Richtlinien zur Ableitung von C++-Kode aus UML/OMT-Zustandsdiagrammen der Entwurfs-phase

Angepaßter Ausschnitt aus der
Projektarbeit von

Karsten Buch

Mai 1997

AG Software Engineering
Fachbereich Informatik
Universität Kaiserslautern

Betreuer: Prof. Dr. H. Dieter Rombach
Dipl.-Inform. Antje von Knethen

Inhaltsverzeichnis

Kapitel 1

Zur Semantik von Zustandsdiagrammen

1.1 Zustandstypen

Zur einfacheren Formulierung der Semantik und der Richtlinien werden für die drei möglichen Typen eines Zustands folgende Begriffe verwendet (s. [Day93]):

1. **BASIC:** der Zustand ist nicht weiter in Unterzustände verfeinert.
2. **OR:** der Zustand ist verfeinert in Zustände, die im wechselseitigen Ausschluß stehen (XOR).
3. **AND:** der Zustand ist verfeinert in parallele Substates.

In Abb. 1 sind die Zustände A vom Typ AND, B und C vom Typ OR und D, E, F und G vom Typ BASIC.

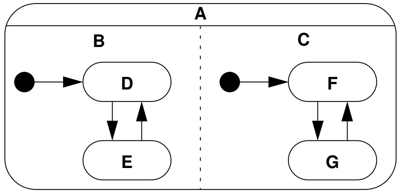


Abb. 1: Zustandstypen

1.2 Was ist ein Schritt?

Ein UML/OMT-Zustandsdiagramm beschreibt einen synchronen Zustandsautomaten. Die Realisierung mit Hilfe eines Laufzeitsystems führt dazu, daß Objekte nur scheinbar parallel sind und sich Rechenzeit teilen müssen. Das hat zur Folge, daß auf innerhalb eines bestimmten Zeitintervalls eintreffende Ereignisse bzw. auf wahr werdende Bedingungen evtl. nicht sofort ein Zustandsübergang erfolgt. Dieser wird zu bestimmten Zeitpunkten (d.h. bei den Aufrufen durch das Laufzeitsystem) durch Bestimmung der nächsten Konfiguration des Zustandsautomaten und Ausführen von Zustandsübergängen und Aktionen durchgeführt. Im folgenden wird dieser Vorgang (d.h. das Bestimmen und Erreichen der neuen Konfiguration) als Schritt bezeichnet. Als weitere Folgerung ergibt sich, daß Bedingungen und Events nicht notwendigerweise in der Reihenfolge ihres Eintretens bearbeitet werden. Hierdurch können Nichtdeterminismen erzeugt werden, die zu einem Fehlverhalten führen können. Diese Situation tritt allerdings nur dann ein, wenn im Zustandsdiagramm implizite Annahmen über den Eintrittszeitpunkt von Ereignissen oder Bedingungen gemacht werden. Solche Annahmen sind in einer realen Umgebung nicht sinnvoll, sofern nicht kausale Beziehungen zwischen den Ereignissen bestehen. Es unterscheiden sich z.B. die Laufzeiten von Nachrichten, so daß eine Annahme über deren Reihenfolge willkürlich ist. Kausalbeziehungen sollten daher explizit gemacht werden.

Abb. 2 verdeutlicht den Sachverhalt. Werden durch Änderungen der externen Variablen x und y beide Bedingungen in der Zeit zwischen zwei Schritten wahr, d.h. beide Transitionen könnten ausgeführt werden, so muß man eine der Transitionen auswählen. Dieses Problem behandelt Abschnitt 1.3.

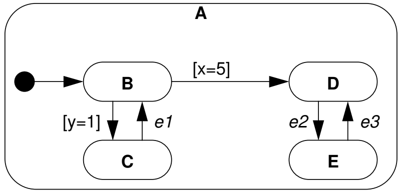


Abb. 2: Nichtdeterminismen

Der Kernpunkt der Semantik liegt in der Definition einer legalen Zustandskonfiguration und darauf aufbauend der Definition eines legalen Schrittes, der eine legale Zustandskonfiguration in eine andere überführt. Unter der Annahme, daß sich ein Zustandsdiagramm momentan in einer konsistenten Konfiguration befindet, müssen nach einem Schritt die folgenden Bedingungen erfüllt sein, damit sich der Zustandsautomat wieder in einer konsistenten Konfiguration befindet (s. auch [Day93]).

1.2.1 Bedingungen der Zustandskonfiguration

1. Befindet sich das System in einem OR-Zustand, dann muß es sich in genau einem seiner Substates befinden.
2. Befindet sich das System in einem AND-Zustand, dann muß es sich in jedem seiner Substates befinden.
3. Befindet sich das System in einem BASIC-Zustand, so gibt es an diesen Zustand keine weiteren Bedingungen.

Legale Zustandskonfigurationen in Abb. 1 sind: {A,B,C,D,F}, {A,B,C,E,F}, {A,B,C,D,G} und {A,B,C,E,G}.

1.2.2 Bedingungen der Transitionen

Bei der Durchführung eines Schritts werden die Transitionen ausgeführt, die den folgenden Bedingungen genügen:

1. Eine Transition kann nur dann vorgenommen werden, wenn sie ausführbar ist. Dies ist der Fall, wenn sich das System in dem Zustand befindet, von dem die Transition ausgeht und die damit assoziierte Bedingung erfüllt bzw. das assoziierte Event eingetreten ist.
2. Zwischen den Zuständen eines OR-Zustands kann nur eine Transition ausgeführt werden.
3. In einem AND-Zustand gilt dies entsprechend für jeden parallelen Substate.
4. Wenn mehrere Transitionen desselben Ausgangszustandes ausführbar sind, so wird eine davon entsprechend den Regeln in Abschnitt 1.3 ausgewählt.
5. Die Bedingungen der Transitionen werden auf Basis der Variablenwerte und Events der Ausgangskonfiguration ausgewertet. Dies ist insbesondere dann von Bedeutung, wenn Variablen durch Aktionen in parallelen Zuständen geändert werden können (s. Abschnitt 3.6).
6. Sind keine Transitionen ausführbar, so wird ein leerer Schritt gemacht (d.h. nichts getan). Dies ist ein legaler Schritt.

1.2.3 Bedingungen der Ergebnisse der Transitionen

Werden die nach obigen Regeln gewählten Transitionen ausgeführt, so müssen die entsprechenden Aktionen durchgeführt werden. Dafür werden folgende Bedingungen formuliert:

1. Wird eine Transition ausgeführt, so befindet sich das System in der neuen Konfiguration in dem Zielzustand der Transition. Die Aktionen, die mit der Transition assoziiert sind, werden ausgeführt.
2. Wenn eine Variable durch keine Transition verändert wird, so behält sie ihren alten Wert.
3. Wenn mehrere Transitionen eine Variable verändern, so ergibt sich der neue Wert der Variablen aus genau einer der Modifikationen und zwar relativ zum Wert der Variablen in der Ausgangskonfiguration. Dies ist insbesondere wichtig bei parallelen Substates und Entry-/Exit-Hierarchien. Hier muß man sehr sorgfältig mit der Zuweisung an Variablen umgehen, damit nicht unerwartete Effekte eintreten und zu einem Verhalten führen, das nicht der Absicht der Entwerfer entspricht. Dies ist im Entwurf zu berücksichtigen.
4. Für die Reihenfolge der Ausführung der Aktionen gilt: Aktionen auf den eingehenden Transitionen, Entry-Aktionen, DO-Aktivitäten, Exit-Aktionen, Aktionen auf den ausgehenden Transitionen (s. dazu auch 1.5).

1.3 Auflösung von Nichtdeterminismen bei Transitionen

Nichtdeterminismen müssen nicht zwangsläufig Entwurfsfehler sein. Man sollte sich aber bewußt sein, daß das Verhalten eines Objekts dadurch undurchsichtiger und schwerer testbar wird. Wie schon bei der Formulierung eines Schrittes in 1.2 erwähnt, sollten Kausalbeziehungen explizit modelliert werden. Sind Nichtdeterminismen vorhanden, so wird hier unterstellt, daß diese (nach kurzer Zeit) zur selben Konfiguration (d.h. demselben Objektverhalten) führen. Wird eine explizite Modellierung von Nichtdeterminismen gewünscht (z.B. zur Implementierung von Fairneßbedingungen), so muß diese vom

Programmierer selbst vorgenommen werden. Im folgenden wird angenommen, daß ein Nichtdeterminismus stets aufgelöst werden soll. Dafür werden die nachstehenden Regeln formuliert (nach absteigender Priorität):

1. Transitionen, die von einem Superstate ausgehen, haben Priorität vor Transitionen, die von seinen Substates ausgehen. Dies erscheint sinnvoll, da Superstates häufig genutzt werden, um Abbruchbedingungen und Interrupts zu realisieren, die für die Menge der Substates gelten sollen. Diese Festlegung wurde aus [Day93] übernommen und widerspricht der ursprünglichen Semantik von Harel (s. [Har87a]). Die Entscheidung wurde aus zwei Gründen getroffen:
   - Die Harelsche Definition steht im Widerspruch zur These, daß durch die Verfeinerung von Zuständen das Objektverhalten verfeinert (dies gilt insbesondere bei der Verfeinerung in Vererbungshierarchien (s. [Col92])), nicht aber verändert werden soll.
   - Die Entscheidung ist die einzige (einfache) Möglichkeit, Prioritäten im Zustandsdiagramm festzulegen. Dies kann einfach durch das Einführen von Superstates geschehen.
2. Events haben Vorrang vor Bedingungen. Eine mögliche Konkurrenz zwischen den Events wird dadurch ausgeschlossen, daß ein eintreffendes Event sofort bearbeitet wird. Da keine internen Events zugelassen sind und verschickte Events während des Schrittes nicht zur Generierung von Events für die auslösende Klasse führen dürfen (siehe Kapitel 4.1), kann zu einem Zeitpunkt nur ein Event existieren. Ein Abweichen von dieser Vereinbarung hat zur Folge, daß für Events z.B. eine Variable definiert werden muß, die das Eintreten signalisiert, so daß bei der Durchführung eines Schrittes auch eine Auswahl unter konkurrierenden Events getroffen werden muß, was nicht sinnvoll erscheint (s. dazu auch Kapitel 3). Außerdem erlaubt die Festlegung, daß Events einfach durch Methodenaufrufe realisiert werden können, die dann jederzeit (auch außerhalb der Aktivierungen durch das Laufzeitsystem) Transitionen auslösen können. Dies gibt auch den aktiven Charakter der Events (im Gegensatz zu Bedingungen) wieder. (Bemerkung: Die sofortige Bearbeitung von Events steht nicht im Widerspruch zur Durchführung von Zustandsübergängen zu diskreten Zeitpunkten. Die wesentlichen Punkte, nämlich nicht ständig alle Bedingungen prüfen zu müssen und Microsteps zu vermeiden, bleiben davon unberührt.)
3. Es bleibt der Fall zu betrachten, daß mehrere durch Bedingungen bewachte Transitionen ausführbar sind. Hier wird eine explizite Priorisierung vorgenommen. Diese Regel könnte auch als eine sinnvolle Erweiterung der OMT/UML-Notation interpretiert werden, da Zustandsdiagramme keine explizite Möglichkeit zur Formulierung von Prioritäten vorsehen. Die Entscheidung, bei der Implementierung vorher nicht vorhandene (und nicht durch die Punkte 1. und 2. entstandene) Prioritäten einzuführen, sollte aber dokumentiert werden.

