# Kapitel 11: Prozeßmodellierung und Workflow-Management

## 11.1 Prozessmodellierung

Beim Entwurf von Informationssystemen müssen außer den Daten auch Funktionen und Prozesse der Anwendung modelliert werden. Unter Funktionen verstehen wir dabei elementare Bausteine der Anwendungsfunktionalität aus Benutzersicht, also mächtige Methoden auf „Business Objects“ wie z.B. die betriebswirtschaftliche Verbuchung einer Bestellung oder die Prämienberechnung für eine Versicherungspolice. Mehrere solcher Funktionen können zu einem Geschäftsprozeß kombiniert werden, und ein Geschäftsprozeß kann u.U. selbst wieder als umfassendere Funktion angesehen werden. Die primitiveren Funktionen der Anwendung schließlich werden auf Methoden der Datenobjekte und programmiertechnische Klassen abgebildet, die z.B. mit UML modelliert werden. Entscheidend für die Anpassungsfähigkeit eines Unternehmens (oder auch einer Behörde) an die Dynamik der sich ständig wandelnden Kundenbedürfnisse und Marktanforderungen ist eine saubere Trennung von Funktionen, den betrieblichen Bausteinen, und Prozessen, den angepaßten Abläufen im Unternehmen und auch (z.B. in Logistikketten) gegenüber anderen Unternehmen.

Die Computerunterstützung betrieblicher Abläufe kann sich auf die einzelnen Funktionen beschränken, so daß die Koordination langlebiger Geschäftsprozesse beim menschlichen Sachbearbeiter verbleibt, oder sie kann auch die (weitgehend) automatisierte Steuerung der Prozesse umfassen. Für den zweiten Fall hat sich der Begriff *Workflow-Management* eingebürgert. Beispielanwendungen sind etwa die Bearbeitung von Kreditanträgen in einer Bank oder Schadensfällen in einer Versicherung, die Planung und Durchführung einer Marketingkampagne, die elektronische Einreichung, Begutachtung und Publikation wissenschaftlicher Zeitschriftenartikel oder - heute noch fiktiv - die elektronische Einreichung und automatisierte Verarbeitung von Steuererklärungen oder die Abwicklung aller bei Immobilienkäufen notariell, behördlich sowie finanz- und versicherungstechnisch erforderlichen Schritte. Derartige Workflows bestehen aus mehreren, automatisierten oder intellektuellen Arbeitsschritten - in der Literatur häufig *Aktivitäten* genannt -, die verschiedenen Ausführungsorganen - Sachbearbeitern oder Computerapplikationen - zugeordnet werden. Die von Aktivitäten aufgerufenen Applikationsprogramme sind die eigentlichen Funktionen der Anwendung, d.h. Aufrufe von Methoden auf *„Business Objects“* wie zum Beispiel Konto- oder Bestellungs-Objekten. Insofern ergibt sich aus der objektorientierten Datenmodellierung und der funktionsorientierten Modellierung von Geschäftsprozessen eine integrierte Darstellung der Anwendung.

In den folgenden Abschnitten werden verschiedene Spezifikationsmethoden für die Modellierung von Prozessen vorgestellt.

### 11.1.1 Prozeßmodellierung mit State-Charts (bzw. State– und Activity–Charts)

Der State– und Activity–Chart–Formalismus wurde in Arbeiten von David Harel (ursprünglich für “reaktive", eingebettete Steuerungssysteme) entwickelt und inzwischen in den Industrie–Standard UML übernommen (in UML heißen sie Zustandsübergangsdiagramme, englisch: State Transition Diagrams). State– und Activity–Charts stellen duale Sichtweisen einer Spezifikation dar.

Ein *Activity–Chart* spezifiziert den Datenfluß zwischen den Aktivitäten in Form eines gerichteten Graphen mit Aktivitäten als Knoten und Datenelementen als Kantenbeschriftung.
(Achtung: Diese Art von Activity-Chart hat nichts mit den Aktivitätsdiagrammen von UML zu tun. Letztere sind eine Variante von Petrinetzen und beschreiben Kontrollfluß.)

