1. Tycoon Statecharts zur Modellierung dynamischen Systemverhaltens

Dynamisches Systemverhalten bezeichnet, wie im Abschnitt **??** dargestellt, die Gesamtheit der einnehmbaren Zustände und der möglichen Zustandsübergänge eines Systems unter Berücksichtigung zustandsübergansauslösender Stimuli. Bei der Erstellung eines Spezifikationsmodells des Systemverhaltens eines zu entwickelnden Anwendungssystems werden in vielen Software Entwicklungsmethoden, wie z.B. in der Strukturierten Analyse oder der *Structured Systems Analysis and Design Method* (SSADM) endliche Automaten zur Spezifikation des Systemverhaltens verwendet [Fricke 95].

Dabei stellt ein endlicher Automat ein mathematisches Modell eines Systems mit diskreten Ein- und Ausgaben dar. Zu jedem Zeitpunkt befindet sich das System in einem Zustand aus einer endlichen Menge von möglichen Zuständen, den Konfigurationen [Hopcroft, Ullman 90]. Zum Entwurf des Verhaltens von Systemen mit endlichen Zustandsmengen und diskreten Zustandsübergängen bietet die Theorie der endlichen Automaten ein adäquates Werkzeug zur Systemspezifikation.

Allerdings zeigt sich, daß bei großen komplexen Systemen leicht die Grenzen der Modellierungsmächtigkeit endlicher Automaten erreicht werden, da sie in ihrem Umfang rasch an Unhandlichkeit und Unübersichtlichkeit zunehmen. Eine Ergänzung um Zustandstabellen oder -matrizen kann zwar ihre Aussagekräftigkeit ergänzen, führt aber zu einer noch undurchschaubareren Darstellung des aus endlichen Automaten und Zustandstabellen oder -matrizen bestehenden Verhaltensmodells des Systems. Außerdem wird so der Vorteil einer ausschließlich grafischen Repräsentation des Verhaltens eines Systems mit endlicher Zustandsmenge aufgegeben.

Eine Erweiterung des Modells endlicher Automaten stellt die in diesem Kapitel dargestellte und im Rahmen dieser Arbeit innerhalb des Tycoon Systems exemplarisch in einer textuellen Repräsentation implementierte Spezifikationssprache der Statecharts dar [Harel 87]. Schwerpunktmäßig werden Statecharts wie im Abschnitt **??** gezeigt zur grafischen Modellierung und Spezifikation des Verhaltens ereignisgesteuerter Systeme eingesetzt [Huizing,

de Roever 91], die in einer fortwährenden Interaktion mit ihrer Systemumgebung stehen. Solche reaktiven Systeme sind z.B. Steuerungseinrichtungen für Flugzeuge oder Kraftwerke oder das im Abschnitt **??** eingeführte Beispiel der Fernsehsteuerung. Für die Modellierung rein statischer Systeme, wie z.B. eines *Repositorys*, sind Statecharts nicht geeignet.

Ausgehend von der Statechart Originalnotation aus [Harel 87] sind sowohl die Statechart Syntax als auch die Semantik in vielen Forschungsarbeiten verschieden dargestellt und erweitert worden. In [v.d. Beeck 94] ist ein Vergleich einiger unterschiedlicher Statechart Varianten enthalten und bezüglich ihrer Inhalte verglichen und bewertet.

Grundlage der in diesem Kapitel vorgestellten Tycoon Statechart Syntax innerhalb einer Tycoon Umgebung für eine textuelle statechartbasierte Modellierung dynamischen Systemerhaltens als Basis für die im Abschnitt **??** dargestellte Tycoon Validierungsumgebung sind die Statechartelemente des dynamischen Modells der im Abschnitt **??** dargestellten Software Entwicklungsmethode OMT. Allerdings handelt es sich bei den von der OMT Methode im Rahmen des dynamischen Modells verwendeten Statechartelementen um eine Teilmenge der in [v.d. Beeck 94] verglichenen Syntaxelemente der zahlreichen Statechart Varianten.

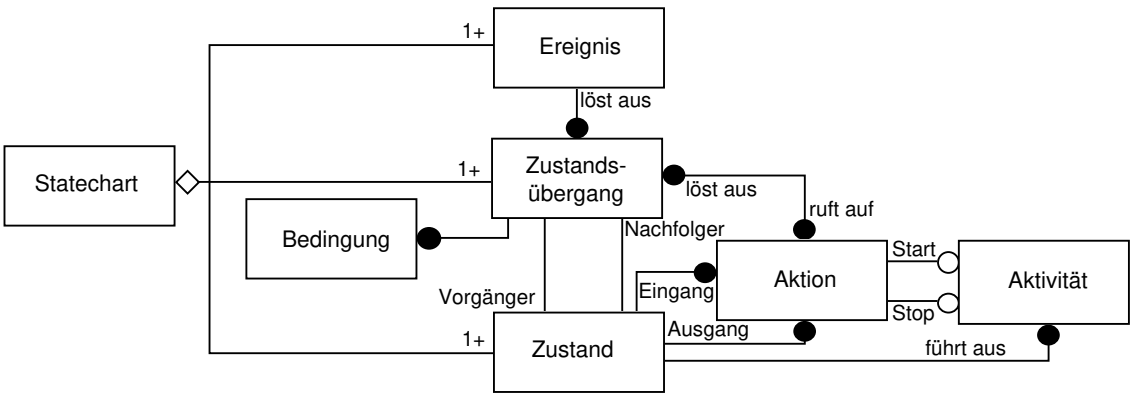

Abbildung 1.1: Statechart Metamodell

Diese Teilmenge wird von mir im Rahmen des exemplarischen Aufbaus der statechartbasierten Tycoon Modellierungsumgebung zur Modellierung dynamischen Systemverhaltens auf die in der Abbildung 1.1 in der Notation des OMT Objektmodells [Rumbaugh et al. 91] dargestellten Statechart Syntaxelemente weiter eingeschränkt. Eine Erläuterung der Abbildung 1.1 erfolgt im anschließenden Abschnitt.