1.4 Hierarchie als „Vererbung“

Die Verschachtelung von Zustandsdiagrammen hat zur Folge, daß alle Zustände, die nicht vom Typ BASIC sind, als abstrakte Zustände zu sehen sind. Die Abstraktion liegt darin, daß alle Transitionen, die von einem Superstate ausgehen bzw. zu ihm hin nicht direkt zu einer neuen Konfiguration führen,

sondern erst eine rekursive Fortführung auf die Substates bis zum Erreichen der elementaren Zustände eine legale Konfiguration erzeugt. D.h. alle Transitionen von einem Superstate werden so interpretiert, als gingen sie von allen seinen Substates aus. Hier wird unterstellt, daß diese Interpretation ebenso für Entry- und Exit-Aktionen (s. unten), DO-Aktivitäten und interne Aktionen gilt. So ist auch eine Abstraktion der Aktivitäten möglich. Diese kann z.B. nützlich sein, wenn sich in allen Substates eine Berechnung wiederholt. Abb. 3 zeigt die vervollständigte Notation.

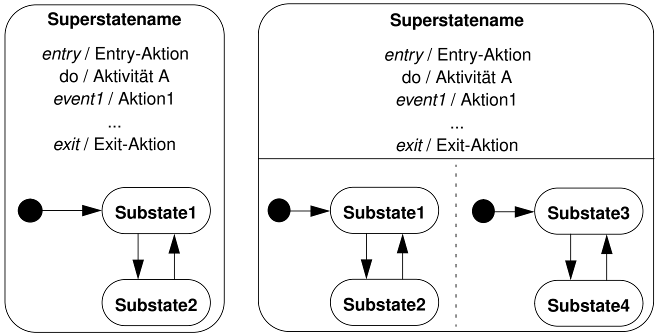


Abb. 3: Vervollständigte Notation für Hierarchien

1.5 Entry- und Exit-Hierarchien

Gibt es eine Aktion, die bei jeder Transition aus einem bzw. in einen Zustand ausgeführt werden soll, so kann man hierfür äquivalent die Exit- bzw. Entry-Notation verwenden. Setzt man diese Interpretation der Entry- und Exit-Aktionen rekursiv fort, so ergibt sich, daß in beiden Fällen in

Abb. 4 (a) und (b) die Zustände A, B und C bei Eintreten des Events e verlassen und die Zustände D, E und F betreten werden, d.h. die entsprechenden Exit- und Entry-Aktionen müssen ausgeführt werden. Dies geschieht bei den Exit-Aktionen von innen nach außen und bei den Entry-Aktionen von außen nach innen. Hier muß man also die obersten Zustände bestimmen, die durch die Transition verlassen bzw. betreten werden und die entsprechenden Hierarchien durchlaufen. Im Beispiel ergibt sich die Folge A3, A2, A1, A, A4, A5, A6.

Bemerkung: Die in den Richtlinien angegebenen Entry-Hierarchien können auch dann sinnvoll verwendet werden, wenn in den Statemodels keine Entry-Aktionen spezifiziert sind. Die Entry-Hierarchie kann dann genutzt werden, um die Default-Zustände von Superstates automatisch in den Entry-Methoden zu bestimmen und zu aktivieren. Der Programmierer muß die Default-Zustände nur einmal bestimmen und in den Entry-Methoden eintragen. Dies erleichtert eine Änderung der Default-Zustände, die dann lokal in der jeweiligen Entry-Methode durchgeführt werden kann. Dadurch müssen nicht alle Transitionen, die zum Superstate zeigen (und damit seinen Default-Zustand betreten), geändert werden. Außerdem kann die Bestimmung der neuen Zustandskonfiguration dadurch vereinfacht werden, daß man diese nur soweit definiert, bis man auf Default-Werte trifft. Alles weitere übernehmen dann die Entry-Methoden.

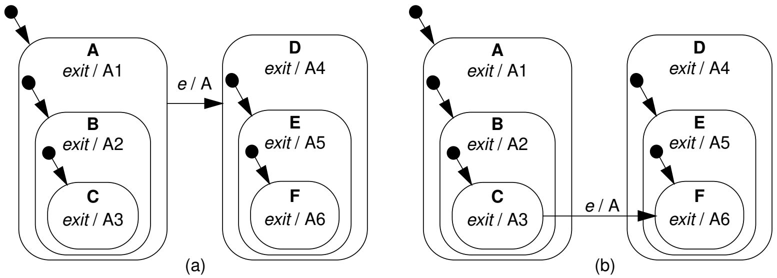


Abb. 4: Entry- und Exit-Hierarchien

1.6 DO-Aktivitäten

Bei der Wirkung von Transitionen auf DO-Aktivitäten entsteht ein Problem, das jedoch durch die gemachten Einschränkungen wieder aufgelöst wird. Bei DO-Aktivitäten können zwei Möglichkeiten unterschieden werden: Eine DO-Aktivität beschreibt eine Berechnung, die beim Betreten eines Zustands begonnen wird und entweder nach einer gewissen Zeit von selbst endet oder solange ausgeführt wird, bis der Zustand verlassen wird. Dabei wird die Aktivität abgebrochen, sobald eine Transition ausführbar ist, die den Zustand verläßt. Bei der Implementierung werden die dauerhaften Aktivitäten periodisch durch Call-Back-Methoden (s. Abschnitt 2.1) angestoßen und auch zu Ende ausgeführt, da die Methoden nicht unterbrechbar sind. Da zu einem Zeitpunkt nur das aktuelle Objekt aktiv ist (wegen der Single-Thread-Realisierung) und dieses auch keine internen Events verschicken darf, ist eine Unterbrechung der Aktivität durch eintretende Events überhaupt nicht möglich. Problematisch sind nun noch Aktivitäten, die länger dauern als das Aufrufintervall des Laufzeitsystems. Diese müssen in Teile zerlegt werden, die in verschiedenen Aufrufintervallen erledigt (etwa durch die Realisierung eines Zählers, der die Aufrufanzahl speichert) oder mitteilungsorientiert angestoßen werden (etwa durch das Senden einer Nachricht an einen Server, der die Aufgabe erledigt und bei Beendigung ein Event zurücksendet). Akzeptiert man die Einschränkung der Zustandsübergänge zu diskreten Zeitpunkten, so ist das erzeugte Verhalten mit der Semantik von OMT/UML vereinbar.

Kapitel 2

Richtlinien für die Implementierung

Die hier vorgestellten Richtlinien gelten jeweils für das Zustandsdiagramm einer Klasse aus dem Klassendiagramm. Als C++-Richtlinien für die Programmbeispiele wurde [CoD96b] zugrunde gelegt (die Bezeichner und Kommentare sind allerdings in Deutsch gehalten). In Anlehnung an den Aufbau dieser Richtlinien enthalten die folgenden Abschnitte die nachstehenden Punkte (einige davon sind optional):

- ***Problem:*** zeigt kurz, worum es in der Richtlinie geht.
- ***Richtlinien*** sind aus dem Problem abgeleitet und verbindlich für den Implementierer.
- ***Empfehlungen*** sollten nach Möglichkeit berücksichtigt werden.
- ***Beispiel:*** kurzes Kodebeispiel zur Verdeutlichung der Richtlinien
- ***Bemerkungen*** zur Problematik oder zur Verwendung der Richtlinien.

Einige der Richtlinien basieren auf [CoD96c]. Weitere Punkte wurden aus den Kode-Beispielen in [RAc96] abgeleitet.

Die Umsetzung der Richtlinien soll anhand des Beispiels einer Ampel verdeutlicht werden. Dieses ist angelehnt an [RAc96]. Als Vereinfachung wird angenommen, daß direkt zwischen rot und grün ohne Gelbphase umgeschaltet wird. Bei einem Fehler (Event Reset) wird ein Reset durchgeführt, und die Ampel blinkt gelb. Ein Event Init führt zur Reaktivierung des Normalbetriebs. Die Umschaltung der Phasen erfolgt bei einem Timeout, der als externes Event (TimeoutNS bzw. TimeoutOW) angenommen wird oder wenn mehr als zwei Autos in der Gegenrichtung neu ankommen. Die Ankunft eines Wagens wird durch das Ereignis NeuOW bzw. NeuNS signalisiert. Überquert ein Wagen bei rot die Kreuzung (AutoOW bzw. AutoNS), so soll eine Kamera ein Photo schießen.

Abb. 6 zeigt eine mögliche Realisierung als Zustandsdiagramm. Der komplette Kode für das Beispiel ist als Anhang beigefügt.

