Zustandsautomaten (Stichworte)

Udo Kelter

11.01.2013

Zusammenfassung dieses Lehrmoduls

Zustandsautomaten sind erweiterte Formen von endlichen Automaten bzw. einfachen Zustandsübergangsdiagrammen. Die UML 2.0 definiert zwei Arten von Zustandsautomaten: Verhaltenszustandsautomaten und Protokollzustandsautomaten. Beide basieren auf den gleichen Grundlagen, weisen aber diverse Detailunterschiede auf. Dieses Lehrmodul stellt vor allem die Besonderheiten von Verhaltenszustandsautomaten vor: spezielle Arten von Triggern, Pseudozustände und Transitionspfade sowie zusammengesetzte Zustände.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Zustandsübergangsdiagramme
- Petri-Netze

Stoffumfang in Vorlesungsdoppelstunden: 1.0

Inhaltsverzeichnis

1 Motivation und Einordnung **3**
1.1 Steuerungsmodelle in der UML 2.* 3
1.2 Herkunft der Konzepte von Zustandsautomaten 4
1.3 Verhaltens- vs. Protokollzustandsautomaten 4

2 Zustände **5**

3 Transitionen **6**
3.1 Allgemeine Form einer Transition 7
3.2 Arten von Triggern 8
3.3 Guards 9

4 Pseudozustände und Transitionspfade **11**
4.1 Pseudozustände und Transitionspfade 11
4.2 Pseudozustand Kreuzung (*junction*) 12
4.3 Pseudozustand Auswahlknoten (*choice pseudo state*) 13
4.4 Pseudozustand Startzustand (*initial pseudostate*) 13
4.5 Pseudozustand Terminator (*terminate pseudostate*) 13
4.6 Pseudozustand Gabelung (*fork*) 14
4.7 Pseudozustand Vereinigung (*join*) 14

5 Zusammengesetzte Zustände **14**
5.1 Zusammengesetzte Zustände mit nur einer Region 16
5.2 Historien 17
5.2.1 Pseudozustand flache Historie (*shallow history*) 18
5.2.2 Pseudozustand tiefe Historie (*deep history*) 19
5.3 Regionen 19

Literatur 20
Index 20

1 Motivation und Einordnung

Die *Grundformen* von ZÜD (endliche Automaten) und Petri-Netzen sind

- Kernkonzepte zur Beschreibung von sequentiellen und/oder parallelen Steuerungen / Algorithmen
- mit einer sehr klaren Semantik + theoretischem Fundament

... haben aber Nachteile hinsichtlich der Modellierung realer Systeme:

- keine Berücksichtigung von Daten und datenabhängigen Steuerungen
- keine Abstraktionshierarchien, durch die komplexere Modelle strukturiert und besser verstehbar / entwickelbar werden

daher: diverse Erweiterungen
→ bessere Modellierungsfähigkeiten
→ Verlust der klaren semantischen Grundlage, viele interessierende Eigenschaften nicht mehr entscheidbar

1.1 Steuerungsmodelle in der UML 2.*

(nach vielen Irrungen und Umwegen über die UML 1.* ...)

1. **Zustandsautomaten** (*state machines*): endlichen Automaten (Mealy- und Moore-A.) + weitere Zutaten
   - Modell = gerichteter Graph
   - **Knoten** modellieren **Zustände**;
     können "Tokens" beinhalten / puffern;
     in beschränktem Umfang parallele Teilzustände möglich
   - **Kanten** modellieren **Zustandsübergänge**
     hierbei komplizierte Fallunterscheidungen möglich
     Kante ist i.d.R. mit einem Ereignis beschriftet
2. **Aktivitätsdiagramme**: ∼Programmablaufplan + Petri-Netz

- Modell = gerichteter Graph,
- **Knoten**: modellieren **Verarbeitungsschritte** / Funktionen, aktive Systemteile
- **Kanten**: modellieren **Kontroll- und Datenflüsse**, transportieren Daten- oder Kontrolltoken zwischen Aktionen, beinhalten Ablaufsteuerung

viele gemeinsame Diagrammelemente in beiden Diagrammtypen, insb. bei der Ablaufsteuerung
aber teilweise verschiedene Bedeutung, Verwechslungsgefahr!

