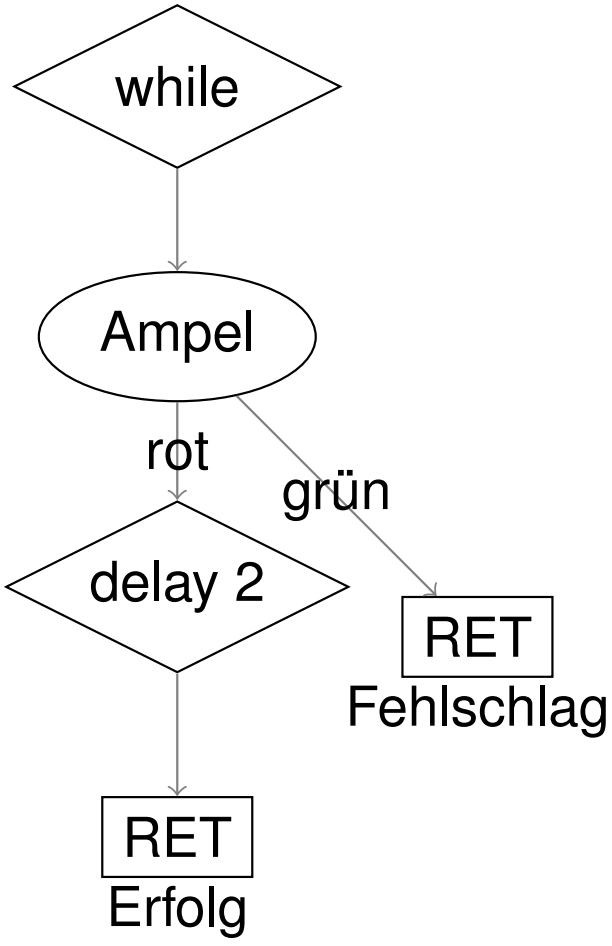



t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343



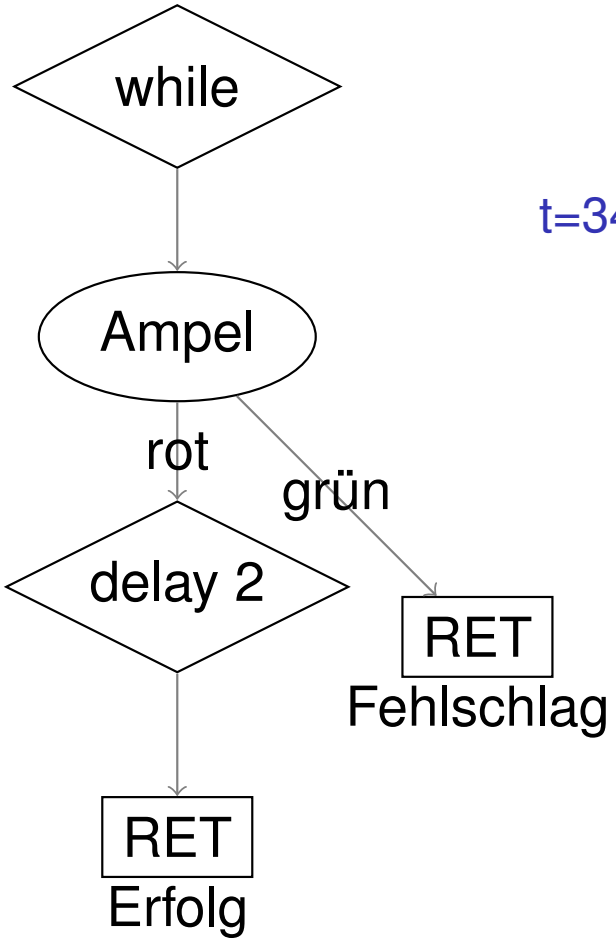

t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