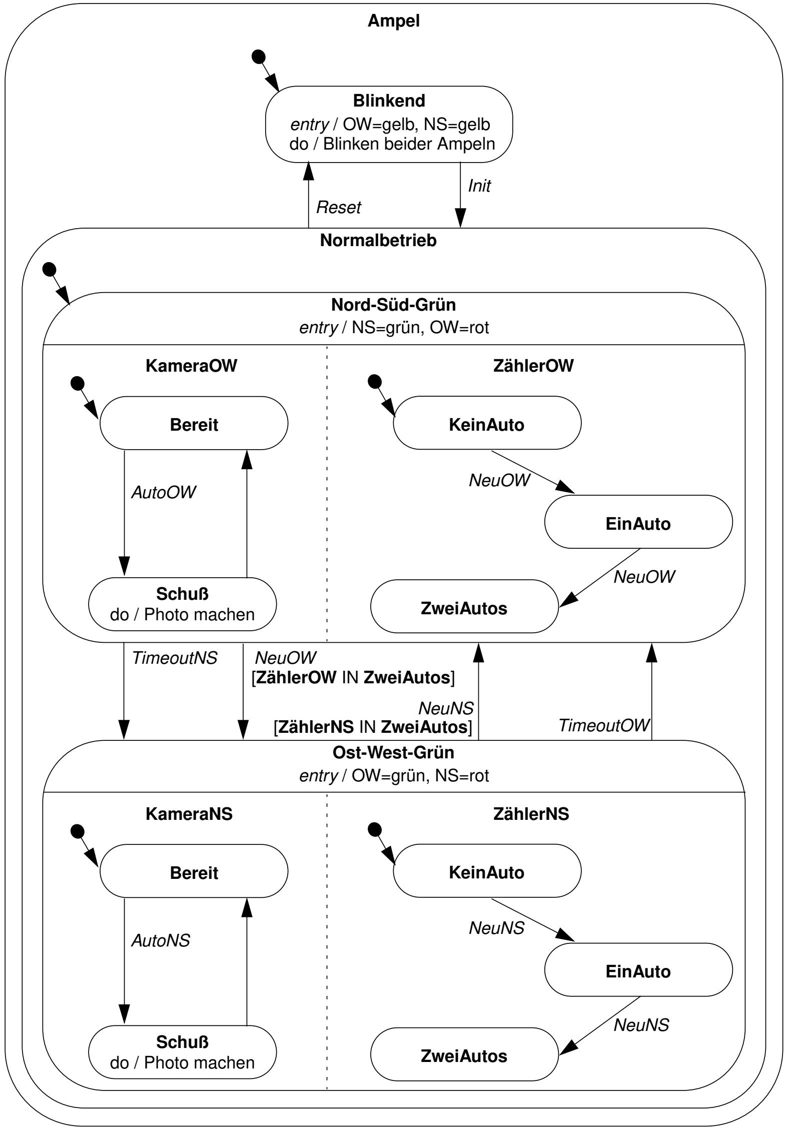


Abb. 5: Zustandsdiagramm für das Ampelbeispiel

2.1 Call-Back-Methoden

Problem

Der Zustand eines Objekts soll periodisch durch die Ausführung eines Schrittes aktualisiert werden.

Bemerkung

Für passive Klassen, bei denen Zustandsübergänge nur durch Events erfolgen, kann die Anmeldung beim Laufzeitsystem entfallen. Die Zustandsübergänge werden dann beim Eintreffen eines entsprechenden Events durchgeführt. Die Entscheidung, ob in diesem Fall Call-Back-Methoden definiert werden sollen, hängt von der Art der Realisierung der Events ab (s. Abschnitt 2.5).

Richtlinien

Um eine periodische Aktualisierung des Zustandsautomaten zu erreichen, wird für jede aktive Klasse eine sog. *direkte Call-Back-Methode* definiert, die im Konstruktor der Klasse einem zur Verfügung gestellten Laufzeitsystem übergeben wird.

Als Laufzeitsystem wird das Kommunikationssystem BAuOS des CoDEx-Projekts (s. [CoD96a]) wiederverwendet. Eine detaillierte Beschreibung von Aufbau, Funktionsweise und Verwendung des Schedulers kann in der Dokumentation nachgelesen werden. An dieser Stelle werden nur die folgenden Punkte benötigt:

- Die Klasse, die das vorgegebene Zustandsdiagramm realisiert, ist als Unterklasse der vom Laufzeitsystem zur Verfügung gestellten Klasse `ScheduledFct` zu deklarieren.
- Im Konstruktor der Klasse muß die geerbte Funktion `registerCallBack` aufgerufen werden, die das Objekt beim Scheduler anmeldet.
- Im Destruktor der Klasse muß die geerbte Funktion `unregisterCallBack` aufgerufen werden, die das Objekt beim Scheduler abmeldet.

Beispiel

Im Kodebeispiel sind die hier relevanten Teile (An- bzw. Abmelden beim Laufzeitsystem und Deklaration der direkten Call-Back-Methode sowie Deklaration als Unterklasse von ScheduledFct) fett gedruckt.

```
// Header und Includes...

// Eigenes Zustandsdiagramm
class Ampel : public ScheduledFct
{
public:
        // Konstruktor
        Ampel()
        {
          // Initialisierungen machen
          // Initialen Zustand herstellen
          // Anmelden beim Laufzeitsystem
          registerCallBack ();
        }; // Ampel::Ampel
```

```
        // Die Call-Back-Methode realisiert das Zustandsdiagramm
        void callBack (void);

        // Destruktor
        ~Ampel()
        {
          // Abmelden beim Laufzeitsystem
          unregisterCallBack ();
        }; // Ampel::~Ampel

        // weitere Methoden...
}; // class Ampel
```

2.2 Die Zustandskonfiguration ohne Verwendung von IN

Problem

Ein Objekt muß sich merken, in welcher Zustandskonfiguration sich sein Zustandsautomat momentan befindet. Dabei sollen an das Zustandsdiagramm angelehnte Zustandsbezeichnungen verwendet werden.

Richtlinien

Zur Darstellung der Zustandskonfiguration einer Klasse wird eine Reihe von Attributen definiert. Dies geschieht nach folgenden Regeln, wobei wieder auf die rekursive Anwendbarkeit auf Substates zu achten ist:

1. Ist ein Zustand vom Typ OR, so wird für ihn ein Attribut definiert. Dieses stellt einen Aufzählungstyp (enum) dar, dessen Werte durch die möglichen Substates gegeben sind.
2. Ist der Zustand vom Typ AND, so erhält er kein eigenes Attribut. Für jeden seiner parallelen Substates wird ein Attribut definiert.
3. Zustände vom Typ BASIC finden sich als Attributwerte ihres Superstates wieder.

Bemerkung

Da eine geschachtelte Definition von Aufzählungstypen nicht möglich ist, wurde hier auf die Darstellung der zusammengesetzten Zustandskonfiguration durch mehrere Attribute ausgewichen. Da C++ für die Werte einer Aufzählung Konstanten definiert, ist auf die Eindeutigkeit der Attributwerte zu achten.

Empfehlungen

- Attributbezeichnern sollte „state“ vorangestellt werden.
- Um Attributwerte eindeutig zu machen, sollten sinnvolle Ergänzungen der Bezeichner durch Teile des Namens des Superstates gemacht werden.

Beispiel

Im Ampelbeispiel (s. Abb. 6) ergeben sich folgende Attribute:

```
// Root-Zustand Ampel ist vom Typ OR -> 1 Attribut
```



```
enum {Cblinkend, CnormalBetrieb} stateAmpel;
// Normalbetrieb ist vom Typ OR -> 1 Attribut
enum {CowGruen, CnsGruen} stateNormalbetrieb;

// OWGruen ist vom Typ AND -> 2 Attribute für die 2 Substates
enum {CnsBereit, CnsSchuss} stateKameraNS;
enum {CnsKeinAuto, CnsEinAuto, CnsZweiAutos} stateZaehlerNS;

// NSGruen ist vom Typ AND -> 2 Attribute für die 2 Substates
enum {CowBereit, CowSchuss} stateKameraOW;
enum {CowKeinAuto, CowEinAuto, CowZweiAutos} stateZaehlerOW;
```

2.3 Die Zustandskonfiguration bei Verwendung von IN

Problem

Bei Verwendung von IN in parallelen Substates werden evtl. die Zustandskonfigurationen vor und nach dem aktuellen Schritt benötigt. Betrachtet man Abb. 6, so wird deutlich, daß der Übergang von A nach B nicht gemacht würde, wenn der Übergang von C nach D zuerst durchgeführt und die Konfiguration überschrieben wird.

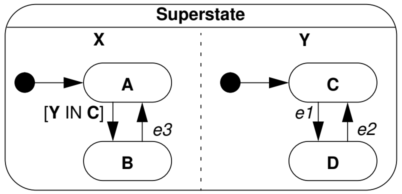


Abb. 6: Probleme bei der Verwendung von IN

2.3.1 Definition der Zustandskonfiguration

Richtlinien

Es werden Attribut*typen* mit denselben Attributwerten wie oben definiert. Ein weiterer Attributtyp mit der Vorsilbe „stateConfig“ wird erstellt, der von null beginnend die „state“-Attribute aufzählt. Dabei werden die Attribute, die zu einem AND-Zustand gehören, auch nacheinander aufgezählt. Ein Integer-Array, das so viele Elemente hat wie es Attribute gibt, nimmt nun die Zustandskonfiguration auf. Diese umständlich erscheinende Definition erlaubt die Begrenzung der Variablenmenge, da für jedes Zustandsattribut der momentane und der zukünftige Zustand gespeichert werden müssen. Die Vorgehensweise soll am Ampelbeispiel erläutert werden:

```
// Die enum-Variablen von oben müssen nun als Typen deklariert
// sein, damit die Attributwerte hier zur Verfügung stehen, d.h.
// hier steht: stateAmpel {Cblinkend, CnormalBetrieb}; usw.

enum stateConfigAmpel {Campel=0,
```

```
                           CnormalBetrieb1, // 1 wg. Eindeutigkeit!
                           CkameraNS, CzaehlerNS,
                           CkameraOW, CzaehlerOW};
int prevState[6], newState[6];
```

Es werden nun zwei dieser Konfigurationen benutzt. Die eine (prevState) enthält den Zustand vor dem Schritt, die andere (newState) den Zustand, der durch die Ausführung des Schrittes erreicht wird. Mit diesen zwei Konfigurationen ist eine sichere Realisierung von IN möglich.

Bemerkung

Natürlich lassen sich die beiden Konfigurationen auch durch die Definition von jeweils zwei Variablen pro Attributtyp realisieren. Die Darstellung als Array vereinfacht aber die nötige Initialisierung und Aktualisierung bei größeren Zustandsdiagrammen erheblich.

2.3.2 Verwendung der Zustandskonfiguration

Problem

Die neue Konfiguration newState muß vor Abarbeitung der Transitionen initialisiert werden. Außerdem muß nach Durchführen eines Schrittes die Konfiguration prevState aktualisiert werden.