1.1 Statecharts als strukturierte endliche Automaten

Zustände und Zustandsübergänge stellen die Grundelemente endlicher Automaten dar. Zustandsübergange, auch als Transitionen bezeichnet, arbeiten auf einer endlichen Menge von Eingabesysmbolen aus einem Eingabealphabet und überführen einen Zustand in einen Folgezustand [Hopcroft, Ullman 90]. Formal sind Zustandsübergänge als eine Übergangsfunktion von Quellzustand und Eingabezeichen in einen Zielzustand definiert. Überführt eine Folge von Eingabezeichen den endlichen Automaten von einem Startzustand in einen Endzustand, so wird das Eingabewort vom Automaten akzeptiert. Es gehört somit zur Sprache dieses endlichen Automatens. Neben diesen akzeptierenden endlichen Automaten gibt es endliche Automaten, deren Zustände (Moore Maschine) oder Zustandsübergänge (Mealy Maschine) mit der Ausgabe von Zeichen assoziiert sind.

Bildlich dargestellt entspricht ein endlicher Automat einem gerichteten Graphen, bei dem die Knoten den Zuständen und die gerichteten Kanten (Pfeile) zwischen den Knoten den Transitionen des endlichen Automaten entsprechen. Die Eingabesymbole werden an die Pfeile notiert. Bei einer Moore Maschine werden zusätzlich noch die Ausgabezeichen an die entsprechenden Zustände geschrieben, mit Zustandsübergängen assoziierte Ausgabezeichen werden bei der Mealy Maschine an der entsprechenden Transition vom Eingabezeichen durch einen Schrägstrich getrennt, dargestellt.

Statecharts erweitern das Modell endlicher Automaten um die Eigenschaften der Dekomposition von Zuständen und der Orthogonalität von Zuständen und Zustandsübergängen [Glinz 95]. Bei ihrer Darstellung werden Eingabezeichen als Ereignisse bezeichnet und Ausgabezeichen als Aktionen. Sowohl Ereignisse als auch Aktionen sind mit Transitionen verknüpft. Allerdings können auch wie im Abschnitt 1.3.2 dargestellt Aktionen mit Zuständen assoziiert werden. Ereignisse lösen Zustandsübergänge aus, zusätzlich können Begingungen Zustandsübergänge einschränken.

Wie in der Abbildung 1.1 wiedergegeben, betrachte ich im Rahmen der exemplarischen Implementierung der Tycoon Umgebung *StatechatEnv* zur Modellierung dynamischen Systemverhaltens in einer textuellen TL Statechart Repräsentation die folgenden Statechart Syntaxelemente:

- Zustand
- Zustandsübergang zwischen Zuständen
- Ereignis, sowohl externes Ereignis als auch internes Ereignis mit Triggerwirkung an Zustandsübergängen
- Aktion als „Generator“ interner Ereignisse
- Bedingung zur temporären Einschränkung von Zustandsübergängen
- Aktivität als zeitandauernde Handlung

Ein Statechart aggregiert Zustände, Zustandsübergänge und Ereignisse, wobei jeder Zustandübergang einen Vorgänger- und Nachfolgerzustand besitzt und ein Ereignis einen oder mehrere Zustandsübergänge auslösen kann, falls eine eventuell für das Stattfinden eines Zustandsübergangs definierte Bedingung gültig ist. Ein Zustandsübergang kann eine oder mehrere Aktionen umfassen, die in der Form von intern generierten Ereignissen wiederum einen oder mehrere Zustandsübergänge auslösen können. Entsprechend der Moore Maschine bei endlichen Automaten kann ein Zustand mit einer oder mehreren Aktionen assoziiert werden. Eine oder mehrere in einem Zustand definierte Eingangsaktionen können einen oder mehrere Zustandsübergänge beim Übergang in diesen Zustand auslösen, entsprechendes gilt für Ausgangsaktivitäten beim Übergang in den Folgezustand. Laut Synchronitätshypothese [Huizing et al. 88] treten Ereignisse und Zustandsübergänge zeitverzugslos auf. Eine Aktivität hingegen stellt eine zeitandauernde Handlung dar, die durch eine initiierende Startaktion und eine die Aktivität terminierende Stopaktion kontrolliert wird oder solange durchgeführt wird, wie die aktuelle Konfiguration des Systems, wie im Abschnitt 1.4.1 erläutert, den mit der Aktivität assoziierten Zustand umfaßt.

David Harel definiert Statecharts in [Harel 87] mit der Formel:

$$statecharts = state\ diagrams + depth + orthogonality + broadcast\text{-}communication$$

Tiefe (*depth*) entsteht innerhalb von Zustandsdiagrammen durch Zusammenfassung oder Verfeinerung von Zuständen. Diese Strukturierung und Hierarchisierung von Zuständen erlaubt es, selbst komplexe und von Größe her umfangreiche Zustandsdiagramme übersichtlich und für die Anwender und Entwickler durchschaubar darzustellen. Im Gegensatz zu endlichen Automaten können somit eine Vielzahl von Zuständen und Zustandsübergängen innerhalb eines von der Aussage her mit einem endlichen Automaten vergleichbaren Statecharts wegfallen. Ein modelliertes System kann beliebig verfeinert auf unterschiedlichen Abstraktionsebenen in Teildiagrammen des Gesamtsystems dargestellt werden, was dazu führt, daß die Teildiagramme für sich selbst betrachtet übersichtlich bleiben. Allerdings kann durch die Verwendung unterschiedlicher Abstraktionsebenen bei der Diagrammverfeinerung in vielen Teildiagrammen der Überblick über das Gesamtsystem verloren gehen.

Nebenläufigkeit, in [Harel 87] und in dieser Arbeit als Orthogonalität (*orthogonality*) bezeichnet, umfaßt innerhalb von Statecharts die Möglichkeit, unabhängige Systemkomponenten durch unabhängige Statecharts verfeinert darzustellen. Innerhalb der Verfeinerung von orthogonalen Zuständen können wie im Abschnitt 1.2.2 dargestellt, nebenläufige Zustandsübergänge stattfinden.