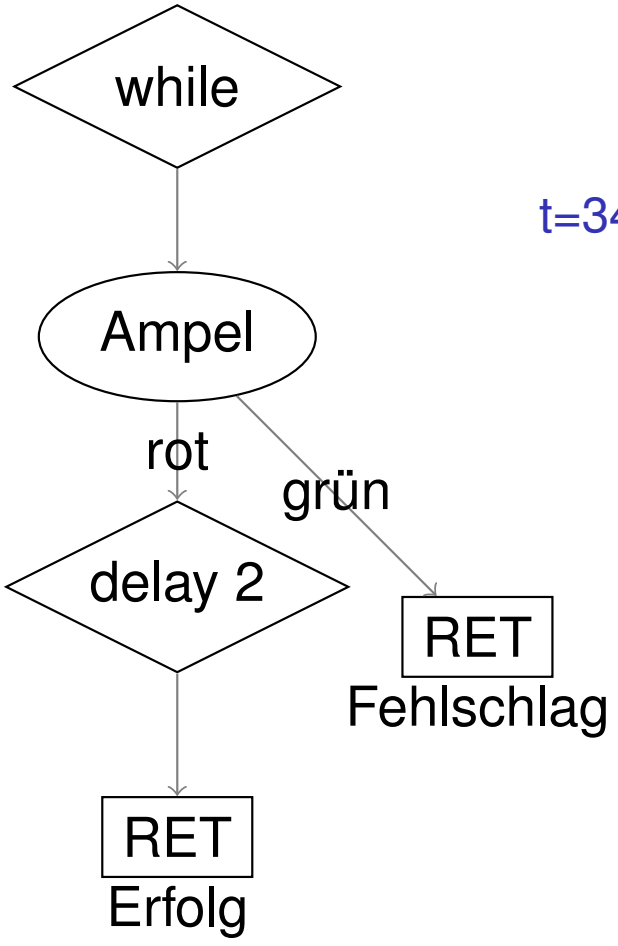
„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

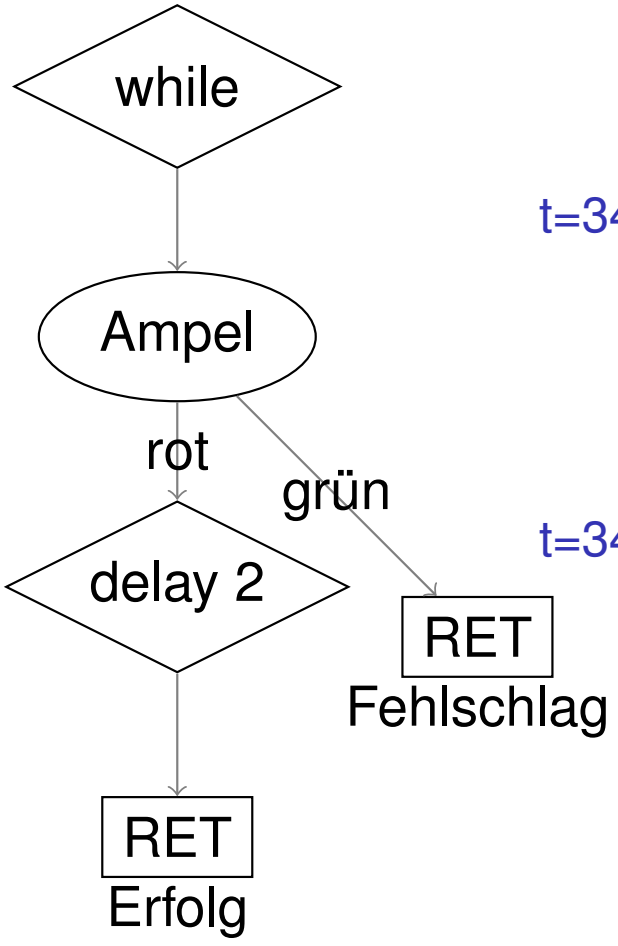
„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