Richtlinien

- Zu Beginn der Methode callBack wird die newState-Konfiguration mit Dummy-Werten initialisiert.
- Am Ende der callBack-Methode erfolgt die Aktualisierung der prevState-Konfiguration. Dabei werden die Elemente von prevState durch die entsprechenden Elemente von newState an den Stellen ersetzt, die geändert wurden (d.h. diejenigen, die nicht mehr den Dummy-Wert enthalten).
- Das Überprüfen der IN-Bedingung erfolgt durch Testen des Attributwerts des betreffenden Zustands. Im Ampelbeispiel könnte das folgendermaßen aussehen: „[ZählerOW IN ZweiAutos]“ entspricht „if (prevState[CzaehlerOW]==CowZweiAutos) “.

Beispiel

Im Ampelbeispiel sieht der Kode dann folgendermaßen aus (beachte: im Beispiel muß nicht unbedingt mit der Array-Variante gearbeitet werden. Die Transitionen mit IN sind hier unkritisch, da sie Priorität haben vor evtl. Zustandsänderungen in den Substates):

```
const int Cdummy = -10;

void Ampel::callBack (void)
{
  int i;
  // Initialisierung der newState-Konfiguration
  for (i=Campel; i<=CzaehlerOW; i++) {
    newState[i] = Cdummy;
  } // for
```



```
  // Kode fuer die Realisierung des Zustandsdiagramms

  // Aktualisierung der prevState-Konfiguration
  for (i=Campel; i<=CzaehlerOW; i++) {
    if (newState[i] != Cdummy) {
      prevState[i] = newState[i];
    } // if
  } // for
} // Ampel::callBack
```

Bemerkungen

In den folgenden Richtlinien wird die Realisierung der Zustandskonfiguration wie in Abschnitt 2.2 zugrunde gelegt, um zu vermeiden, daß die Richtlinien durch die Darstellung mehrerer Implementierungsmöglichkeiten zu umfangreich werden. Eine Umstellung auf die Array-Variante ist aber einfach möglich.

Dabei ist zu beachten, daß bei der Abarbeitung von Exit-Hierarchien und allen Aktivitäten und Aktionen innerhalb eines Zustands auf der Konfiguration prevState gearbeitet wird. Dies bedeutet, daß man z.B. „stateNormalbetrieb“ im Kode durch „prevState[CnormalBetrieb1]“ ersetzen muß.

Für die Berechnung der neuen Zustandskonfiguration und der Ausführung der Entry-Hierarchie wird das Array newState benutzt, d.h. man muß z.B. „stateNormalbetrieb“ durch „newState[CnormalBetrieb1]“ ersetzen.

Beispiel

Neben Abb. 7 ist der Kode für den parallelen Substate X aus Abb. 6 angegeben. Zum Verständnis sind die folgenden Abschnitte 2.4-2.6 notwendig. Es soll hier nur die Verwendung der Zustandskonfiguration in der Array-Variante aufgezeigt werden.

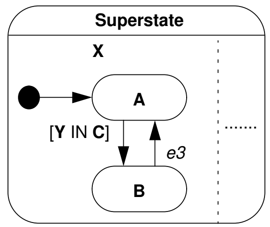


Abb. 7: Zustandsdiagramm mit Verwendung von „IN“

```
void Zustandsdiagramm::callBackX ()
{
  switch (prevState[Cx]) {
    case Ca:
      if (prevState[Cy]==Cc) {
        newState[Cx]=Cb;
      } // if
      break;
    case Cb:
      if (evente3) {
        newState[Cx]=Ca;
      } // if
      break;
    default:
      break;
  } // switch
} // Zustandsdiagramm::callBackX
```

2.4 Implementierung der Zustände und Zustandsübergänge

Problem

- Bei jedem Aufruf der callBack-Methode müssen für jeden Zustand, in dem sich das Objekt befindet, einige Berechnungen durchgeführt werden:
- Überprüfen, ob eine Transition ausführbar ist.
- Evtl. Auswahl der Transition, die ausgeführt wird und Berechnung des neuen Zustands.
- Evtl. müssen Zustandsübergänge gemacht werden. Dabei werden Exit- und Entry-Aktionen sowie Aktionen auf den ausgehenden Transitionen ausgeführt.
- Ansonsten Ausführen von Aktivitäten.

Die drei folgenden Abschnitte beschreiben den Koderahmen für die Implementierung der drei Zustandstypen AND, OR und BASIC.

Bemerkung

Die angegebenen Kodesegmente sind vollständig in dem Sinn, daß sie eine lauffähige Implementierung des jeweiligen Teils des Zustandsdiagramms darstellen. Deshalb enthalten sie auch die erst später behandelten Teile wie die Bestimmung der neuen Zustandskonfiguration oder die Behandlung von Events.

2.4.1 Zustände vom Typ BASIC

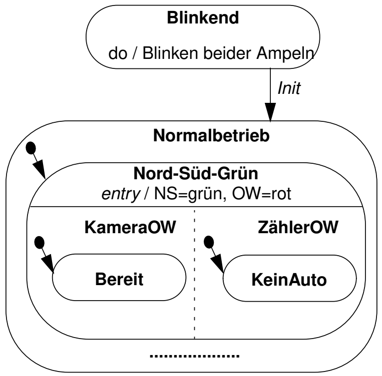
Richtlinien

Für jeden Zustand vom Typ BASIC wird eine IF-Anweisung definiert. Die Reihenfolge der Bedingungen realisiert die Prioritäten der Transitionen wie in Abschnitt 1.3 definiert. Der Programmierer muß daher zuerst die Prioritäten bestimmen. Damit ergibt sich ein Kodesegment der Form:

```
if (...) {
  // Exit-Hierarchie durchlaufen
  // Ausgelöste Aktionen ausführen
  // neue Zustandskonfiguration berechnen
  // Entry-Hierarchie durchlaufen
} // if
else if (...) {
      ...
} // else if
else { // es findet kein Zustandswechsel statt, daher:
  // DO-Aktivitäten ausführen
} // else
```


Beispiel

Für den Zustand Blinkend aus dem Ampelbeispiel sieht der Kode so aus:




```
if (eventInit) {
  // keine Exit-Aktionen ...
  // neue Konfig. berechnen:
  stateAmpel = CnormalBetrieb;
  stateNormalbetrieb=CnsGruen;
  stateKameraOW=CowBereit;
  stateZaehlerOW=CowKeinAuto;
  // keine Aktionen ...
  // Entry-Aktionen ausführen:
  ns=Cgruen;
  ow=Crot;
} // if
else {
  // DO-Aktivitaeten:
  nsBlinken (); // noch zu kodieren
  owBlinken (); // noch zu kodieren
} // else
```

Abb. 8: Blinkend-Zustand der Ampel

2.4.2 Zustände vom Typ OR

Richtlinien

Für jeden Zustand vom Typ OR wird eine indirekte Call-Back-Methode „callBack<u>Zustandsname</u>“ definiert (Ist der oberste Zustand vom Typ OR, so ist hierfür schon die direkte Call-Back-Methode callBack definiert. Ihr Aufbau ist aber analog). Die Funktion besteht aus zwei Teilen:

1. Der erste Teil realisiert die Berechnungen, die für alle Substates gelten, d.h. die für den Superstate definierten Transitionen, DO-Aktivitäten usw.

   Dieser Teil ist genauso aufgebaut wie der Kode für Zustände vom Typ BASIC.

2. Die Implementierung der Substates wird durch ein switch-Statement realisiert. Die Substates werden als Selektoren benutzt. Die Switch-Anweisung wird nur ausgeführt, wenn der Superstate nicht verlassen wird (d.h. im else-Teil der IF-Anweisung von 1.).

Beispiel

Für den Zustand Normalbetrieb des Ampelbeispiels ergibt sich die Implementierung:

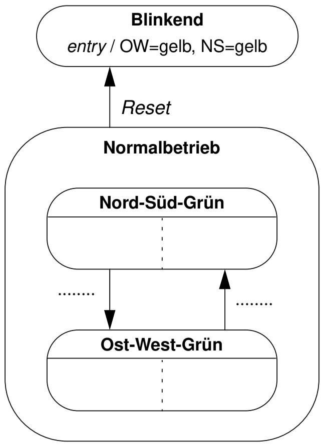


Abb. 9: Zustand Normalbetrieb der Ampel

```
void Ampel::callBackNormalbetrieb()
{
 // Superstate behandeln:
 if (eventReset) {
   // Bestimme neue Zustandskonf.
   stateAmpel=Cblinkend;
   ow=Cgelb;
   ns=Cgelb;
 } // if
 else {
   // keine DO-Aktivitäten ...
   // Substates behandeln:
   switch (stateNormalbetrieb) {
     case CnsGruen:
       callBackNSGruen ();
       break;
     case CowGruen:
               callBackOWGruen ();
       break;
     default:
       // kann nicht vorkommen!
       break;
   } // switch
 } // else
} // Ampel::callBackNormalbetrieb
```

2.4.3 Zustände vom Typ AND

Richtlinien

Für jeden Zustand vom Typ AND wird eine eigene indirekte Call-Back-Methode „callBackZustandsname" definiert (Ist der oberste Zustand vom Typ OR, so ist hierfür schon die direkte Call-Back-Methode callBack definiert. Ihr Aufbau ist aber analog). Diese besteht aus zwei Teilen:

1. Der erste Teil realisiert die Berechnungen, die für alle Substates gelten, d.h. die für den Superstate definierten Transitionen, DO-Aktivitäten usw.

   Dieser Teil ist genauso aufgebaut wie der Kode für Zustände vom Typ BASIC.

2. Für jeden parallelen Substate wird eine indirekte Call-Back-Methode definiert. Diese werden im else-Teil von 1. nach den DO-Aktivitäten für den Superstate aufgerufen.