1.2 Herkunft der Konzepte von Zustandsautomaten

Zustandsautomaten beinhalten diverse schon erlernte bzw. andiskutierte Konzepte (werden hier nur kurz wiederholt)

von **Zustandsübergangsdiagrammen**:

- Zustände mit internen Aktionen
- Zustandsübergänge mit Ereignissen, bedingten Transitionen und Aktionen

von **Petri-Netzen**:

- Plätze als Tokenpuffer
- Transitionen mit mehreren Eingangs- oder Ausgangsplätzen

1.3 Verhaltens- vs. Protokollzustandsautomaten

Verhaltenszustandsautomat (*behavioral state machine*)

- mit Aktionen / Verhalten an Transitionen
  (s. Lehrmodul ZÜD)

Protokollzustandsautomat (*protocol state machine*):

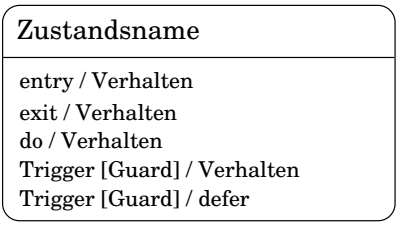
- ohne Aktionen / Verhalten an Transitionen
- Konzentration auf Zustandsübergänge, Darstellung der Vor- und Nachbedingungen für Zustandsübergänge

- Darstellung von Invarianten in Zuständen

beide Arten mit gleichen Grundlagen, aber diversen Detailunterschieden; Protokollzustandsautomaten werden hier nicht detailliert behandelt

2 Zustände

Allgemeine Form eines Zustands:



Ausführungsmodell für Zustandsübergänge:

A. Verhalten bei "Betreten" (Aktivierung) eines Zustands:

1. Zustand wird **aktiv**, wenn eine hereinkommende Transition durchlaufen wird
2. sofort nach der Aktivierung wird das Eintrittsverhalten (*entry / ...*) ausgeführt
   wird *nie abgebrochen*; währenddessen ankommende Ereignisse werden in einer FIFO-Schlange gepuffert
3. sofort danach wird das Zustandsverhalten (*do / ...*) gestartet

B. Verhalten bei Verlassen eines Zustands:

1. Zustand wird verlassen, wenn ein Ereignis eintritt, das zum Durchlaufen einer herausgehenden Transition führt

2. falls das Eintrittsverhalten des aktuellen Zustands noch nicht abgearbeitet ist, wird dies erst komplett abgearbeitet
3. falls das Zustandsverhalten abläuft, wird es abgebrochen
4. danach wird das Austrittsverhalten (*exit / ...*) ausgeführt
5. erst danach ist der Zustand **inaktiv**.

C. Interne Transitionen in einem Zustand
(*Trigger [Guard] / ...*):

- Wirkung analog zum allgemeinen Fall einer Transition, die einen Zustandswechsel bewirkt,
  aber *kein Durchlaufen des Ein- und Austrittsverhaltens*
- Aktion *defer* führt in bestimmten Situationen zu einer späteren Verarbeitung des Triggers (Details später)

3 Transitionen

Denkweise ist stark von kommunizierenden Objekten (um nicht zu sagen GUI-Programmierung...) beeinflußt:

- Ereignisse (Trigger) sind i.d.R. *Operationsaufrufe* gemäß der Schnittstelle des Typs des Objekts
- Aufrufe können von anderen Objekten kommen, aber auch vom gleichen Objekt
- Verhalten an Transitionen fehlt häufig, weil meist das Verhalten im Zielzustand abläuft
  (also Zuordnung der Aktionen wie bei Moore-Automaten)
  Vorteil: man kann über verschiedene Transitionen das gleiche Verhalten auslösen

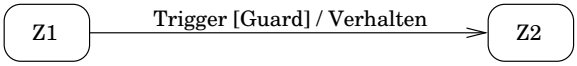
Ereignisverarbeitung in der UML 2.*:

- sehr komplexes Verarbeitungsmodell

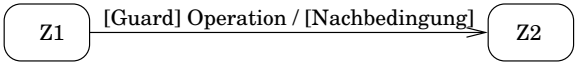
- diverse Details offen (*semantic variation points*), um verschiedene Applikationsbereiche mit gegensätzlichen Anforderungen bedienen zu können – problematisch

3.1 Allgemeine Form einer Transition

allgemeine Form einer Transition bei einem *Verhaltenszustandsautomaten*:



allgemeine Form einer Transition bei einem *Protokollzustandsautomaten* (*ProtocolTransition*):



Besonderheiten von Protokollzustandsautomaten:

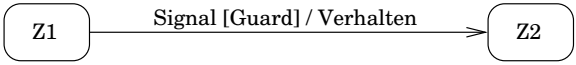
- nur Operationsaufrufe als Trigger
- Notationsreihenfolge Guard – Trigger/Operation vertauscht
- keine Aktionen (Implementierung von Verhalten), sondern nur Nachbedingungen (Wirkung des Verhaltens)
- Guard in beiden Fällen gleich

kompaktere Notation bei mehreren Transitionen mit gleichem Ausgangszustand, Guard, Verhalten und Zielzustand:
- nur ein Pfeil,
- unterschiedliche Trigger mit Kommata getrennt angeben

3.2 Arten von Triggern

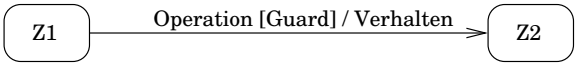
hier für Verhaltens-ZA, analog für Protokoll-ZA

1. *SignalTrigger:* Empfang eines externen Ereignisses; Darstellung:



   Situation:
   - Objekt / System ist in Zustand Z1,
   - empfängt externes Ereignis,
   - im Guard genannte Bedingungen sind erfüllt;
   → Objekt / System geht in Zustand Z2 über.
   gut geeignet für kommunizierende Objekte in verteilten Systemen geeignet (asynchrone Aufrufe)

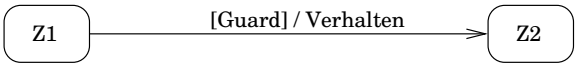
2. *CallTrigger:* Aufruf einer Operation des Objekts; Darstellung:



   Situation:
   - Objekt / System ist in Zustand Z1,
   - empfängt Aufruf der genannten Operation
   - im Guard genannte Bedingungen sind erfüllt;
   → Objekt / System geht in Zustand Z2 über.

   Schreibweise z.T. in der Form *operation()*
   Parameter der Operation können zusätzlich angegeben werden und im Guard und in den Aktionen benutzt werden.

3. *ChangeTrigger:* Zustandsänderung; Darstellung:

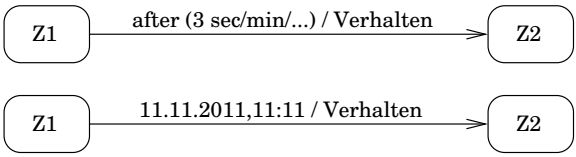


   Situation:

- Objekt / System ist in Zustand Z1,
- eine der im Guard genannten Variablen ändert ihren Wert,
- Guard evaluiert anschließend zu WAHR,
→ Objekt / System geht in Zustand Z2 über.

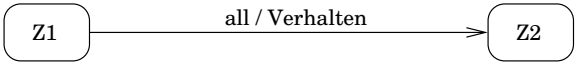
Kein expliziter Trigger, Trigger wird sozusagen implizit durch Wertänderungen generiert

4. *TimeTrigger:* absolute oder relative Zeitangaben erlaubt



Trigger wird implizit durch Zeitablauf nach Betreten des Zustands Z1 bzw. Erreichen des absoluten Zeitpunkts generiert

5. *AnyTrigger:* für nicht explizit genannte, restliche Trigger



Zitat UML Superstructure, 2006, Abschnitt 13.3.1:

> "A transition trigger associated with AnyReceiveEvent specifies that the transition is to be triggered by the receipt of any message that is not explicitly referenced in another transition from the same vertex."
>
> "Any **Any**ReceiveEvent is denoted by the string '**all**' used as the trigger."