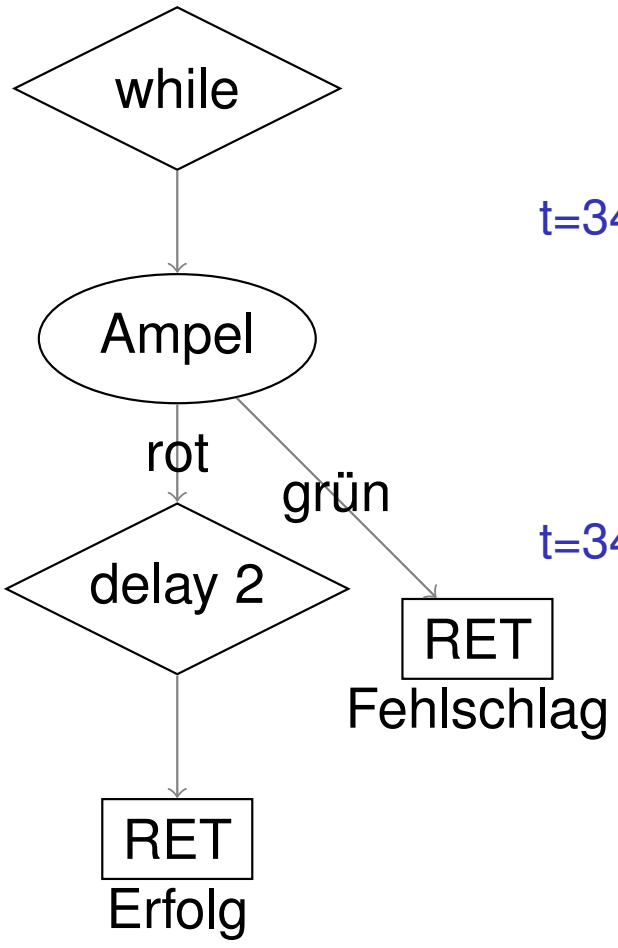
„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

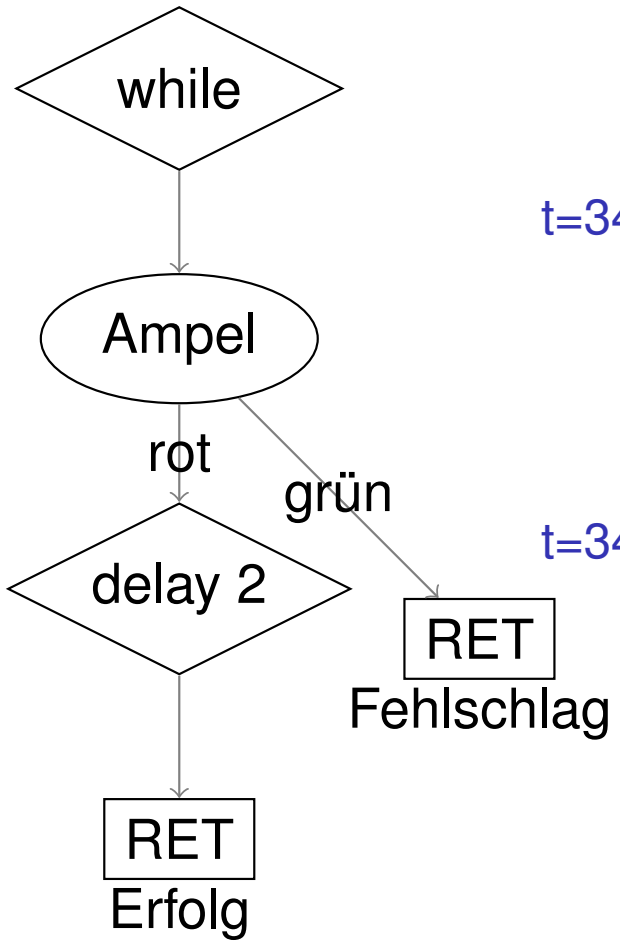
„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 = 345$, also Rückgabe `Erfolg()`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

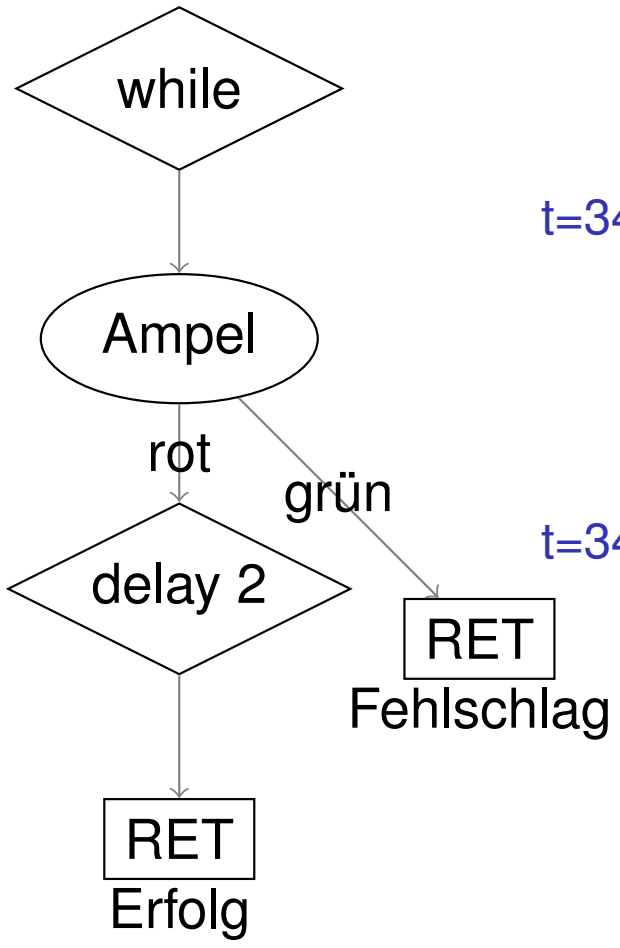
t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 = 345$, also Rückgabe `Erfolg()`