Broadcast Kommunikation (*broadcast-communication*) ermöglicht eine Synchronisation von Zustandsübergängen. Hierzu werden externe Ereignisse oder durch Aktionen generierte interne Ereignisse dem Statechart als Trigger seiner Zustandsübergänge global zur Verfügung gestellt. Das bedeutet, daß sowohl ein intern generiertes als auch ein externes Ereignis an beliebiger Stelle innerhalb des Statecharts einen Zustandsübergang auslösen kann.

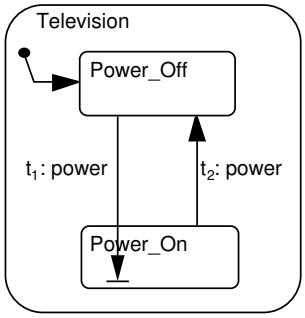

Abbildung 1.2: *Television* Statechart

In der Abbildung 1.2 ist exemplarisch das Verhalten eines Fernsehgerätes (*Television*) als Statechart auf oberster Abstraktionsebene in einer von der OMT Notation des dynamischen OMT Modells abweichenden und in der Literatur üblicherweise verwendeten Darstellung gezeigt. In Anlehnung an das im Abschnitt **??** dargestellte OMT Metamodell spezifiziert ein Statechart innerhalb des dynamischen OMT Modells das Verhalten einer Klasse des OMT Objektmodells (hier: OMT Klasse *Television*). Auf der in der Abbildung 1.2 wiedergegebenen Ebene der Abstraktion kann ein Objekt der Klasse *Television* die Zustaände *Power_Off* und *Power_On* einnehmen, wobei der Zustand *Power_On* nach dem Konzept der Tiefe auf einer nächst tieferen Abstraktionsebene in einen weiteren Zustandsdiagramm verfeinert ist (siehe Abbildung 1.4.

Die Konzepte der Tiefe, Orthogonalität und Broadcast Kommunikation werden in den Ausführungen über Zustände und Zustandsverfeinerungen im Abschnitt 1.2 und über die Semantik von Ausführungsschritten in Statecharts im Abschnitt 1.4 vertiefend erläutert und anhand der konkreten TL Implementierung der Tycoon Umgebung *StatechartEnv* für die statechartbasierte textuelle TL Modellierung von dynamischen Systemverhalten dargestellt.

1.2 Zustände und Zustandsverfeinerungen

Nach dem im Abschnitt 1.1 eingeführten Konzept der Tiefe kann jeder Zustand eines Statecharts auf beliebig tief verfeinerten Abstraktionsebenen Unterzustände beinhalten. Dabei sind unterschiedliche Arten der Verfeinerung von Zuständen möglich.

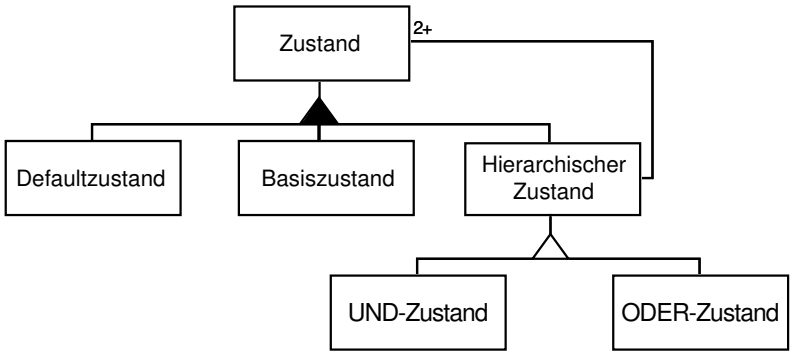

Abbildung 1.3: Generalisierungseigenschaften von Zuständen

In der Abbildung 1.3 sind mit Hilfe der Notation des OMT Objektmodells die Generalisierungseigenschaften von Zuständen dargestellt. Generalisierung steht innerhalb der OMT Objektmodellierung für eine Relation zwischen einer Klasse und einer oder mehrerer spezialisierter Unterklassen dieser Klasse [Rumbaugh et al. 91]. Beispielsweise ist ein Zustand als Oberklasse eines Default-, eines Basiszustands oder eines hierarchischen Zustands anzusehen, wobei seine Unterklassen nicht disjunkt sind. Das heißt, daß ein Objekt der Klasse *Basiszustand* oder ein Objekt der Klasse *Hierarchischer Zustand* die Ausprägung eines Objekts der Klasse *Defaultzustand* annehmen kann. Ein hierarchischer Zustand umfaßt mindestens zwei Zustände, die ihrerseits wiederum nach den eben erläuterten Generalisierungseigenschaften von Zuständen wie in der Abbildung 1.3 dargestellt verfeinert werden können.