Ein *State–Chart* beschreibt das Verhalten einer Spezifikation, indem es den Kontrollfluß zwischen Aktivitäten als Transitionen zwischen Zuständen spezifizieren. Ein State–Chart ist im wesentlichen ein endlicher Automat mit einem ausgewiesenen Anfangszustand und Transitionen, die mit *Event–Condition–Action Regeln (ECA–Regeln)* beschriftet sind. Eine Transition von Zustand *X* nach Zustand *Y*, die mit der ECA–Regel *E[C]/A* beschriftet ist, feuert genau dann, wenn sich der Automat in Zustand *X* befindet, das Ereignis *E* eintritt und die Bedingung C erfüllt ist. Der Automat wechselt dann in den Zustand *Y* und die Aktion *A* wird ausgeführt. Bedingungen sind aussagenlogische Formeln über elementaren Bedingungen auf Zuständen (z.B. *in(S)*, die wahr ist, wenn Zustand S betreten ist) und - kontrollflußrelevanten - globalen Variablen (u.a. denjenigen Variablen, die für die Datenflußspezifikation im Activity-Chart verwendet werden). Eine Aktion kann sowohl explizit eine Aktivität starten (durch den Ausdruck *st!(aktivität))*, als auch Ereignisse erzeugen oder Bedingungen setzen (z.B. *fs!(C)* setzt die Bedingung *C* auf den Wert falsch). Jeder der drei Teile des *E[C]/A*–Tripels kann auch leer sein.

Zwei weitere Besonderheiten unterscheiden State–Charts von herkömmlichen Zustandsautomaten. Zum einen ist dies die im Hinblick auf schrittweise Verfeinerung und Abstraktion wichtige Möglichkeit zur *Schachtelung von Zuständen*. Ein Zustand kann ein komplettes State–Chart beinhalten, wobei beim Betreten eines übergeordneten Zustands implizit der jeweilige initiale Unterzustand betreten wird und beim Verlassen eines Elternzustandes alle betretenen Kinderzustände verlassen werden. Zum anderen erlauben State–Charts *orthogonale Zustandskomponenten*, mit denen Parallelität ausgedrückt wird, so daß Transitionen in verschiedenen orthogonalen Komponenten gleichzeitig feuern können.

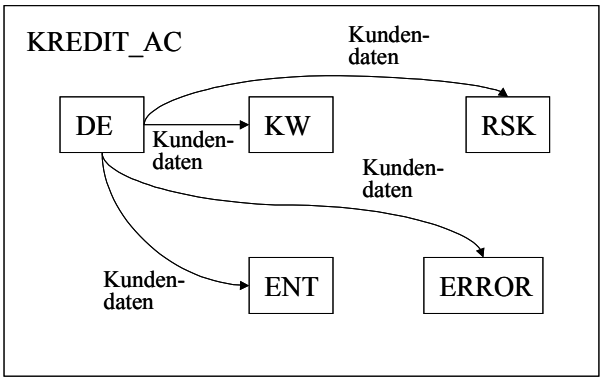
**Einfaches Beispiel 1: Simplifizierte Kreditantragsprüfung**

Eine simplifizierte Kreditantragsprüfung bestehe aus den Aktivitäten
- Dateneingabe (DE) zur Erfassung der Kundendaten (KD),
- Kreditwürdigkeitsprüfung (KW) des Kunden,
- Risikoabschätzung (RSK) aufgrund des Betrags, der Währung und Zahlungsart des Kredits,
- Entscheidung (ENT) über den Kreditantrag und einer
- generischen Fehlerbehandlungsaktivität (ERROR).

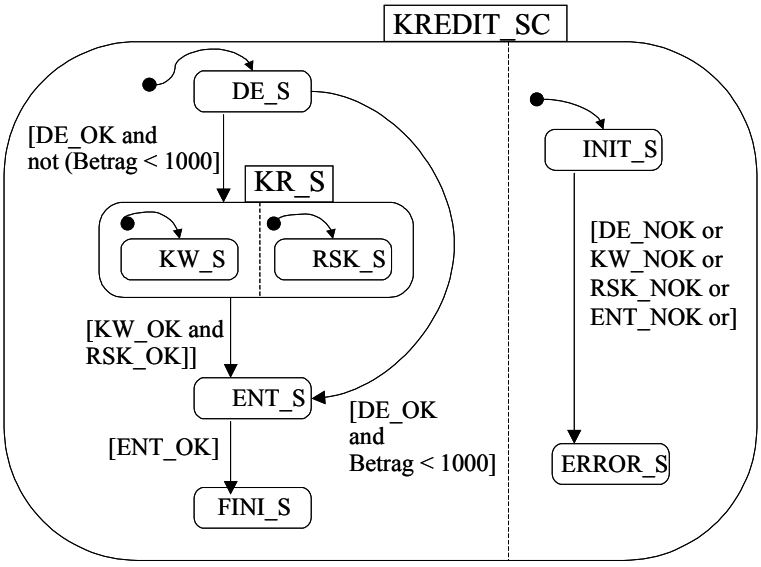
Die Aktivitäten KW und RSK sind nur bei Krediten über einer bestimmten Höhe notwendig; sie sind außerdem parallel ausführbar.