Damit gibt „Ampel“ ebenfalls `Erfolg()` zurück

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

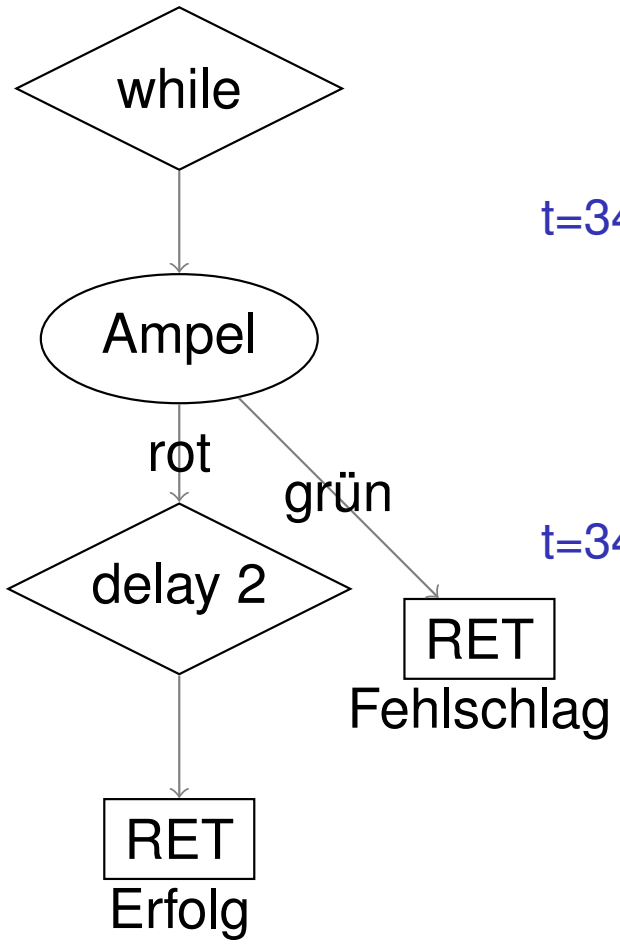
t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 = 345$, also Rückgabe `Erfolg()`

Damit gibt „Ampel“ ebenfalls `Erfolg()` zurück

„while“ bekommt `Erfolg()`, Aufruf „Ampel“

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 = 345$, also Rückgabe `Erfolg()`