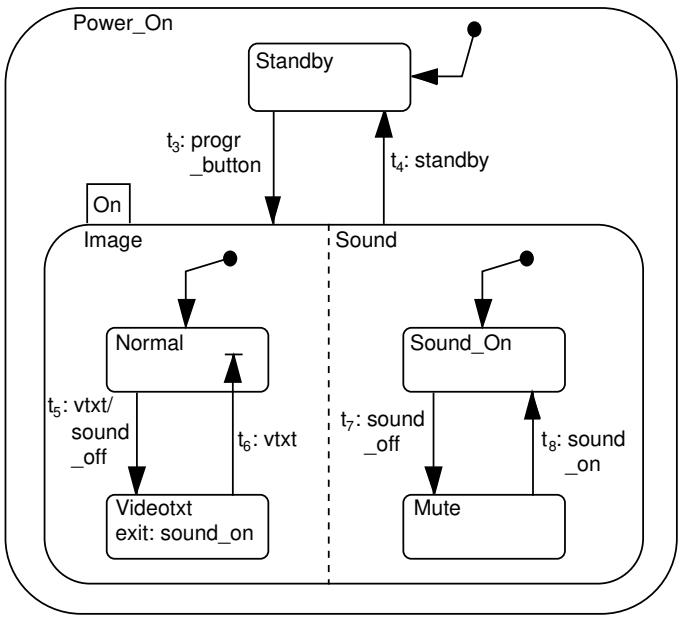

Abbildung 1.4: Verfeinerter Zustand *Power_On*

Eine Darstellung des in der Abbildung 1.2 auf oberster Abstraktionsebene unverfeinert wiedergegebenen Zustands *Power_On* innerhalb des *Television* Statecharts auf einem niedrigeren Abstraktionsniveau ist in der Abbildung 1.4 enthalten. Wie eine adäquate textuelle Repräsentation dieses grafisch repräsentierten Statecharts innerhalb des Tycoon Systems vorgenommen werden kann, wird in den folgenden Abschnitten exemplarisch dargestellt. Eine detaillierte Wiedergabe der TL Repräsentation des im Anhang **??** komplett grafisch spezifizierten Anwendungsbeispiels der Fernsehsteuerung ist im Anhang **??** aufgeführt.

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht ein Zustand dem mit dem abstrakten Datentypen *Transition* wechselseitig rekursiven abstrakten Datentypen *State*:

```
Let Rec State <:Ok =
  Tuple
    label :String
    var father :optional.T(State)
    transitions :Transitions
    actionsWhenEntering :Actions
    actionsWhenLeaving :Actions
    throughoutActivities :Activities
    case basic
    case compositeAnd with children :States
    case compositeOr with
      children :States defaultState :State
  end (* Tuple *)
and States <:Ok = set.T(State)
and Transition <:Ok =
  Tuple
    ...
  end (* Tuple *)
...
```

Basiszustände oder hierarchische Zustände stellen spezielle Ausprägungen dieses Datentyps dar. Ein UND-Zustand (*compositeAnd*) umfaßt neben den Attributen eines Basiszustands (*basic*) eine Menge von orthogonalen Komponenten (*children*). Ein ODER-Zustand beinhaltet neben seinen sich gegenseitig ausschließenden Subzuständen einen davon ausgezeichneten Defaultzustand (*defaultState*), welches derjenige Zustand ist, an den die Systemkontrolle bei einem Zustandsübergang in diesen, den Defaultzustand umfaßenden verfeinerten ODER-Zustand übergeben wird. In der Abbildung 1.4 ist dieser ausgezeichnete Defaultzustand innerhalb des ODER-Zustands *Sound* der Basiszustand *Sound_On*.

Eine Erläuterung der abstrakten Tycoon Datentypen *Transition* und *Action* erfolgt im Abschnitt 1.3. Aktivitäten vom abstrakten Tycoon Datentyp *Activity* werden in der in dieser Arbeit entwickelten *StatechartEnv* Implementierung vorerst ohne semantische Bedeutung als leere Menge vom Typ *Activity* realisiert.

1.2.1 Basiszustand

Basiszustände bilden den eigentlichen Zustandsraum von Statecharts [Fricke 95] und werden nicht weiter verfeinert. In der einfachsten Form besteht ein Statechart aus einer Menge von Basiszuständen, die wie im Abschnitt 1.2.2 beschrieben zu einem ODER-Zustand zusammengefaßt werden.

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein Basiszustand mit der Funktion *createBasicState* in einer textuellen Repräsentation generiert:

```
let createBasicState(stateLabel :String  actionsWhenEntering :Actions
                     actionsWhenLeaving :Actions
                     throughoutActivities :Activities) :State =
  begin
    let state = tuple_case basic of State with StateLabel
      var optional.nil(:State) newTransitionSet() actionsWhenEntering
      actionsWhenLeaving throughoutActivities end
    set.insert(:State local.StateSet state)
    state
  end
```

Eine Menge von Basiszuständen wird niemals zu einem UND-Zustand zusammengefaßt, da die Komponenten eines UND-Zustands ausschließlich ODER-Zustände sein können.

Ein Beispiel für eine statische Bindung eines mit der Funktion *createBasicState* generierten Basiszustands *Sound_On* (siehe Abbildung 1.4) an einen Tycoon Bezeichner lautet:

```
let soundOn = createBasicState("Sound_On" emptySet(:Action) emptySet(:Action)
  emptySet(:Activity))
```

Eine komplette textuelle Tycoon Repräsentation des im Anhang **??** dargestellten *Television* Statecharts ist im Anhang **??** aufgeführt. Das gleiche gilt für die in den folgenden Abschnitten vorgestellten Implementierungsbeispiele, die alle Auszüge des Gesamtbeispiels im Anhang **??** sind.

1.2.2 Hierarchischer Zustand

Ein hierarchischer Zustand eines Statecharts kann nach dem im Abschnitt 1.1 eingeführten Konzept der Tiefe rekursiv eine Dekomposition eines Zustands in einer der folgenden Formen sein:

- UND-Dekomposition von Zuständen innerhalb von zwei oder mehreren orthogonalen Statecharts auf einer niedrigeren Abstraktionsebene als UND-Zustand
- ODER-Dekomposition von Zuständen innerhalb eines Statecharts auf einer niedrigeren Abstraktionsebene als ODER-Zustand

ODER-Zustand

In der Abbildung 1.5 ist der verfeinerte ODER-Zustand *Videotxt* dargestellt.