Jede Aktivität ist genau einem Zustand zugeordnet und soll mit dem Betreten des Zustands gestartet und mit dem Verlassen des Zustands beendet werden. Die Aktivitäten können als Methoden eines Objekts “Kreditantrag" angesehen werden. Insofern beschreiben State- und Activity-Charts das dynamische Verhalten eines solchen Objekts.

Activity-Chart:



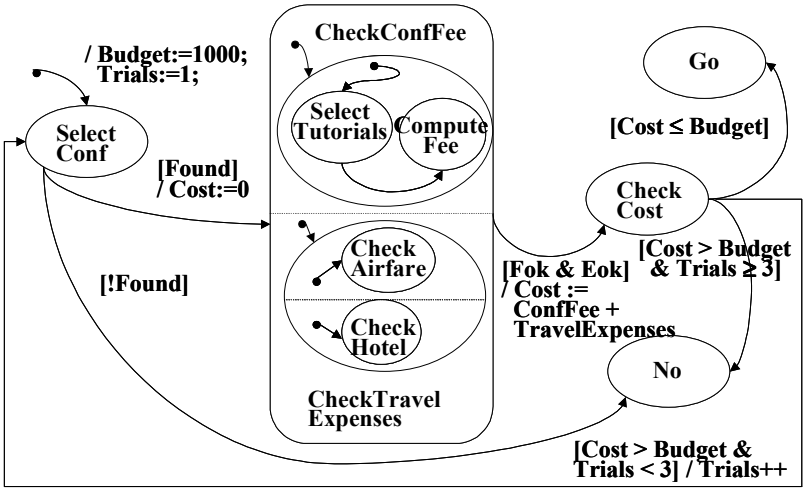

State-Chart:



**Einfaches Beispiel 2: Planung einer Konferenzreise**

Bei der – fiktiven - Planung einer Konferenzreise (z.B. eines wissenschaftlichen Mitarbeiters eines Universitätslehrstuhls) wird zunächst eine thematisch interessante Konferenz gewählt. Dann werden parallel die Konferenzgebühren und die Reisekosten für Hotel und Flug ermittelt. Wenn die Gesamtkosten im Rahmen des vorgesehenen Budgets liegen, fällt eine positive Entscheidung für die Konferenzteilnahme (Endzustand „Go“); ansonsten werden bis zu drei alternative Konferenzen analog evaluiert. Nach drei erfolglosen Versuchen, gibt man auf (Endzustand „No“).

Statechart:



**Komplexeres Beispiel: Bestellungen via Internet (Electronic Commerce)**

**Kontrollfluß.** Der Workflow beginnt mit einer Aktivität, die eine *neue Bestellung* in einer Datenbank erfaßt. Das Versandhaus stellt zwei Arten von Zahlungsmodalitäten zur Verfügung. Der Kunde kann entweder bei der Bestellung seine Kreditkartennummer angeben oder eine Rechnung anfordern, nach deren Erhalt er die angefallene Rechnungssumme überweisen muß. Der Kontrollfluß muß an dieser Stelle aufgespalten werden. Wurde eine *Kreditkartenzahlung* gewünscht, wird zunächst eine Aktivität zur Bonitätsprüfung der Kreditkarte gestartet. Nach der Prüfung der Kreditkarte werden die beiden Kontrollfluß–Pfade wieder zusammengeführt.