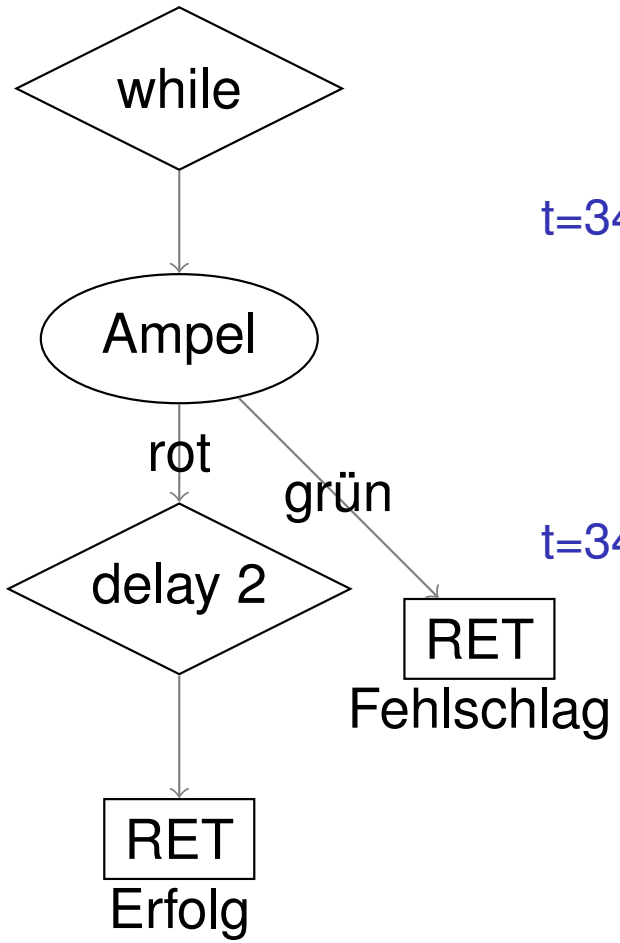
Damit gibt „Ampel“ ebenfalls `Erfolg()` zurück

„while“ bekommt `Erfolg()`, Aufruf „Ampel“

„Ampel“ stellt fest: Situation grün, ruft also `Fehlschlag()` auf mit Rückgabe `Fehlschlag`

# Beispiel Ausführung Verhaltensbaum

## Beispielaufruf gekürzter Verhaltensbaum, Startzeitpunkt: 343




t=343 „while“ ruft Kind „Ampel“ auf stellt fest: Situation rot, ruft also Kind „delay 2“ auf

„delay 2“ gibt zurück `BearbDelay(345,Erfolg())`

„Ampel“ gibt zurück `Ampel(BearbDelay(...))`

t=344 Rekursive Aufrufe auf den Kindern/Bearbeitungszuständen bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 > 344$, also an jeder Stelle gleicher Rückgabewert wie bei Zeitpunkt 343

t=345 Rekursive Aufrufe bishin zu „delay“ mit `BearbDelay(345,Erfolg())` „delay“ stellt fest $345 = 345$, also Rückgabe `Erfolg()`

Damit gibt „Ampel“ ebenfalls `Erfolg()` zurück

„while“ bekommt `Erfolg()`, Aufruf „Ampel“

„Ampel“ stellt fest: Situation grün, ruft also `Fehlschlag()` auf mit Rückgabe `Fehlschlag`

„while“ bekommt `Fehlschlag`, hatte aber 1 erfolgreichen Durchlauf, also Rückgabe `Erfolg()`

# Wegfindung

Flugzeuge müssen von der Simulation intelligent zu Ihrem nächsten Wegpunkt geführt werden, ohne dass dabei Unfälle passieren.

Intelligent erscheinende Wegführung der Flugzeuge wird durch Verhaltensbäume implementiert $\Rightarrow$ eigenes Vorlesungskapitel

Simulation speicher Position und Geschwindigkeit von allen Flugzeugen.
Zusätzlich dazu jedes Flugzeug einen **inneren Zustand** welcher nur für den Verhaltensbaum relevant ist:

- Zielgeschwindigkeit
- Wegplan: Liste von Konten-Bezeichner
- Alternativer Wegplan: Liste von Knoten-Bezeichner

Modelliert durch Klasse `Agent`

# Optionale Vorlagen als Hilfestellung

```
public enum StateKind { SUCCESS, FAILURE, RUNNING}
```

---

```
public interface BehaveNode {
  /** Neuen Zustand eines Verhaltensbaums starten.
   *  @param Aktueller Zeitpunkt in Sekunden
   *  @return Der erzeugte Verhaltensbaumzustand  */
  public BehaveState init(int now, Agent agent);
}
```

---

```
public interface BehaveState {
  /** Was ist der aktuelle Ausfuehrungszustand?
   *  @return Art des Ausfuehrungszustands                    */
  public StateKind kind();
  /** Verhaltensbaum fortsetzen bis zum Zeitpunkt.
   *  @param Neuer Zeitpunkt in Sekunden nach Mitternacht
   *  @return Neuer Ausfuehrungszustand. Kann this sein,
   *     kann aber auch ein neues Objekt sein, vielleicht
   *     sogar aus einer anderen Klasse sein                  */
  public BehaveState tick(int now);
}
```