Beispiel

Für den Zustand Nord-Süd-Grün des Ampelbeispiels sieht das so aus:

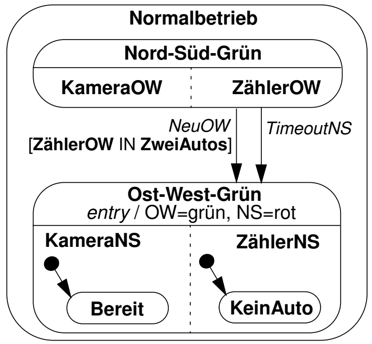


Abb. 10: Zustand Nord-Süd-Grün der Ampel

```
void Ampel::callBackNSGruen ()
{
 // Superstate behandeln:
 if (eventTimeoutNS) {
   // Bestimme neue Zustandskonfig.
   stateNormalbetrieb=CowGruen;
   stateKameraNS=CnsBereit;
   stateZaehlerNS=CnsKeinAuto;
 } // if
 else if (eventNeuOW &&
    (stateZaehlerOW==CowZweiAutos)) {
   ... // wie oben
 } // else if
 else {
   // keine DO-Aktivitaeten ...
   // parall. Substates behandeln:
   callBackKameraOW ();
   callBackZaehlerOW();
 } // else
} // Ampel::callBackNSGruen
```

2.5 Realisierung von event-getriggerten Transitionen

Problem

Das Senden und Empfangen von Events soll durch C++-Methodenaufrufe realisiert werden. Für jedes Event, das eine Klasse von außen empfängt, wird eine Methode definiert, die das Event annimmt und für seine Verarbeitung sorgt.

Richtlinie

Zur Realisierung eines Events wird ein Teil-Zustandsdiagramm in einer zusätzlichen Methode der Klasse implementiert. Der Kode ist analog zu der Methode callBack konstruiert, enthält jedoch nur die für dieses Event relevanten Zustände (d.h. die Zustände, von denen Transitionen ausgehen, die durch das Event ausgelöst werden können).

Um die oben vereinbarte Priorität der Events zu gewährleisten, muß ihre Verarbeitung direkt nach ihrem Eintreffen eingeleitet werden, um das Eintreffen weiterer Events zu verhindern (dies geht, da die Objektmethoden nicht durch andere Objekte unterbrechbar sind).

Beispiel

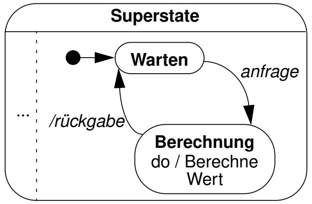


Abb. 11: Modellierung einer Abfrage im Entwurf

Die Realisierung der Abfrage sieht dann so aus:

```
rueckgtyp Klasse::anfrage()
{
  rueckgtyp wert = berechneWert();
  return (wert);
} // anfrage()
```

Als Beispiel eines Teil-Zustandsdiagramms kann man das Event *NeuOW* aus dem Ampelbeispiel umsetzen. Der Kode ist im Anhang A abgedruckt. Dabei wurde darauf verzichtet, das Teil-Zustandsdiagramm durch eine Schachtelung mehrerer Methoden nachzubilden. Die Hierarchie wird durch die Schachtelung der if- und switch-Anweisungen modelliert. Zu beachten ist, daß es nicht ausreichen würde, z.B. nur „if (stateZaehlerOW == CowKeinAuto)“ zu überprüfen, um festzustellen, ob man im Zustand KeinAuto des ZählerOW ist. Zusätzlich ist zu testen, ob der Superstate Nord-Süd-Grün und dessen Superstate Normalbetrieb aktiv sind. Diese Bedingungen werden durch den hierarchischen Aufbau garantiert.

2.6 Die Bestimmung der neuen Zustandskonfiguration

Problem

Eine Transition kann mehrere Zustandsübergänge auf mehreren Hierarchieebenen auslösen, wenn sie einen Superstate verläßt und einen anderen betritt. Deshalb müssen nicht nur die Attribute des Ausgangs- und des Zielzustands der Transition geändert werden, sondern die Attribute aller betroffenen Zustände angepaßt werden, um eine konsistente Zustandskonfiguration zu erhalten. Zu beachten ist, daß beim Betreten eines Substates auch sein Superstate betreten wird. Umgekehrt müssen beim Betreten eines Zustands vom Typ AND alle seine Substates und bei einem OR-Zustand genau ein Substate aktiv werden.

Richtlinien

Für die Bestimmung der neuen Zustandskonfiguration werden die folgenden Regeln aufgestellt:

1. Bestimme den Zustand Z der höchsten Hierarchieebene, der durch die Transition betreten wird.
2. Setze das Attribut seines Superstates (dieser ist vom Typ OR, da keine Transitionen zwischen parallelen Substates erlaubt sind!) auf den Wert für Z. Fahre mit Z bei 3. fort.
3. Ist der Zustand vom Typ OR, dann: endet die Transition auf dieser Ebene, so setze sein Attribut auf seinen Default-Zustand, sonst setze sein Attribut auf den Substate, der betreten wird. Fahre mit dem Substate analog bei 3. fort.

4. Ist der Zustand vom Typ AND, dann: setze alle seine Attribute auf ihren Default-Zustand. Fahre mit allen parallelen Substates bei 3. fort.
5. Ist der Zustand vom Typ BASIC, so ist nichts weiter zu tun.

Beispiel

Die Bestimmung der neuen Zustandskonfiguration ist in den vorangegangenen Kodebeispielen schon implizit enthalten und soll hier noch einmal anhand eines Beispiels mit einer komplexeren Transition verdeutlicht werden:

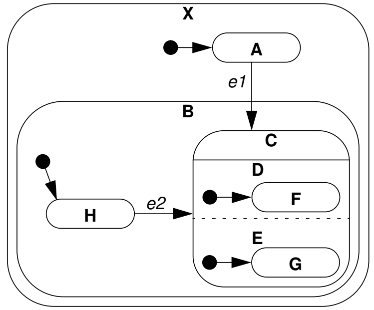


Abb. 12: Transitionen über mehrere Hierarchieebenen

Annahme:

A ist der aktuelle Zustand, Event e1 trifft ein.

1. Zustand der obersten Hierarchieebene, der betreten wird, ist B.
2. Der Superstate von B ist X; sein Attribut wird auf B gesetzt.
3. Zustand B ist vom Typ OR, darin wird C betreten.
4. Zustand C ist vom Typ AND, darin werden in den parallelen Substates D und E jeweils die Default-Zustände F und G betreten, d.h. ihre Attribute werden auf F bzw. G gesetzt.
5. Zustand F ist vom Typ BASIC.
6. Zustand G ist vom Typ BASIC.

2.7 Exit-Hierarchie

Bemerkung

Werden in den Zustandsdiagrammen keine Exit-Aktionen spezifiziert, so ist hier auch nichts zu tun und es kann mit dem nächsten Abschnitt fortgefahren werden.

Problem

Sind Exit-Aktionen definiert (insbesondere, wenn sie über mehrere Ebenen verschachtelt sind), so sollte ihre Ausführung in eine eigene Funktion verlagert werden. Das hat den Grund, daß dann bei der Implementierung der Zustände immer nur der betreffende Zustand und seine Transitionen betrachtet werden müssen. Die Abläufe bei den Exit-Hierarchien sind aber nicht direkt einem Zustand zugeordnet und sollten deshalb auch in der Implementierung von den Call-Back-Methoden getrennt werden. Außerdem wird das Durchlaufen von ähnlichen Exit-Hierarchien oft von verschiedenen Zuständen ausgelöst, was die Definition einer eigenen Funktion nahelegt. Als Alternative wären verschachtelte Aufrufe der Call-Back-Methoden denkbar (wobei die Exit-Hierarchie von innen nach außen durchlaufen wird) bzw. alle Exit-Aktionen, die eine Transition auslöst, müßten explizit bestimmt und in der Call-Back-Methode aufgeführt werden. Dies würde zu erhöhter Komplexität bzw. zu Wiederholungen führen und den Kode so fehleranfälliger und schwerer verständlich machen.

Richtlinien

Zuerst ist der Superstate der höchsten Hierarchieebene zu bestimmen, der durch die Transition verlassen wird. Dessen Exit-Methode „exitZustandsname“ wird dann aufgerufen. Die Exit-Methode ist ähnlich aufgebaut wie die Call-Back-Methoden und arbeitet auf der alten Konfiguration rekursiv wie folgt:

1. Ist der Zustand vom Typ BASIC, so führe die zugehörige Exit-Aktion aus.
2. Ist der Zustand vom Typ OR, so fahre rekursiv mit der Exit-Methode für seinen aktuellen Substate fort und führe danach die Exit-Aktion dieses Zustands aus.
3. Ist der Zustand vom Typ AND, so fahre rekursiv mit den Exit-Methoden für alle seine aktuellen Substates fort und führe danach die Exit-Aktion dieses Zustands aus.

Diese Bedingungen legen die Definition einer Exit-Methode für jeden Zustand vom Typ OR bzw. AND nahe. Die Umsetzung der OR-Zustände erfolgt wieder über die Verwendung eines Switch-Statements mit den Zuständen als Selektoren. Bei AND-Zuständen wird für jeden parallelen Substate eine weitere Exit-Methode definiert.

Beispiel

Im Ampelbeispiel sind keine Exit-Aktionen angegeben. Man kann dennoch z.B. die Exit-Methode des Zustands Ost-West-Grün definieren (die natürlich nichts tut), um den prinzipiellen Aufbau zu zeigen:

```
void Ampel::exitOWGruen ()// Beispiel für einen AND-Zustand
{
  exitKameraNS (); // parallelen Substate verlassen
  exitZaehlerNS(); // parallelen Substate verlassen
  // Exit-Aktion von Ost-West-Grün ausführen (hier keine)
} // Ampel::exitOWGruen()
void Ampel::exitKameraNS ()// Beispiel für einen OR-Zustand
{
  switch (stateKameraNS){ // Substate herausfinden und verlassen
    case CnsBereit:
      // Exit-Aktion von Bereit ausführen (hier keine)
      break;
    case CnsSchuss:
      // Exit-Aktion von Schuss ausführen (hier keine)
      break;
    default: // kann nicht erreicht werden!
      break;
  } // switch
  // Exit-Aktion von KameraNS ausführen (hier keine)
} // Ampel::exitKameraNS()
```

2.8 Entry-Hierarchie

Problem

Die hierarchische Ausführung von Entry-Aktionen kann ähnlich wie bei den Exit-Aktionen nicht direkt einem Zustand zugeordnet werden und sollte deshalb im Kode von den Call-Back-Methoden getrennt werden. Bei der Durchführung von Entry-Aktionen kann man gleichzeitig eine automatische Bestimmung von Default-Zuständen vornehmen, die sich sonst evtl. in mehreren verschiedenen Zuständen wiederholt und so zur Fehlerquelle wird.