Nach der Registrierung der Bestellung werden der *Versand* sowie das Senden einer *Bestätigungs–Mail* an den Kunden initiiert. Dazu werden parallel zwei Sub–Workflows gestartet. Einer der beiden Sub–Workflows startet eine Aktivität, die eine Bestätigung an den Kunden versendet. Das Versenden von Waren erfolgt extern aus einem oder mehreren Lagern, die u. U. sogar von separaten Großhändlern autonom verwaltet werden. Daher startet der zweite Sub–Workflow zunächst eine Aktivität, die für jedes Lager die auszuliefernden Artikel bestimmt. Die Zuordnung wird in Artikellisten festgehalten. Innerhalb eines jeden Lagers, aus dem Artikel für die aktuelle Bestellung ausgeliefert werden müssen, wird eine Anwendung gestartet, die als Parameter die Artikelliste erhält und eine Versendegarantie zurückliefert. In einer Schleife werden so oft einzelne Lager angesprochen, bis alle Artikel auf der Bestellung bearbeitet wurden und die Versendung garantiert wurde. Wenn beide Subworkflows beendet sind, wird der Kontrollfluß erneut aufgespalten, abhängig davon, welche Zahlungsmodalität gewünscht wurde. Wurde Zahlung per Kreditkarte gewählt, wird die Kreditkarte nun belastet und der Workflow beendet. Auf diesem Kontrollfluß–Pfad werden keinerlei menschliche Eingriffe benötigt.

Soll eine Rechnung gestellt werden, könnte zusätzlich eine Aktivität gestartet, die auf den *Eingang der Zahlung* wartet. Diese Aktivität wäre intellektuell–interaktiver Natur, da die Eingabe von Daten, z.B. ob der Rechnungsbetrag vollständig eingegangen ist, durch einen Mitarbeiter des Versandhauses erfolgt. Ist die Zahlung erfolgt, ist das Szenario beendet. Sollte die Rechnung nicht innerhalb einer Frist von 14 Tagen beglichen sein, wird dem Kunden eine Mahnung geschickt. Dieser Vorgang wiederholt sich maximal dreimal im Abstand von 14 Tagen, bis die Rechnung beglichen ist. Wurde die Zahlung 14 Tage nach der dritten Mahnung noch immer nicht geleistet, so wird die Rechtsabteilung des Versandhauses informiert. Das Mahnverfahren könnte mit einer Schleife spezifiziert werden. Innerhalb der Schleife würde eine Aktivität gestartet, die eine Mahnung versendet, der Stichtag für das Versenden der nächsten Mahnung ermittelt und der Zähler für die bereits versendeten Mahnungen inkrementiert.

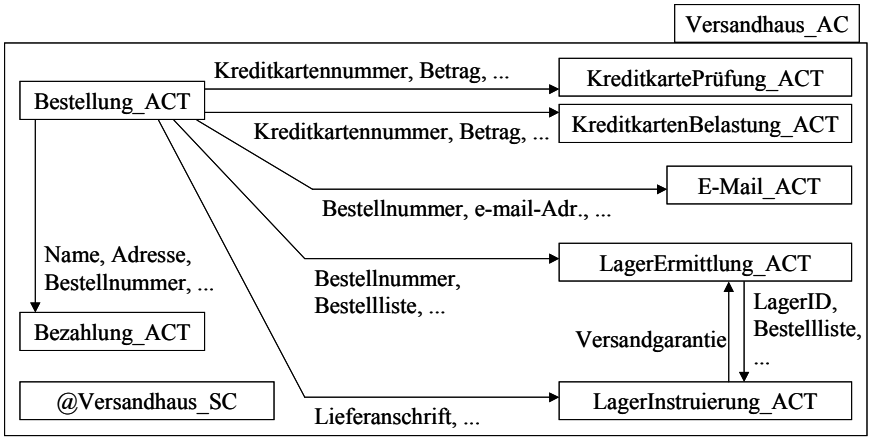
**Datenfluß.** Alle Eingabedaten werden in der Aktivität gesammelt, die die Bestellung erfaßt, und von ihr (bedarfsweise) an die übrigen Aktivitäten verteilt. Es ergibt sich ein sternförmiger Datenfluß, ausgehend von dieser Aktivität. Lediglich innerhalb der Schleife, die die Artikel auf die Lager verteilt, gibt es zusätzlichen Datenfluß, nämlich die jeweiligen Artikellisten in die eine und die Versandgarantien in die andere Richtung.