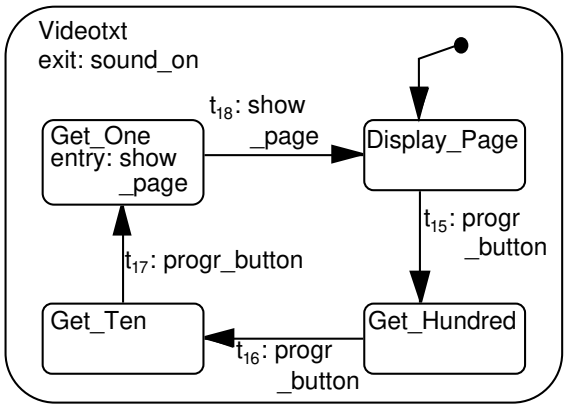

Abbildung 1.5: ODER-Zustand *Videotxt*

Generiert wird die textuelle Tycoon Repräsentation eines ODER-Zustands mit der Funktion *createCompositeOrState*:

```
let createCompositeOrState(stateLabel :String  actionsWhenEntering :Actions
                           actionsWhenLeaving :Actions
                           throughoutActivities :Activities
                           children :States  defaultState :State) :State =
  if set.size(:State children) > 1
    andif set.member(:State defaultState children) then
      let state = tuple_case compositeOr of State with stateLabel
        var optional.nil(:State) newTransitionSet() actionsWhenEntering
```

```
      actionsWhenLeaving throughoutActivities children defaultState end
    iter.forEach(:State set.elements(children) fun(st :State) :Ok
      if set.member(:State st local.stateSet)
        andif optional.null(st.father) then
          st.father := optional.new(state)
      else
        raise childrenError end
      end)
    set.insert(:State local.stateSet state)
    state
  else
    raise argumentError end
  end
```

Zur Erstellung einer Bindung eines mit der Funktion *createCompositeOrState* generierten ODER-Zustands *Videotxt* (siehe Abbildung 1.5) an einen Tycoon Bezeichner sind die folgenden Schritte erforderlich:

```
let displayPage = createBasicState("Display_Page" ...)
...
let getOne = createBasicState("Get_One" ...)

let stateSet = emptySet(:State)
set.insert(stateSet display Page)
...
set.insert(stateSet getOne)

let videotxt = createCompositeOrState("Videotxt" emptySet(:Action)
  actionSet emptySet(:Activity) stateSet displayPage)
```

Das komplette Beispiel des *Television* Statecharts ist im Anhang **??** enthalten.

UND-Zustand

In der Abbildung 1.6 ist der UND-Zustand *Display* dargestellt, welcher die beiden orthogonalen ODER-Komponenten *Show_Picture* und *Show_ProgrNr* beinhaltet:

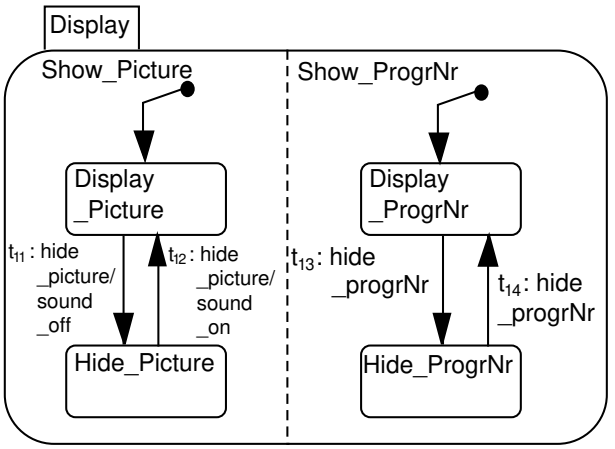

Abbildung 1.6: UND-Zustand *Display*

Generiert wird die Tycoon Repräsentation eines UND-Zustands mit der Funktion *createCompositeAndState*:

```
let createCompositeAndState(stateLabel :String  actionsWhenEntering :Actions
                            actionsWhenLeaving :Actions
                            throughoutActivities :Activities
                            children :States) :State =
  if set.size(:State children) > 1 then
    let state = tuple_case compositeAnd of State with stateLabel
      var optional.nil(:State) newTransitionSet() actionsWhenEntering
      actionsWhenLeaving throughoutActivities children end
    iter.forEach(:State set.elements(children) fun(st :State) :Ok
      if set.member(:State st local.stateSet)
        andif optional.null(st.father) then
          st.father := optional.new(state)
      else
        raise childrenError end
      end)
    set.insert(:State local.stateSet state)
    state
  else
    raise childrenError end
```

end

Eine statische Bindung eines mit der Funktion *createCompositeAndState* generierten UND-Zustands *Display* (siehe Abbildung 1.6) an einen Tycoon Bezeichner kann wie folgt vorgenommen werden:

```
let displayPicture = createBasicState("Display_Picture" ...)
...
let hideProgrNr = creatBasicState("Hide_ProgrNr" ...)

let showPicture = createCompositeOrState("Show_Picture" ...)
let showProgrNr = createCompositeOrState("Show_ProgrNr" ...)

let stateSet = emptySet(:State)
set.insert(stateSet showPicture)
set.insert(stateSet showProgrNr)

let display = createCompositeAndState("Display" emptySet(:Action)
  emptySet(:Action) emptySet(:Activity) stateSet)
```

Abgeschlossen wird die Spezifikation eines Statecharts mit einem ausgezeichneten Superzustand, dem Wurzelzustand, der über alle Verfeinerungsebenen hinweg der Vorfahr aller in diesem Statechart enthaltenen Zustände ist.

Die Tycoon Implementierung zur Generierung eines Wurzelzustandes (hier: Wurzelzustand *Television* aus der Abbildung 1.2) folgt der Implementierung der Funktion *createCompositeOrState*, mit dem Unterschied, daß die Funktion *createRootState* den so erzeugten Zustand als Superzustand des Statecharts kennzeichnet:

```
let powerOff = createBasicState("Power_Off" ...)
...
let powerOn = createCompositeOrState("Power_On" ...)

let stateSet = emptySet(:State)
set.insert(stateSet powerOff)
set.insert(stateSet powerOn)

let television = createRootState("Television" emptySet(:Action)
  emptySet(:Action) emptySet(:Activity) stateSet powerOff)
```

Weitere Zustände können innerhalb dieses einen Statecharts nicht als Wurzelzustände ausgezeichnet werden.

1.3 Zustandsübergänge

Ein Zustandsübergang überführt im einfachsten Fall [Huizing et al. 88] einen Zustand in einen Folgezustand. Innerhalb dieser Arbeit schränke ich Zustandsübergänge auf diesen einfachsten Fall ein. Die semantischen Konsequenzen von verzweigten Transitionen, die einen Quellzustand mit mehreren Folgezuständen oder mehrere Quellzustände mit einem Folgezustand verbinden, werden hier nicht untersucht. Ein Zustandsübergang wie zum Beispiel die Transition t_6 zwischen dem Zustand *Videotxt* und *Display* in der verfeinerten Darstellung des Zustands *Image* im Anhang **??** kann Zustände über Hierarchieebenen hinweg verbinden.