Richtlinien

Die Entry-Methoden „entryZustandsname“ für Zustände vom Typ AND und OR sind analog zu den Exit-Methoden definiert, nur daß hier die Entry-Aktion eines Zustands vor denen der Substates ausgeführt wird und eine Behandlung der Default-Zustände erfolgt. Außerdem arbeitet diese Methode auf der neuen Zustandskonfiguration. Für jeden Zustand, bei dem die Bestimmung eines Default-Zustands nötig ist, wird ein zusätzlicher Attributwert CdefaultZustandsname eingeführt (dieser steht dann nicht für einen konkreten Zustand, sondern ist ein Platzhalter, der bei der Durchführung eines Zustandsübergangs durch einen konkreten Zustand ersetzt wird).

Die Bestimmung der neuen Zustandskonfiguration sieht nun wie folgt aus (Änderungen sind kursiv gedruckt):

1. Bestimme den Zustand Z der höchsten Hierarchieebene, der durch die Transition betreten wird.
2. Setze das Attribut seines Superstates (dieser ist vom Typ OR, da keine Transitionen zwischen parallelen Substates erlaubt sind!) auf Z. Fahre mit Z bei 3. fort.
3. Ist der Zustand vom Typ OR, dann: endet die Transition auf dieser Ebene, so setze sein Attribut auf *CdefaultZustandsname*. Sonst: setze sein Attribut auf den Substate, der betreten wird und fahre mit dem Substate analog bei 3. fort.
4. Ist der Zustand vom Typ AND, dann: *es ist nichts weiter zu tun, da die Entry-Methode die Auflösung der Default-Zustände übernimmt*.
5. Ist der Zustand vom Typ BASIC, so ist nichts weiter zu tun.

Beispiel

Am Beispiel von Ost-West-Grün sieht das so aus:

```
void Ampel::entryOWGruen ();// Beispiel für AND-Zustand
{
  // Entry-Aktionen von Ost-West-Grün ausführen:
  ow=Cgruen;
  ns=Crot;
  // parallele Substates:
  // Default-Zustaende auflösen ...
  stateKameraNS=CdefaultKamerNS; // OR-Zustand -> Cdefault setzen
  stateZaehlerNS=CdefaultZaehlerNS; // OR-Zust -> Cdefault setzen
  // ... und rekursiv fortfahren:
  entryKameraNS ();
  entryZaehlerNS ();
} // Ampel::entryOWGruen
void Ampel::entryKameraNS ()// Beispiel für OR-Zustand
{
  // Entry-Aktionen von KameraNS ausführen (hier keine)
  // Default-Zustand auflösen:
  if (stateKameraNS==CdefaultKameraNS) {
```

```
      stateKameraNS=CnsBereit;
    } // if
    // Behandeln der Substates
    switch (stateKameraNS) {
      case CnsBereit:
        // Entry-Aktion von Bereit ausführen (hier keine)
        break;
      case CnsSchuss:
        // Entry-Aktion von Schuss ausführen (hier keine)
        break;
      default:
        // wird nicht erreicht!
        break;
    } // switch
  } // Ampel::entryKameraNS
```

2.9 Interne Aktionen

Problem

Interne Aktionen (s. Abb. 13) reagieren auf Events, ohne einen Zustandsübergang auszulösen. (Prinzipiell können interne Aktionen als parallele Substates des Zustandes, in dem sie definiert sind, angesehen werden. Diese Sichtweise ist aber zu kompliziert umzusetzen und bei dem einfachen Aufbau der Substates (ein Zustand mit einer selbstbezüglichen event-getriggerten Transition) auch für die Umsetzung nicht nötig.)

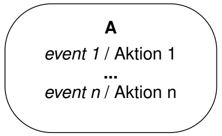


Abb. 13: Zustand mit internen Aktionen

Richtlinie

Interne Aktionen werden wie event-getriggerte Transitionen implementiert. Es gelten die in Abschnitt 2.5.2 gemachten Empfehlungen. Bei einer Umsetzung in der Call-Back-Methode muß das entsprechende Event-Attribut vor der Behandlung der Substates im else-Zweig abgepüft werden.

Kapitel 3

Spezialfälle und Erweiterungen

Die im vorigen Abschnitt vorgestellten Richtlinien decken einen großen Teil der Notation von Zustandsdiagrammen ab. Die im folgenden behandelten Punkte führen z.T. zu komplexeren Implementierungen oder werden nicht häufig benutzt, so daß von ihrer Darstellung im vorigen Kapitel abgesehen wurde. Um die Allgemeinheit der Richtlinien zu zeigen, werden hier auch Erweiterungen vorgeschlagen, die z.T. auch eingangs gemachte Einschränkungen überflüssig machen, allerdings um den Preis einer komplexeren Umsetzung. Man sollte daher immer abwägen, ob die Verwendung in einem Zustandsdiagramm nötig ist.

3.1 Automatische Transitionen (Lambda-Transitionen)

Problem

Lambda-Transitionen sind Transitionen, die sofort nach Abschluß der mit einem Zustand assoziierten Aktivität feuern.

Richtlinie

Die Implementierung geschieht einfach, indem nach dem DO-Teil (d.h. am Ende des else-Zweigs) des Kodes, der den Zustand umsetzt, die neue Konfiguration bestimmt wird und eine Exit-Aktion-Entry-Sequenz eingeleitet wird.

Beispiel

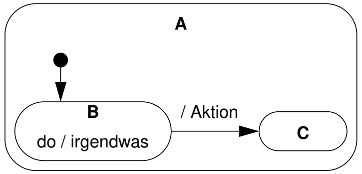


Abb. 14: Zustand mit einer Lambda-Transition

```
void callBackA ()
{
  switch stateA {
    case Cb:
      // DO-Aktivität ausführen
      // Keine Exit-Aktionen.
      Aktion ();
      // Neuen Zustand bestimmen:
      stateA = Cc;
      // Keine Entry-Aktionen.
      break;
  ...
} // callBackA ()
```

3.2 Aufspalten der Kontrolle

Problem

Beim Aufspalten der Kontrolle (siehe Abb. 21 in Anhang A) in mehrere parallele Substates werden nicht notwendigerweise die Default-Zustände der parallelen Substates betreten. Deshalb muß man die oben gemachte Voraussetzung, daß keine Transitionen in parallele Substates führen, fallen lassen.

Richtlinien

Die vereinfachte Bestimmung der Zustandskonfiguration aus 2.8 muß modifiziert werden, da bei AND-Zuständen nicht mehr einfach die Default-Zustände gesetzt werden dürfen. Die erweiterten Regeln sehen wie folgt aus (Änderungen zu 2.8 sind kursiv gedruckt):

1. Bestimme den Zustand Z der höchsten Hierarchieebene, der durch die Transition betreten wird.
2. Setze das Attribut seines Superstates (dieser ist vom Typ OR, da keine Transitionen zwischen parallelen Substates erlaubt sind!) auf Z. Fahre mit Z bei 3. fort.
3. Ist der Zustand vom Typ OR, dann: endet die Transition auf dieser Ebene, so setze sein Attribut auf CdefaultZustandsname, sonst setze sein Attribut auf den Substate, der betreten wird. Fahre mit dem Substate analog bei 3. fort.
4. Ist der Zustand vom Typ AND, dann: *endet die Transition auf dieser Ebene, so setze die Attribute aller Substates auf CdefaultZustandsname; sonst: setze die Attribute aller Substates, in denen der Default-Zustand betreten wird, auf CdefaultZustandsname; die anderen Attribute werden auf den entsprechenden Zustand gesetzt und rekursiv weiterverarbeitet.*
5. Ist der Zustand vom Typ BASIC, so ist nichts weiter zu tun.

Man sieht, daß die Entry-Methoden der AND-Zustände komplizierter werden. Werden immer nur die Default-Zustände betreten, so kann die Entry-Methode diese automatisch bestimmen, wie in Abschnitt 2.8 gezeigt. Wird von dieser Voraussetzung abgewichen, so muß ein Teil der Arbeit im Kode des

Zustands, der die Transition auslöst, erledigt werden. Die Entry-Methode kann dann nur noch die verbleibenden Default-Zustände bestimmen.

Abb. 15 gibt dafür ein Beispiel. In den Substates Par1 und Par2 werden die Zustände Z11 bzw. Z21 direkt durch die Transition ausgewählt. Zum Substate Par3 führt hingegen kein Transitionspfeil, d.h. hier wird der Default-Zustand Z31 betreten.

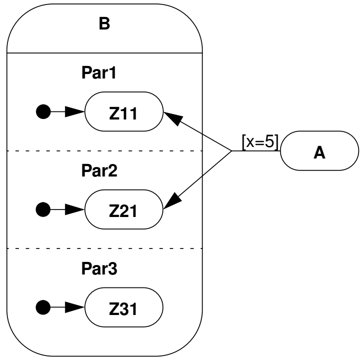


Abb. 15: Transitionen in einen AND-Zustand

```
// Kode für Zustand A:
if (x==5) {
  CstateModel = Cb;
  CbPar1 = Cz11;
  CbPar2 = Cz21;
  CbPar3 = CdefaultPar3;
  entryB ();
} .....

// Kode für Entry-Methode von B:
void entryB () {
  // Defaults der Substates best.:
  if (CbPar1 == CdefaultPar1) {
    CbPar1 = Cz11; } // if
  if (CbPar2 == CdefaultPar2) {
    CbPar2 = Cz21; } // if
  if (CbPar3 == CdefaultPar3) {
    CbPar3 = Cz31; } // if
  // Entrys der Substates:
  entryPar1 ();
  entryPar2 ();
  entryPar3 ();
} // entryB
```

3.3 Synchronisation paralleler Aktivitäten

Problem

Eine Synchronisation der parallelen Aktivitäten (siehe Abb. 21 in Anhang A) ist schwieriger umzusetzen, da evtl. auf das Eintreffen verschiedener Events oder das Eintreten verschiedener Bedingungen gewartet werden muß.

Empfehlung

Als Lösung wird vorgeschlagen, jedem parallelen Substate, der auf die Synchronisation mit weiteren Substates wartet, explizit einen weiteren Attributwert „CwaitForSync“ hinzuzufügen. Der Zustand des Substates wird nach Eintreten der Synchronisationsbedingung auf CwaitForSync gesetzt. Nun kann in dem Superstate, der diese Substates beinhaltet, abgeprüft werden, ob die Synchronisationsbedingung für alle relevanten Substates gilt und ein entsprechender Zustandsübergang ausgelöst werden.