**Spezifikation als State– und Activity-Chart** (ohne das Mahnverfahren). Im State-Chart wird der Übersichtlichkeit halber auf die Darstellung von Ausnahmebehandlungen verzichtet (z.B. Nichtlieferbarkeit bestellter Artikel). Initial wird die Aktivität *BestellungErfassen* gestartet (hier explizit mit der Aktion st!(Bestellung_ACT)) und der Workflow geht in den Zustand *NeueBestellung_S* über. Die Bedingung *KreditkartenZahlung* spezifiziert die gewünschte Zahlungsmodalität und wird von der Aktivität *BestellungErfassen* gesetzt. Ist die Bedingung wahr, wird die Aktivität *KreditkartePrüfen* gestartet, die bei ihrer Terminierung die Bedingung *KreditkarteOk* auf wahr oder falsch setzt. Schlägt der Test fehl, wird der Workflow beendet, ansonsten geht er in den geschachtelten Zustand *Versand_S* über. Innerhalb dieses geschachtelten Zustandes werden zwei Sub–Workflows ausgeführt. Die Parallelität der Ausführung wird durch die gestrichelte Linie spezifiziert. In den Sub–Workflows werden die Aktivitäten *Bestätigen*, *LagerErmitteln* und *LagerInstruieren* gestartet. Sie setzen jeweils die Bedingungen *BestätigungGesendet*, *LagerErmittelt* und *VersandGarantiert* auf wahr. Dabei durchlaufen die Sub–Workflows die Zustände *Bestätigen_S*, *LagerErmitteln_S* und *LagerInstruieren_S* und terminieren in den Zuständen *BestätigungGesendet_S* beziehungsweise *Abgeschickt_S*.

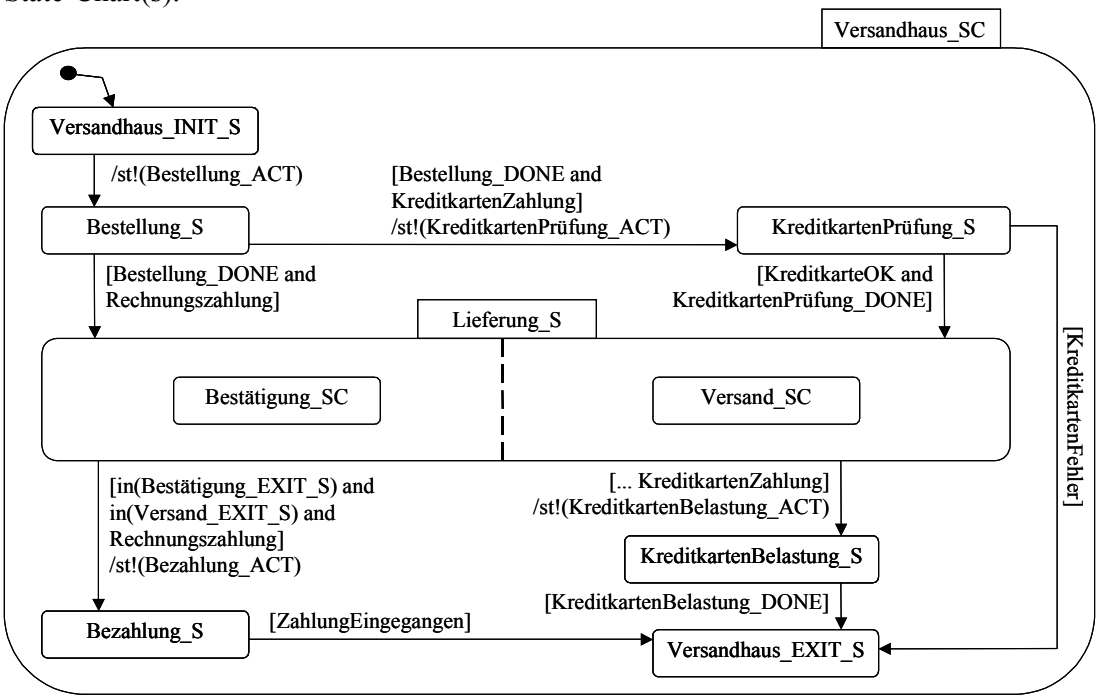
Haben beide Sub–Workflows ihren Endzustand erreicht, angezeigt durch die Bedingungen *in(Abgeschickt_S)* und *in(BestätigungGesendet_S)*, wird der geschachtelte Zustand *Versand_S* verlassen. Abhängig vom Wert der Bedingung *KreditkartenZahlung* wird die Kreditkarte durch Starten der Aktivität *KreditkarteBelasten* belastet und der Workflow endet im Zustand *Ende_S,* oder die Aktivität *Bezahlung* wird gestartet und der Workflow geht in den Zustand *Bezahlung_S* über. Bei der Aktivität *Bezahlung* wird die Bedingung *ZahlungEingegangen* auf wahr gesetzt, sobald die Rechnung beglichen wurde. Dadurch geht der Workflow in den Zustand *Ende_S* über.