Kommentar überflüssig. Woanders heißt das `else` oder `otherwise` und ist klar verständlich

3.3 Guards

Guards können beliebige Bedingungen enthalten, müssen zu jedem Zeitpunkt eindeutig auswertbar sein

Mehrere Guards für den gleichen Trigger:
zu einem Zustand Z1 können zum *gleichen Trigger mehrere Transitionen* mit i.a. unterschiedlichen Guards vorhanden sein

Frage 1: dürfen 2 oder mehr Guards den gleichen Fall abdecken?
Antwort: ja, Auswahl der Transition dann zufällig

Frage 2: müssen die Guards alle Fälle abdecken?
Antwort: nicht unbedingt → s. folgende Folie

Interpretation nicht abgedeckter Fälle:

1. **irrelevant**: eingetroffenes Ereignis bewirkt keine Zustandsänderung und wird ignoriert
   entspricht der Notationskonvention in ZÜD!

2. **schwerer Fehler**:
   eingetroffenes Ereignis hätte eigentlich gar nicht kommen dürfen und kann nicht sinnvoll verarbeitet werden;
   System ist in einem inkonsistenten Zustand und wird beendet / Objekt wird aufgelöst ("Panik") oder Übergang zu einer Standard-Fehlerbehandlung
   → projektspezifische Konvention!

3. **"bitte warten"**:
   eingetroffenes Ereignis ist im Prinzip OK, kommt aber zu unpassender Zeit und soll später verarbeitet werden
   hierzu explizite Aktionsangabe `..../defer`
   Nur an internen Transitionen erlaubt!
   Effekt: aufgeschobenes Ereignis kommt in einen *event pool* zu diesem Objekt / System und löst ggf. später eine Transition aus, nachdem sich der Zustand und/oder in Guards benutzte Variablen geändert haben
   viele Details offen, z.B. Strategie, wie aus mehreren aufgeschobenen Ereignissen das nächste abzuarbeitende gewählt wird (→ gefährliches Konzept)

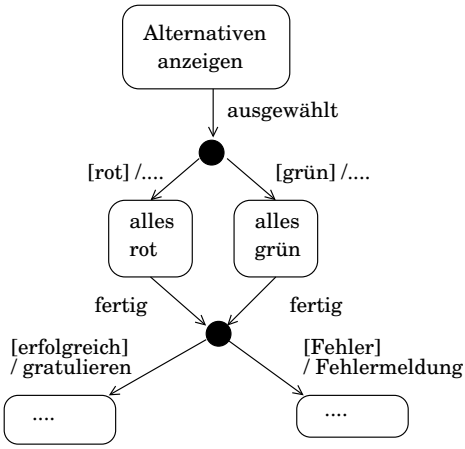
4 Pseudozustände und Transitionspfade

Steuerung / "Ablauflogik" des Zustandsautomaten bisher nur in den Transitionen und den Triggern

Pseudozustände: erweitern die Möglichkeiten, Zustandsübergänge zu steuern

imitiert bekannte Kontrollstrukturen aus Programmiersprachen (viel neue Syntax, konzeptuell wenig Neues)

Beispiel:



4.1 Pseudozustände und Transitionspfade

Merkmale von Pseudozuständen:

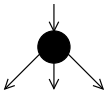
- sind zwar Knoten im Zustandsnetzwerk, *repräsentieren aber keinen Zustand!*
  d.h. dort kann kein Token parken
  (Vorsicht: deswegen ist der Endzustand überraschenderweise *kein* Pseudozustand!)
- haben ein- und ausgehende Transitionen mit teilweise unvollständigen Beschriftungen

- bilden Transitionspfade,
  z.B. von “Auswahl anzeigen” nach “alles rot”

Transitionspfade:

- beginnen und enden in “echten” Zuständen (also Knoten, die keinen Pseudozustand repräsentieren)
- können beliebig lang sein
- können unterschiedliche Pseudozustände als Etappen haben
- die kompletten Pfade werden in einem Schritt (“atomar”) durchlaufen, inkl. der Prüfung *aller Guards* und Ausführung *aller Aktionen* an den Transitionen
- nur Transitionspfade, auf denen alle Guards erfüllt sind, können durchlaufen werden
- es ist zulässig, daß mehrere Pfade durchlaufbar sind! dann wird ein erlaubter Pfad *zufällig* ausgewählt