In dem Spezialfall, daß nur eine Transition aus einem Substate eines AND-Zustands zeigt, kann der Übergang wie bei gewöhnlichen Transitionen erfolgen.

3.4 Transitionen zwischen parallelen Substates

Problem

Transitionen zwischen parallelen Substates wie in Abb. 16 (a) sind in Harelschen Statecharts erlaubt, nicht aber in der UML/OMT-Notation. Sie können jedoch nützlich sein, um Synchronisationen zu realisieren. Solche Transitionen können in dieser Form auch nicht mit den angegebenen Richtlinien behandelt werden. Dies liegt daran, daß konkurrente Zuweisungen (durch die Call-Back-Methoden mehrerer paralleler Substates) an Zustandsvariablen gemacht werden, die zu illegalen Konfigurationen führen können.

Richtlinie

Es wird die äquivalente Notation aus Abb. 16 (b) verwendet. So gibt es keine Probleme mehr, da die parallelen Substates jetzt nur noch durch IN gekoppelt sind, das auf der gepufferten alten Zustandskonfiguration aufbaut. Da aber der IN-Übergang erst im nächsten Schritt gemacht würde, muß mit Microsteps (s. Abschnitt 3.7) gearbeitet werden.

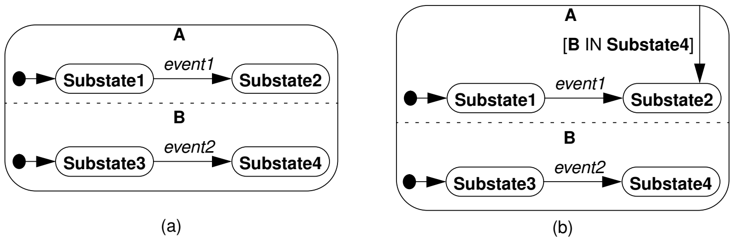


Abb. 16: Transitionen zwischen parallelen Substates

3.5 Termination

Problem

In den obigen Richtlinien wurde die Termination noch nicht betrachtet. Es kann unterschieden werden zwischen der Termination eines Teils eines Zustandsdiagramms und die des ganzen Zustandsdiagramms beim Eintreten in einen Endzustand:

1. Bei der Termination des gesamten Zustandsdiagramms muß das Objekt gelöscht werden. Dies soll normalerweise durch das erzeugende Objekt erfolgen, das deshalb über die Termination unterrichtet werden muß. Die Richtlinien beschäftigen sich eigentlich nicht mit dem Erzeugung und dem Löschen von Objekten. Beim Eintreten eines Zustandsdiagramms in einen Endzustand muß jedoch eine Koordination zwischen erzeugendem Objekt (das kann auch das main-Programm sein) und dem terminierenden Objekt erfolgen.
2. Die Termination eines Teil-Zustandsdiagramms hat eine andere Semantik. Die Situation kann bei der Verfeinerung eines Zustands auftreten. Abb. 17 (a) zeigt einen Zustand mit einer DO-Aktivität. In Abb. 17 (b) ist die Aktivität in ein verschachteltes Zustandsdiagramm verfeinert worden. Die Beendigung der Aktivität wird nun durch die Termination des verfeinerten Zustandsdiagramms dargestellt. Im Falle der Termination muß der Superstate des beendeten Zustands über die Termantion unterrichtet werden.

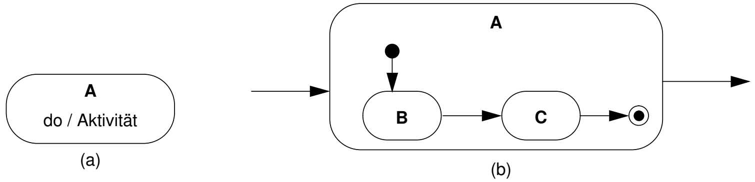


Abb. 17: Termination bei der Verfeinerung von Zuständen

Empfehlung

Terminiert ein Objekt O einer Klasse K, so sollte in der Methode callBack bzw. den Methoden, die Events empfangen, dafür gesorgt werden, daß keine weitere Verarbeitung erfolgt. Außerdem sollte auf Anfrage des erzeugenden Objekts O' der aktuelle Zustand (terminiert oder nicht) gemeldet werden. Dies kann z.B. durch eine Methode

```
bool K::alive();
```

geschehen, die entsprechend wahr oder falsch zurückgibt. O' muß dann prüfen, ob eine Termination erfolgt ist und evtl. das Objekt O vernichten. Außerdem wird O mit Hilfe der Methode „unregisterCallBack ()" der Klasse ScheduledFct beim Laufzeitsystem abgemeldet.

Bei der Termination eines Teils des Zustandsdiagramms kann anders vorgegangen werden. Für diesen Fall werden die folgenden Empfehlungen aufgestellt:

1. Ein Zustand, der in einen Endzustand eintreten kann (oben Zustand A), erhält einen weiteren Attributwert `Cterminated`<u>`Zustandsname`</u>. Im Beispiel sieht das so aus:

   ```
   enum {Cb, Cc, CterminatedA} stateA;
   ```

2. Terminiert ein Teil des Zustandsdiagramms, so wird dessen Attribut von seiner Call-Back-Methode auf `Cterminated`<u>`Zustandsname`</u> gesetzt.

Der Superstate kann nun durch Abfragen der Attribute seiner Substates nach dem Aufruf ihrer Call-Back-Methoden herausfinden, ob eine Termination vorliegt und entsprechende Transitionen ausführen.

3.6 Interne Events

Problem

Läßt man interne Events zu, so können während der Bearbeitung eines Events weitere (interne) Events entstehen. Diese Events können während eines Schrittes Transitionen auslösen, die zu Beginn des Schrittes nicht ausführbar waren. Die gleichen Effekte treten auf, wenn eine Variable von mehreren parallelen Substates modifiziert wird.

Empfehlung

Auf interne Events ist möglichst zu verzichten. Erlaubt sind nur Spezialfälle wie beispielsweise in Abb. 18. Werden interne Events implementiert, so müssen diese Events gespeichert (z.B. in Form von boolschen Variablen oder Structs, falls es Attribute gibt) und im nächsten Schritt bearbeitet werden, d.h. es müssen zwei Zustände der Events gespeichert werden: zum einen die Events, die im aktuellen Schritt gültig sind und zum anderen die Events, die durch Aktionen im aktuellen Schritt ausgelöst werden. Diese Event-Konfigurationen müssen initialisiert und aktualisiert werden. Dies geschieht analog zu den Zustandskonfigurationen in der Array-Variante. Interne Events führen auch zur Problematik der Microsteps.

3.7 Microsteps

Problem

In Ausnahmefällen kann es möglich sein, daß sich mit den vorgestellten Richtlinien sinnvolle Einsatzmöglichkeiten von Zustandsdiagrammen nicht umsetzen lassen. Abb. 18 gibt dazu ein Beispiel. Der Zustand des Zählers ist erst dann wieder konsistent, wenn alle durch das externe Event inc0 ausgelösten Zustandsübergänge (und damit die Verarbeitung der internen Events) abgeschlossen sind.

Empfehlung

Microsteps werden nur dann benötigt, wenn gefordert wird, daß in einem konsistenten Zustand keine Transitionen ausführbar sind, d.h. Zustandsübergänge nur als Reaktion auf zukünftige externe Ereignisse erfolgen können.

Richtlinie

Verwendet man die Array-Variante der Zustandskonfiguration und die Realisierung von internen Events aus dem letzten Abschnitt, dann lassen sich Microsteps wie folgt realisieren:

Um zu erkennen, ob kein Zustandsübergang mehr erfolgen kann, durchläuft man die callBack-Methode so oft, bis keine Veränderung in der Konfiguration mehr auftritt und alle internen Events verarbeitet sind. Dies kann bei der Aktualisierung der Konfiguration am Ende von callBack geprüft werden.

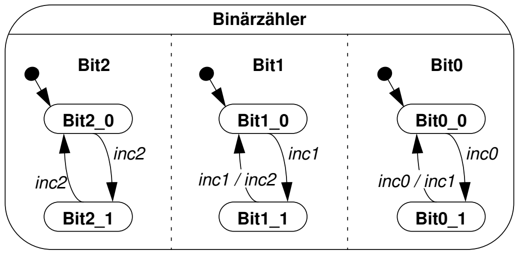


Abb. 18: Interne Events und Microsteps bei einem Binärzähler (s. [Day93])

Anhang A

Kode für die Implementation des Events NeuOW als Teil-Zustandsdiagramm

```
void Ampel::neuOW()
{
  switch (stateAmpel) {
    case CnormalBetrieb:
      switch (stateNormalbetrieb) {
        case CnsGruen:
          // Prüfen, ob IN-Transition ausführbar ist:
          if (stateZaehlerOW == CowZweiAutos) {
             stateNormalbetrieb=CowGruen;
            stateKameraNS=CnsKeinAuto;
             stateZaehlerNS=CnsBereit;
             // Keine Exits spezifiziert.
             // Keine Aktionen spezifiziert.
             // Entry-Aktionen:
            ow=Cgruen;
            ns=Crot;
          } // if
          // Keine weiteren Transitionen von NSGruen zu prüfen, also:
          else {// Hier sind die relevanten Substates zu überprüfen:
             // KameraOW ist nicht relevant, sonst
              // stünde hier der Kode ...
             // ZaehlerOW ist relevanter Substate:
            switch (stateZaehlerOW) {
          case CowKeinAuto:
               stateZaehlerOW=CowEinAuto;
               // hier ist nichts weiter zu beachten.
               break;
          case CowEinAuto:
               stateZaehlerOW=CowZweiAutos;
               // hier ist nichts weiter zu beachten.
               break;
          default: // keine anderen Zustände von ZaehlerOW relevant!
                 break;
            } // switch (stateZaehlerOW)
          } // else
        default: // keine anderen Zustände von
```

```
                    // Normalbetrieb relevant!
           break;
       } // switch (stateNormalbetrieb)
    default: // keine anderen Zustände von Ampel relevant!
      break;
  } // switch (stateAmpel)
} // Ampel::neuOW()
```


Anhang B

Kode für das Ampel-Beispiel

Bemerkungen

Die Implementierung verwendet immer die einfachste Variante der Richtlinien. Da keine Exit-Aktionen definiert sind und die Entry-Aktionen nur eine Hierarchieebene umfassen, wurde auf eine Realisierung von Entry- und Exit-Hierarchien verzichtet. Trotz der Verwendung von IN braucht man hier nicht mit zwei Konfigurationen zu arbeiten, da die IN-Transitionen Priorität haben vor evtl. Zustandsübergängen der abgefragten Zähler-Substates. Auch die Auflösung von Default-Zuständen ist für dieses Beispiel zu viel Aufwand bei relativ geringem Gewinn an Übersichtlichkeit der Call-Back-Methoden. Aus Gründen der kompakten Darstellung sind keine Header angegeben und es wurde keine Unterteilung in verschiedene Files vorgenommen.