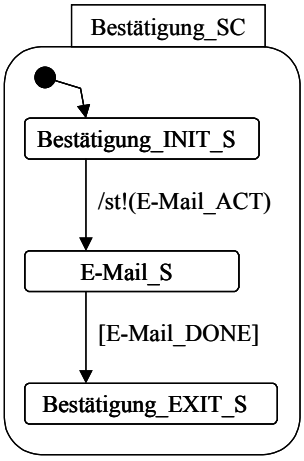
Das Activity–Chart enthält alle an dem Workflow beteiligten Aktivitäten. Alle Eingabedaten werden in der Aktivität *BestellungErfassen* gesammelt und von ihr bedarfsweise an die übrigen Aktivitäten verteilt. Zusätzlich übermittelt die Aktivität *LagerErmitteln* der Aktivität *LagerInstruieren* eine *LagerID* mit der zugehörigen Artikelliste. Die Aktivität *LagerInstruieren* schickt an ihrem Ende der Aktivität *LagerErmitteln* die erhaltenen Versandgarantien zurück.

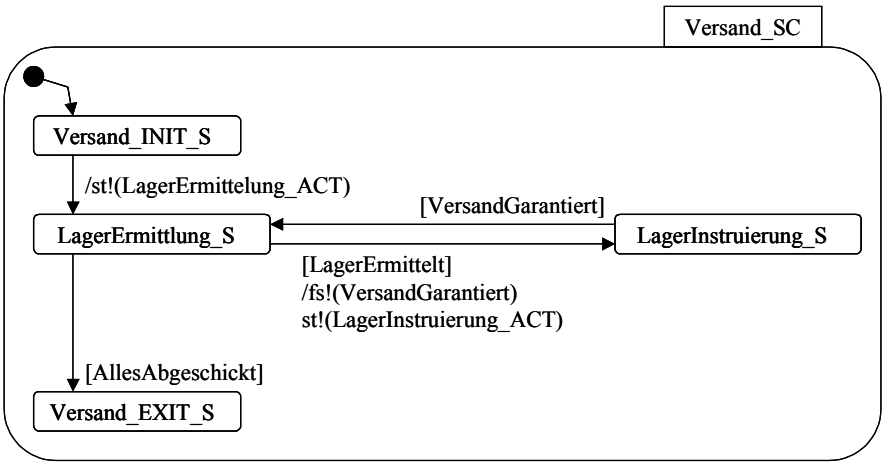
Activity-Chart:



State-Chart(s):