4.2 Pseudozustand **Kreuzung (*junction*)**



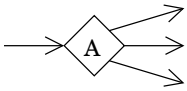
- Darstellung als schwarzer Kreis (wie Startknoten, aber geringfügig kleiner)
- eine oder mehrere hereinkommende Transitionen
- eine oder mehrere herausgehende Transitionen

- einer der ausgehenden Transitionen darf `else` als Guard haben; ist erfüllt, wenn die Guards aller anderen ausgehenden Transitionen nicht erfüllt sind
- alle Guards werden ausgewertet, *bevor* irgendwelche Aktionen ausgeführt werden (“*static conditional branch*”)
- Aktionen werden erst **nach** Durchlaufen aller Verzweigungen ausgeführt

  tückisch! entspricht nicht der Lesereihenfolge → Aktionen möglichst an die letzten Transitionen der (Teil-) Pfade

- erlauben kompakte Darstellung ähnlicher Transitionen in Transitionspfaden (s.o. Beispiel)

4.3 Pseudozustand **Auswahlknoten (*choice pseudo state*)**



- stellt eine bedingte Verzweigung dar
- Guards wie bei einer Kreuzung, inkl. `else`-Konstrukt

- Aktionen an eingehenden Transitionen werden ausgeführt, **bevor nachfolgende Guards ausgewertet werden** !!
  (UML-Diktion: *"dynamic conditional branch"*)
  → kann zu sehr komplizierten Ablauflogiken führen
  das ist der Hauptunterschied zu einer Kreuzung
- Raute kann innen einen Variablennamen enthalten, auf den sich die Test in den Guards beziehen (syntaktischer Zucker...)

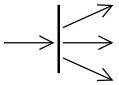
4.4 Pseudozustand **Startzustand (*initial pseudostate*)**

- nur einmal erlaubt (pro Region, s.u.)
- nur eine ausgehende Transition erlaubt
- diese darf eine Aktion haben, aber keinen Trigger oder Guard

4.5 Pseudozustand **Terminator (*terminate pseudostate*)**

- stellt Löschung des Objekts / Systems dar
- *kein Austrittsverhalten* aus dem Zustand mehr (anders als beim Endzustand)

4.6 Pseudozustand **Gabelung (*fork*)**



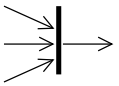
– Notation wie Petri-Netz-Transition mit *genau einem* Eingangsplatz

– ausgehende Transitionen (bzw. Transitionspfadsegmente)

1. dürfen keine Guards oder Aktionen haben
2. müssen in unterschiedlichen Regionen (s.u.) eines zusammengesetzten Zustands enden

→ keine allgemeinen Petri-Netze hiermit darstellbar

4.7 Pseudozustand **Vereinigung (*join*)**



– Notation wie Petri-Netz-Transition mit *genau einem* Ausgangsplatz

– ankommende Transitionen (bzw. Transitionspfadsegmente)

1. dürfen keine Guards oder Aktionen haben
2. müssen in unterschiedlichen Regionen (s.u.) eines zusammengesetzten Zustands starten

5 Zusammengesetzte Zustände

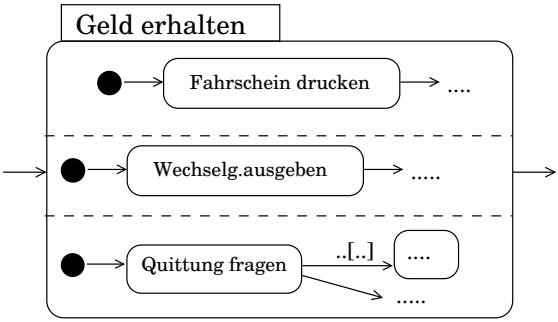
Begriff *zusammengesetzter Zustand* faßt zwei völlig verschiedene Dinge zusammen:

1. baumartige **Zustandshierarchien** wie in Lehrmodul ZÜD dargestellt:

   ein "nichtelementarer" Zustand (innerer Konten des Zustandsbaums) hat mehrere Unterzustände und das System befindet sich immer in *genau einem* der Unterzustände

2. **autarke Teilzustände** wie in Petri-Netzen[1], die durch eigene Tokens repräsentiert werden

   autarke Teilzustände werden als **Regionen** repräsentiert
   Beispiel:



   zusammengesetzter Zustand hat i.a.:

- eine oder mehrere Regionen
  → Parallelität = Zahl der Regionen, ist statisch erkennbar
  → Strukturen im Vergleich zu Petrinetzen eingeschränkt
- pro Region mehrere Unterzustände, darunter i.a. einen Startzustand

   Darstellung:

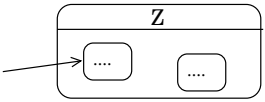
- durch Schachtelung der "Zeichenflächen"
- Zustandsname oft nicht innen, sondern als Reiter oben auf den Rechteck
- Trennung der Regionen durch gestrichelte Linie
- ggf. oben eigener Abschnitt mit *entry / exit / do*-Angaben

[1] also zusammengesetzte Zustände im wörtlichen Sinn, der Gesamtzustand des Systems setzt sich aus den Teilzuständen zusammen

5.1 Zusammengesetzte Zustände mit nur einer Region

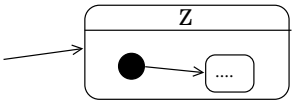
Betreten eines zusammengesetzten Zustands:

1. direkter Übergang in einen Unterzustand (*Explicit Entry*): Transitionspfeil überquert Grenze der Zeichenfläche und endet an einem Unterzustand



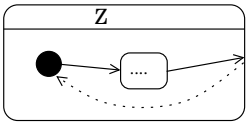
   unschön, widerspricht dem *information hiding* über die Details des zusammengesetzten Zustands

2. *Default Entry*: Transitionspfeil endet an der Grenze der Zeichenfläche



   Übergang zum inneren Startzustand und von dort aus spontan weiter...
   Frage: immer innerer Startzustand vorhanden?
   Antwort: Nein → UML-Diktion: *semantic variation point*[2]

3. Transition *von innen* endet an der Grenze der Zeichenfläche



   Übergang zum inneren Startzustand und von dort aus spontan weiter;
   kein Durchlaufen des Aus- und Eintrittsverhaltens

[2] Auf Deutsch: niemand weiß, wie es weitergeht....

Verlassen eines zusammengesetzten Zustands:

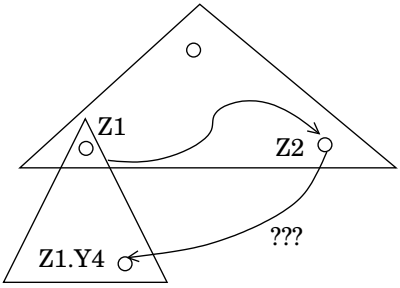
1. ein Endzustand wird erreicht:
   sofern außen eine triggerlose Transition vorhanden, wird diese ausgelöst
2. eine "äußere" Transition, die für den gesamten zusammengesetzten Zustand gilt (d.h. der Transitionspfeil beginnt am Rand der Zeichenfläche), feuert
3. direkter Übergang aus einem Unterzustand nach außen (analog zum *Explicit Entry*): Transitionspfeil überquert Grenze der Zeichenfläche und endet außerhalb

bei allen Varianten, den zusammengesetzten Zustand zu verlassen, wird das Austrittsverhalten ausgelöst

5.2 Historien

Motivation:

Beispiel einer
Zustandshierarchie



- System ist in einem speziellen Unterzustand
  Beispiel: Unterzustand Z1.Y4 des zusammeng. Zustands Z1
- irgendein Trigger führt zum Verlassen des Zustands Z1
- später gewünscht: Rückkehr zum *gleichen Unterzustand, von dem aus der zusg. Zustand (Z1) zuletzt verlassen wurde*