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht ein Zustandsübergang dem mit dem abstrakten Datentypen *State* wechselseitig rekursiven abstrakten Datentyp *Transition*:

```
Let Rec State <:Ok =
  Tuple
    ...
  end
and States <:Ok = set.T(State)
and Transition <:Ok =
  Tuple
    source :State
    target :State
    trigger :Events
    conditions :States
    actions :Actions
  end
and Transitions <:Ok = set.T(Transition)
```

Als Attribute besitzt ein Zustandsübergang von Typ *Transition* einen Quell- und Zielzustand von Typ *State*, eine Menge von Triggern des Typs *Events* und eine Menge von Aktionen vom Typ *Actions*. Desweiteren kann ein Zustandsübergang durch Bedingungen, eine Menge von Zuständen, eingeschränkt werden. Diese Bedingungen, eine Menge von Typ *States* lassen einen Zustandsübergang nur dann stattfinden, wenn die in der Bedingung spezifizierten Zustände zum Zeitpunkt des möglichen Übergangs in der aktuellen Systemkonfiguration enthalten sind.

Hier folgt ein Bsp. zur Verbindung von Zuständen

1.3.1 Ereignis

Ein Ereignis lst in der Form eines Triggers einen oder mehrere Zustandübergänge aus. Dabei kann das Ereignis in der Bezeichnung der Transition ein zusammengesetzter logischer Ausdruck der Form einer Konjunktion, Disjunktion oder Negation sein. Innerhalb dieser Arbeit

schränke ich Ereignisse auf primitive Ereignisse, also atomare Ereignisse, ein. Ereignisse lassen sich unterscheiden in externe und interne Ereignisse.

Externe Ereignisse sind solche, die in Anlehnung an das im Abschnitt **??** dargestellte OMT Ereignispfaddiagramm zwischen OMT Klassen eines Systems existieren und von außen auf die das Verhalten dieser OMT Klassen spezifizierenden Statecharts einwirken. Interne Ereignisse werden von einer mit einer Transition assoziierten Aktion innerhalb eines Statecharts bei einem entsprechenden Zustandsübergang generiert. Daraus resultiert, daß diese generierten internen Ereignisse bei der Berechnung des nächsten auszuführenden Zustandsübergangs berücksichtigt werden müssen. Solche Transitionen werden als *Micro Steps* bezeichnet [Pnueli, Shalev 91]. Die in einem *Micro Step* ausgeführte Transition kann ebenfalls interne Ereignisse generieren, die ihrerseits wiederum anderen Transitionen als Trigger dienen können.

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht ein Ereignis dem abstrakten Datentypen *Event*:

```
Let Rec Event <:Ok =
  Tuple
    case primitive with eventName :String
    ...
  end
and Events <:Ok = list.T(Event)
```

Als einziges Attribut umfaßt das in dieser Arbeit berücksichtigte primitive Ereignis einen Ereignisnamen von Typ *String*.

Hier folgt ein Bsp. zur Generierung eines Ereignisses ...

1.3.2 Aktion

In der Bezeichnung eines Zustandsübergangs kann eine Menge von Ereignissen spezifiziert werden, die bei der Durchführung des Zustandsübergangs generiert werden sollen. Diese intern generierten Ereignisse führen umgehend zu Zustandsübergängen in der Form der im Abschnitt 1.3.1 erläuterten *Micro Steps*.

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht eine Aktion dem abstrakten Datentyp *Action*:

```
Let Action <:Ok =
  Tuple
    actionLabel :String
    trigger :Event
    var actionPart :Fun :Ok
  end
```

Als Attribute besitzt eine Aktion von Typ *Aktion* einen Bezeichner von Typ *String*, ein zu generierendes Ereignis von Typ *Event* und einen Aktionsteil, der eine beliebige TL Funktionalität beinhalten kann, welche als Rückgabeparameter den Wert **ok** vom Typ **Ok** liefern muß. Detailliert wird dieser Aktionsteil im Abschnitt **??** bei der Anbindung externer Dienste an Zustandsübergänge eines Statecharts erläutert.

Hier folgt ein Bsp. zur Generierung einer Aktion

1.4 Semantik von Ausführungsschritten

Zustandsübergänge überführen ein System von einem Zustand in einen Folgezustand. Ein Ausführungsschritt überführt dabei ein System von einer konkreten Konfiguration in eine Folgekonfiguration, die den Systemzustand repräsentiert. Ausgehend von einer aktuellen Konfiguration kann stets nur eine eingeschränkte Menge von Zustandsübergängen stattfinden. Im Abschnitt 1.4.2 ist die „Berechnung“ solch einer Menge von ausführbaren Zustandsübergängen erläutert. Die in der Tycoon Umgebung *StatechartEnv* exemplarisch implementierte Statechart Semantik zur Ausführung von Statecharts auf der Basis einer gegebenen Konfiguration ist in [Pnueli, Shalev 91] vorgestellt.

1.4.1 Systemkonfiguration

Eine Systemkonfiguration beschreibt den globalen Zustand eines Statecharts, welcher sich aus einer über Abstraktionsebenen betrachteten maximal konsistenten Teilmenge aller Statechart Zustände zusammensetzt. Eine Teilenge von Zuständen eines Statecharts heißt konsistent, falls alternativ gilt:

- Alle Zustände dieser Teilmenge stehen paarweise miteinander in einer Verwandtschaftsbeziehung. Zwei Zustände sind miteinander verwandt, falls der eine Vorfahr oder Nachkomme des anderen ist, wobei zwischen beiden Zuständen mehrere Verfeinerungsebenen liegen können.
- Alle Zustände dieser Teilmenge sind paarweise orthogonal zueinander. Zwei Zustände sind orthogonal, falls sie nicht in einer Verwandschaftsbeziehung stehen und ihr kleinster gemeinsamer Vorfahr ein UND-Zustand ist. Der kleinste gemeinsame Vorfahr zweier Zustände ist der Zustand, bei dem sich die Hierarchiepfade von den Zuständen zur Wurzel des Statecharts hin treffen [Fricke 95].