Die verwendeten Methoden nsBlinken bzw. owBlinken sind ohne Implementierung angegeben. Eine Umsetzung kann z.B. durch Timeouts erfolgen.

```
class Ampel : public ScheduledFct
{
  private:
         // Variablen für die Zustandskonfiguration:
         // Root-Zustand Ampel ist vom Typ OR -> 1 Attribut
         enum {Cblinkend, CnormalBetrieb} stateAmpel;

         // Normalbetrieb ist vom Typ OR -> 1 Attribut
         enum {CowGruen, CnsGruen} stateNormalbetrieb;

         // OWGruen ist vom Typ AND -> 2 Attribute
         enum {CnsBereit, CnsSchuss} stateKameraNS ;
         enum {CnsKeinAuto, CnsEinAuto, CnsZweiAutos} stateZaehlerNS;

         // NSGruen ist vom Typ AND -> 2 Attribute
         enum {CowBereit, CowSchuss} stateKameraOW;
         enum {CowKeinAuto, CowEinAuto, CowZweiAutos} stateZaehlerOW;

         // Variablen aus dem Zustandsdiagramm
         enum {Crot, Cgelb, Cgruen} ow, ns;
```

```
        // Event-Variablen
        bool eventNeuOW, eventNeuNS, eventAutoOW, eventAutoNS;
        bool eventInit, eventReset, eventTimeoutOW, eventTimeoutNS;

  private:
        // Restliche Call-Back-Methoden sind private!
        void callBackNormalbetrieb ();
        void callBackNSGruen ();
        void callBackKameraOW();
        void callBackZaehlerOW();
        void callBackOWGruen ();
        void callBackKameraNS();
        void callBackZaehlerNS();

  public:
       // Konstruktor
        Ampel ();

        // Standard-Destruktor ist korrekt
        ~Ampel ();

       // Die Call-Back-Methode realisiert das Zustandsdiagramm
       void callBack (void);

       // Event-Methoden
        void neuOW ();
        void neuNS ();
        void autoOW ();
        void autoNS ();
        void init ();
        void reset ();
        void timeoutOW ();
        void timeoutNS ();
}; // class Ampel

Ampel::Ampel ()
{
  // Variablen initialisieren:
  eventNeuOW = false;
  eventNeuNS = false;
  eventAutoNS = false;
  eventAutoOW = false;
  eventInit = false;
  eventReset = false;
  eventTimeoutOW = false;
  eventTimeoutNS = false;

  // Initialen Zustand herstellen:
  stateAmpel = Cblinkend;
  ow = Cgelb;
  ns = Cgelb;

  // Anmelden beim Laufzeitsystem
  registerCallBack ();
```



```
}; // Ampel::Ampel

// Call-Back-Methoden
void Ampel::callBack ()
{
  // Ampel ist Root-Zustand, kann deshalb nicht verlassen werden!
  // Es gibt auch keine Aktivitäten oder interne Aktionen.
  // Deshalb steht hier kein Kode für den Superstate Ampel.

 // Kode für Substates:
  switch (stateAmpel) {

    case Cblinkend:
      if (eventInit) {
        // keine Exit-Aktionen ...
        // neue Konfig. berechnen:
        stateAmpel = CnormalBetrieb;
        stateNormalbetrieb=CnsGruen;
        stateKameraOW=CowBereit;
        stateZaehlerOW=CowKeinAuto;

        // keine Aktionen ...

        // Entry-Aktionen ausführen:
        ns=Cgruen;
        ow=Crot;
      } // if
      else {
        // DO-Aktivitäten (Blinken der Ampeln):
        nsBlinken ();
        owBlinken ()
      } // else
      break;
    case CnormalBetrieb:
      callBackNormalbetrieb ();
      break;
    default:
      break;
  } // switch
} // Ampel::callBack

void Ampel::callBackNormalbetrieb()
{
  // Superstate behandeln:
  if (eventReset) {
    // Neuen Zustand berechnen
    stateAmpel=Cblinkend;

    // Entry-Aktionen ausführen
    ow=Cgelb;
    ns=Cgelb;
  } // if
  else {
    // keine DO-Aktivitäten ...
```

```
    // Substates behandeln:
    switch (stateNormalbetrieb) {
      case CnsGruen:
        callBackNSGruen ();
        break;
      case CowGruen:
        callBackOWGruen ();
        break;
      default: // sollte nicht vorkommen!
        break;
    } // switch
  } // else
} // Ampel::callBackNormalbetrieb

void Ampel::callBackNSGruen ()
{
  // Superstate behandeln:
  // Bem.: die beiden Bedingungen, die eine Transition in den Zustand
  // OWGrün auslösen, werden hier gemeinsam behandelt.
  if (eventTimeoutNS || (eventNeuOW &&
        (stateZaehlerOW==CowZweiAutos))) {
    // Neuen Zustand berechnen
    stateNormalbetrieb=CowGruen;
    stateKameraNS=CnsBereit;
    stateZaehlerNS=CnsKeinAuto;

    // Entry-Aktionen ausführen
    ow = Cgruen;
    ns = Crot;
  } // if
  else
  {
    // keine DO-Aktivitäten ...
    // parallele Substates behandeln:
    callBackKameraOW ();
    callBackZaehlerOW();
  } // else
} // Ampel::callBackNSGruen

void Ampel::callBackKameraOW ()
{
  switch (stateKameraOW) {
    case CowBereit:
      if (eventAutoOW) {
       stateKameraOW = CowSchuss;
      } // if
      break;
    case CowSchuss:
      // Eine Realisierung der Kamera ist hier nicht vorgesehen.
      // Hier könnte z.B. KameraOW.schuss() stehen, wenn die
      // Kamera als Objekt einer entspr. Klasse realisiert ist.
      stateKameraOW = CowBereit; // Lambda-Transition
      break;
    default:
```



```
      break;
  } // switch
} // Ampel::callBackKameraOW

void Ampel::callBackZaehlerOW ()
{
  switch (stateZaehlerOW) {
    case CowKeinAuto:
      if (eventNeuOW) {
        stateZaehlerOW = CowEinAuto;
      }  // if
      break;
    case CowEinAuto:
      if (eventNeuOW) {
        stateZaehlerOW = CowZweiAutos;
      }  // if
      break;
    case CowZweiAutos:
      break;
    default:
      break;
  } // switch
} // Ampel::callBackZaehlerOW

void Ampel::callBackOWGruen () // Kode analog zu ...NS
{
  // Superstate behandeln:
  // Bem.: die beiden Bedingungen, die eine Transition in den Zustand
  // NSGrün auslösen, werden hier gemeinsam behandelt.
  if (eventTimeoutOW || (eventNeuNS &&
        (stateZaehlerNS==CnsZweiAutos))) {
    // Neuen Zustand berechnen
    stateNormalbetrieb=CnsGruen;
    stateKameraOW=CowBereit;
    stateZaehlerOW=CowKeinAuto;

    // Entry-Aktionen ausführen
    ns = Cgruen;
    ow = Crot;
  } // if
else
  {
    // keine DO-Aktivitäten ...
    // parallele Substates behandeln:
    callBackKameraNS ();
    callBackZaehlerNS();
  } // else
} // Ampel::callBackNSGruen

void Ampel::callBackKameraNS () // Kode analog zu ...OW
{
  switch (stateKameraNS) {
    case CnsBereit:
      if (eventAutoNS) {
```

```
        stateKameraNS = CnsSchuss;
      } // if
      break;
    case CnsSchuss:
      // Eine Realisierung der Kamera ist hier nicht vorgesehen.
      // Hier könnte z.B. KameraNS.schuss() stehen, wenn die
      // Kamera als Objekt einer entspr. Klasse realisiert ist.
      stateKameraNS = CnsBereit; // Lambda-Transition
      break;
    default:
      break;
  } // switch
} // Ampel::callBackKameraNS

void Ampel::callBackZaehlerNS () // Kode analog zu ...OW
{
  switch (stateZaehlerNS) {
    case CnsKeinAuto:
      if (eventNeuNS) {
        stateZaehlerNS = CnsEinAuto;
      }  // if
      break;
    case CnsEinAuto:
      if (eventNeuNS) {
        stateZaehlerNS = CnsZweiAutos;
      }  // if
      break;
    case CnsZweiAutos:
      break;
    default:
      break;
  } // switch
} // Ampel::callBackZaehlerNS

// Event-Methoden:
void Ampel::neuOW()
{
  eventNeuOW = true;
  callBack ();
  eventNeuOW = false;
} // Ampel::neuOW

void Ampel::neuNS()
{
  eventNeuNS = true;
  callBack ();
  eventNeuNS = false;
} // Ampel::neuNS

void Ampel::autoOW()
{
  eventAutoOW = true;
  callBack ();
  eventAutoOW = false;
```



```
} // Ampel::autoOW

void Ampel::autoNS()
{
  eventAutoNS = true;
  callBack ();
  eventAutoNS = false;
} // Ampel::autoNS

void Ampel::init()
{
  eventInit = true;
  callBack ();
  eventInit = false;
} // Ampel::init

void Ampel::reset()
{
  eventReset = true;
  callBack ();
  eventReset = false;
} // Ampel::reset

void Ampel::timeoutOW()
{
  eventTimeoutOW = true;
  callBack ();
  eventTimeoutOW = false;
} // Ampel::timeoutOW

void Ampel::timeoutNS()
{
  eventTimeoutNS = true;
  callBack ();
  eventTimeoutNS = false;
} // Ampel::timeoutNS
```