Geht nicht mit bisherigen Mitteln, weil dieser Unterzustand nicht statisch festliegt

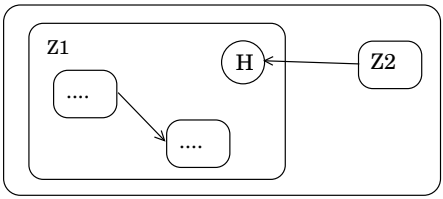
→ man bräuchte ein "Gedächtnis", das sich für jeden zusammengesetzten Zustand merkt, in welchem Unterzustand man das letzte Mal war

Lösung des Problems in der UML: mit Historienzuständen

5.2.1 Pseudozustand **flache Historie (*shallow history*)**

- dargestellt als Kreis mit "H" in der Mitte
- nur als Unterzustand eines zusammengesetzten Zustands erlaubt
- 0 oder 1 ausgehende zu Transition erlaubt
- keine von innen ankommende Transition erlaubt

Beispiel:



Verhalten beim Betreten des Zustands Z1 über den Historien-Knoten:

1. wenn das System *vorher schon einmal im Zustand Z1 war*: seinerzeit aktiven Zustand auf *dieser* Verfeinerungsebene betreten
2. andernfalls, wenn vom Historien-Knoten eine Transition zu einem Zustand ausgeht: in den Zielzustand dieser Transition (*default shallow history state*) übergehen
3. andernfalls, wenn Startknoten auf dieser Verfeinerungsebene vorhanden: dort ausgehender Transition folgen
4. andernfalls: fehlerhaftes Modell oder unklare Situation

5.2.2 Pseudozustand **tiefe Historie (*deep history*)**

- dargestellt als Kreis mit "H*" in der Mitte
- alles wie bei flacher Historie, bis auf folgenden Unterschied

Verhalten, wenn der über den Historien-Knoten erreichte Zielknoten selbst wieder zusammengesetzt ist:

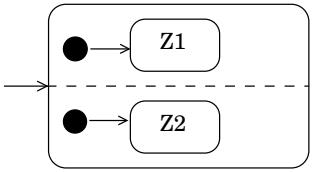
- flache Historie: Zielknoten wird neu betreten (über den lokalen Startzustand)
- tiefe Historie: dort zuletzt aktiver Unterzustand wird betreten usw. rekursiv abwärts

  → für jeden Verfeinerungsschritt muß der zuletzt aktive Zustand bekannt sein

5.3 Regionen

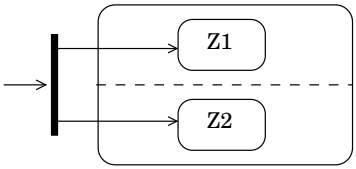
Betreten eines Zustands mit mehreren Regionen:

1. *default entry*: Transition endet am Rand des Zustands



   Wirkung: pro Region wird der Startzustand aktiviert

2. *explicit entry*: eine oder mehrere Transitionen (die von außen aus einer Gabelung stammen) überqueren den Rand des zusammengesetzten Zustands und enden in inneren Zuständen (pro Region max. 1)



Wirkung: die direkt angesteuerten Zustände werden aktiv; in Regionen, in denen keine Transition endet, wird der Startzustand aktiviert

Verlassen eines Zustands mit Regionen:

1. wenn in allen Regionen der Endzustand erreicht und außen eine triggerlose Transition vorhanden ist: diese feuern

   Regionen, die schon den Endzustand erreicht haben, warten darauf, daß alle anderen Regionen ebenfalls den Endzustand erreichen und ignorieren währenddessen alle Trigger
2. wenn eine Transition für den zusammengesetzten Zustand feuert: alle internen Zustände werden sofort inaktiv
3. wenn eine Transition für einen Unterzustand, die außerhalb des zusammengesetzten Zustands endet, feuert: alle internen Zustände werden sofort inaktiv, d.h. auch alle anderen Regionen werden abgebrochen

Literatur

[PN] Kelter, U.: Lehrmodul “Petri-Netze”; 2003

[ZUED] Kelter, U.: Lehrmodul “Zustandsübergangsdiagramme”; 2003