Eine konsistente Teilmenge X von Zuständen eines Statecharts heißt maximal konsistent, falls für jeden Statechart Zustand s außerhalb dieser Teilmenge von Zuständen gilt, daß die Vereinigungsmenge $X \cup s$ eine nicht konsistente Menge von Zuständen ergibt.

In Anlehnung an das Konzept der Markierung von Bedingungen in Netzen [Reisig 82], kann innerhalb von Statechrts ein in einer aktuellen Systemkonfiguration enthaltener Zustand grafisch ebenfalls als markiert dargestellt werden.

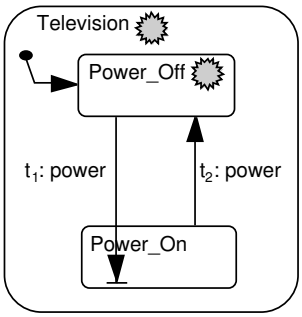

Abbildung 1.7: Ausgangskonfiguration

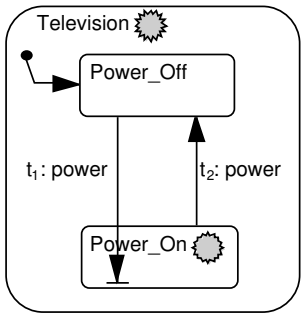

Abbildung 1.8: Folgekonfiguration

In der Abbildung 1.7 ist eine Ausgangskonfiguration als maximal konsistente Teilmenge von Zuständen des *Television* Statecharts dargestellt. Sie umfaßt die markierten Zustände *Television* und *Power_Off*. Ähnlich wie bei Stellen/Transitions-Netzen [Jessen, Valk 87], bei denen eine Transition nur schalten kann, wenn die davor- und dahinterliegenden Stellen entsprechend markiert sind kann in einem Statechart ein Zustandsübergang nur stattfinden, wenn der entsprechende Quellzustand in der aktuellen Systemkonfiguration enthalten ist. Ausgehend von der in der Abbildung 1.7 dargestellten Ausgangskonfiguration $C = \{$*Television*, *Power_On*$\}$ wird beim Eintreten des Ereignisses *power* die in der Abbildung 1.8 gezeigte Folgekonfiguration erreicht, wobei die Konfiguration über diese Abstraktionsebene hinaus die Defaultzustände der weiter verfeinerten Zustände des Zustands *Power_On* umfaßt (siehe Abbildung 1.4). Die in der Abbildung 1.8 dargestellte Folgekonfiguration C' beinhaltet die Zustände *Television*, *Power_On* und *Standby*.

Angelegt werden kann eine Ausgangskonfiguration ausgehend von dem im Anhang **??** dargestellten Tycoon *Television* Statechart innerhalb der Tycoon Umgebung *StatechartEnv* mit der Funktion *actualConfiguration*:

```
...
let stateSet = emptySet(:State)
set.insert(stateSet tv)
set.insert(stateSet powerOff)

television.actualConfiguration(stateSet)
```

1.4.2 Schrittkonstruktion

Wie im Abschnitt 1.4.1 dargestellt, kann von einer aktuellen Konfiguration stets nur eine bestimmte Menge von Zustandsübergängen stattfinden. Die folgenden Bedingungen schränken die Menge der ausführbaren Transitionen ein [Day 93]:

- Ein Zustandsübergang kann stattfinden, falls sich die Systemkontrolle in dem Quellzustand des Zustandsübergangs befindet. Das bedeutet, daß dieser Quellzustand in der aktuellen Systemkonfiguration enthalten sein muß. Liegt das zustandsübergangsauslösende Ereignis vor, so kann das System in eine Folgekonfiguration übergehen.
- Innerhalb eines ODER-Zustands kann nur ein Zustandsübergang zu einem Zeitpunkt stattfinden.
- Innerhalb eines UND-Zustands können orthogonale Zustandsübergänge simultan innerhalb der einzelnen UND-Komponenten stattfinden.

Die Menge der ausführbaren Transitionen ist abhängig von den externen Ereignissen und den innerhalb eines Ausführungsschritts generierten internen Ereignissen, da jede Transition wie im Abschnitt 1.3 dargestellt zustandübergangsauslösende Ereignisse durch Aktionen generieren kann, die ihrerseits wiederum zu einer Folge von Zustandsübergängen (*Micro Steps*) führen können.

Ein Ausführungsschritt (*Macro Step*) ist eine maximale Folge solcher ausführbarer *Micro Steps*, an deren Ende sich das System in einem stabilen Zustand befindet, in dem keine internen Ereignisse mehr generiert werden und die nächste Systemkonfiguration erst wieder durch einen *Macro Step* auf der Basis eines externen Ereignisses erreicht werden kann. Solch eine stabile Konfiguration wird als eine dauerhafte Konfiguration bezeichnet. Während der Abfolge der *Micro Steps* werden keine externen Ereignisse als Trigger für Zustandsübergänge zugelassen, die eine Konfiguration in eine Folgekonfiguration überführen könnten. Diese Bedingung befriedigt die den Statecharts zugrungeliegende Synchronitätshypothese, die besagt, daß ein System immer erst seine vollständige Reaktion auf ein externes Ereignis inklusive aller generierten internen Ereignisse durchführen kann, bevor das nächste externe Ereignis tatsächlich eintritt [Pnueli, Shalev 91].

In der Originalformulierung der Statechartsemantik [Harel 87] bestimmen sowohl die Menge der externen Ereignisse zu Beginn eines Ausführungsschritts als auch die Menge der in einem *Macro Step* generierten internen Ereignisse die Transitionen, die zu einem Zustandsübergang führen können. Das bedeutet, daß die Ereignisse für die Länge eines Ausführungsschrittes vorliegen, nicht aber im nächst folgenden Schritt berücksichtigt werden. Eine neue Menge externer Ereignisse muß vorhanden sein, um den nächsten *Macro Step* auszulösen.

Zur Bestimmung der Zustanbdsübergänge, die beim Vorliegen externer Ereignisse und durch Aktionen generierte interne Ereignisse ausgeführt werden können gilt für eine Menge von

Transitionen T, eine die im Abschnitt 1.4.1 dargestellten Bedingungen für gültige Systemkonfigurationen erfüllende aktuelle Systemkonfiguration C und eine Menge von externen Ereignissen I eines Statecharts, die Funktion *En* (*Enabling Function* [Leveson et al. 94]:

$$En(T,C,I) = relevant(C) \cap consistent(T) \cap triggered(I \cup generated(T))$$

Hierbei bezeichnet nach [Harel et al. 90]

- *relevant(C)* die Menge der Transitionen, deren Startzustände sich in einer aktuellen Konfiguration befinden.
- *consistent(T)* die Menge der Transitionen, die paarweise miteinander konsistent sind. Eine Transition ist konsistent mit sich selbst. Desweiteren sind zwei Transitionen konsistent miteinander, wenn die Arenen beider Transitionen, also der ODER-Zustand, der sowohl den Quell- als auch den Zielzustand der Transition enthält, zueinander orthogonal sind (siehe Abschnitt 1.4.1.
- *triggered(E)* die Menge der Zustandsübergänge, deren Trigger in der Menge der vorliegenden Ereignisse E enthalten sind.
- I die Menge der externen Ereignisse.
- *generated(T)* die Menge der durch die Aktionen der Transitionen der Menge T generierten internen Ereignisse.

Ein konkretes Beispiel zur Berechnung von Ausführungsschritten anhand der *Enabling Function En* ist in [Pnueli, Shalev 91] enthalten. Im Rahmen der Implementierung der im Kapitel **??** vorgestellten Tycoon Validierungsumgebung wird innerhalb der Tycoon Umgebung *StatechartEnv* die soeben dargestellte Semantik von Ausführungsschritten als Grundlage der Funktion *simulate* zur Ausführung von Statecharts benutzt:

```
let rec simulate(input :Events) :Ok =
  if not(list.empty(:Event input)) then
    let actualEvent = list.head(:Event input)
    let remainingEvents = list.tail(:Event input)
    let correspondingTransitions = selectTransitions(actualEvent)
    let transitionIter = set.elements(:Transition correspondingTransitions)
    iter.forEach(:Transition transitionIter
      fun(transition :Transition) :Ok enableTransitions(transition input))
    simulation(remainingEvents)
  end (* if *)
```

Für jedes Ereignis aus der Liste aller externen Ereignisse berechnet die Funktion *simulate* die ausführbaren Zustandsübergänge und führt diese und eventuell intern generierte *Micro Steps* durch, bis am Ende alle externen Ereignisse abgearbeitet sind und sich das System in einer stabilen Konfiguration befindet.

Ab Abschnitt 1.3 wird der Txt. nochmals ueberarbeitet. Viele Grueße, Oli.

Literaturverzeichnis

Day 93: N. Day. „A Model Checker for Statecharts“. Technical Report 93-35, Department of Computer Science, University of British Columbia, Vancouver, Kanada, 1993.

Fricke 95: O. Fricke. „Die Strukturierte Analyse auf der Basis höherer Petrinetze“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

Glinz 95: M. Glinz. „An Integrated Formal Model of Scenarios Based on Statecharts“. In: *W. Schäfer, P. Botella (Hrsg.). Software Engineering - ESEC '95. Springer, Berlin u.a.*, Seite 231–271, 1995.

Harel et al. 90: D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, und M. Trakhtenbrot. „STATEMATE: A Working Environment for the Development of Complex Reactive Systems“. In: *IEEE Transactions on Software Engineering, Jahrg. 16, Nr.4, New York*, Seite 401–413, 1990.

Harel 87: D. Harel. „Statecharts: A Visual Formalism for Complex Systems“. In: *Science of Computer Programming, Nr. 8, Amsterdam*, Seite 231–274, 1987.

Hopcroft, Ullman 90: J.E. Hopcroft und J.D. Ullman. *Einführung in die Automatentheorie.* Addison-Wesley, Bonn u.a., 1990.

Huizing et al. 88: C. Huizing, R. Gerth, und W.P. de Roever. „Modelling Statecharts Behaviour in a fully abstract Way“. In: *M. Dauchet, M. Nivat (Hrsg.). Lecture Notes in Computer Science 299, Springer, Berlin u.a.*, Seite 271–294, 1988.

Huizing, de Roever 91: C. Huizing und W.P. de Roever. „Inroduction to design choices in the semantics of Statecharts“. In: *Information Processing Letters, Jahrg. 37, Nr. 2, Amsterdam*, Seite 205–213, 1991.

Jessen, Valk 87: E. Jessen und R. Valk. *Rechensysteme: Grundlagen der Modellbildung.* Springer, Berlin u.a., 1987.

Leveson et al. 94: N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, und J.D. Reese. „Requirements Specification for Process-Control Systems“. In: *IEEE Transactions on Software Engineering, Jahrg. 20, Nr.9, New York*, Seite 684–707, 1994.

Pnueli, Shalev 91: A. Pnueli und M. Shalev. „What is in a Step: On the Semantics of Statecharts“. In: *T. Ito, A.R. Meyer (Hrsg.). Lecture Notes in Computer Science 780, Springer, Berlin u.a.*, Seite 244–264, 1991.

Reisig 82: W. Reisig. *Petrinetze: Eine Einführung.* Springer, Berlin u.a., 1982.

Rumbaugh et al. 91: J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, und W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, London, 1991.

v.d. Beeck 94: M. v.d. Beeck. „A Comparison of Statecharts Variants“. In: *H. Langmaack, W.P. de Roever, J.Vytopil (Hrsg.). Lecture Notes in Computer Science 863, Springer, Berlin u.a.*, Seite 128–148, 1994.