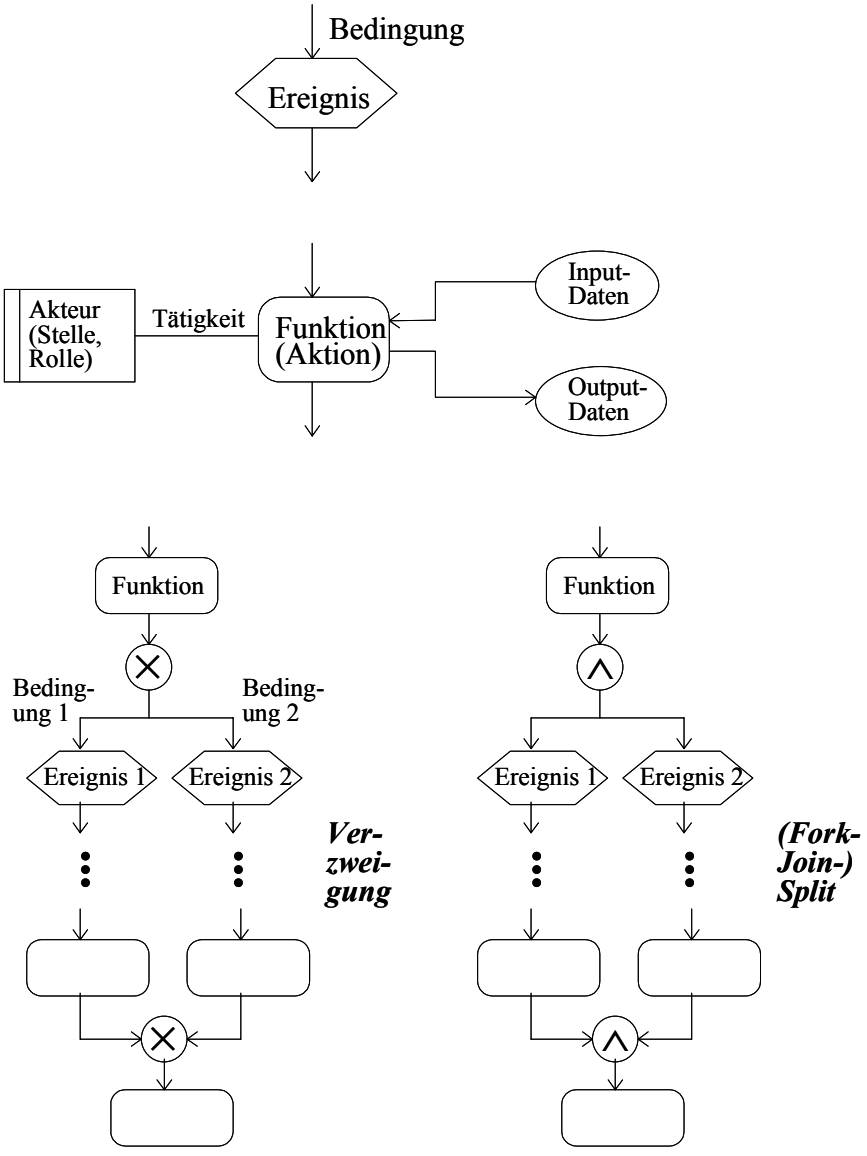



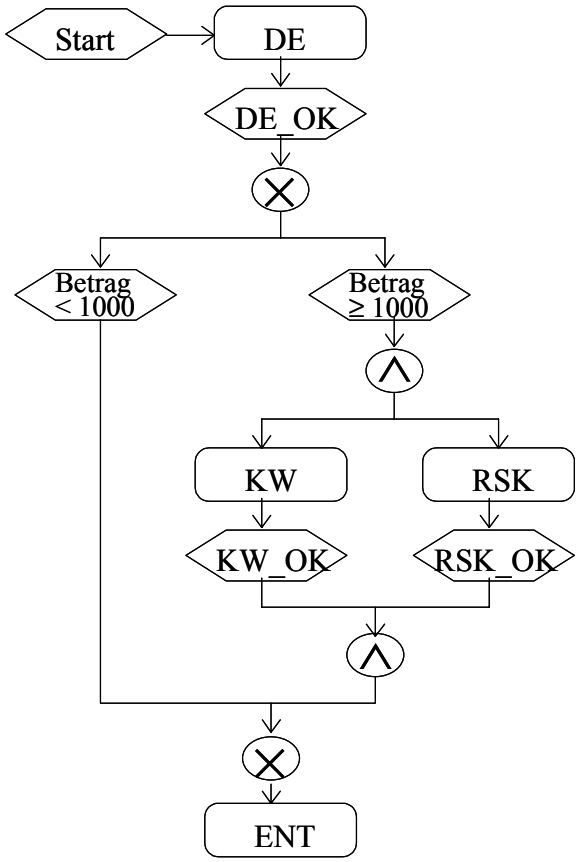
### 11.1.2 Prozeßmodellierung mit Ereignis-Prozeß-Ketten (EPKs)

Zur Modellierung von Geschäftsprozessen gibt es zahlreiche weitere “semiformale" oder auch stärker formalisierte Methoden, z.B. solche, die auf Varianten von Petrinetzen beruhen. Eine in der Praxis weit etablierte Methode ist die der *Ereignis-Prozeß-Ketten (Event-Process Chains), kurz: EPKs*. Dabei werden Aktivitäten mit Kontrollfluß-Konnektoren der Typen XOR (Verzweigung), AND (Parallelausführung) und OR (Nichtdeterminismus) verknüpft. Sofern die logischen Bedingungen für den Kontrollfluß, z.B. Verzweigungsbedingungen, hinreichend präzise und formal spezifiziert werden, lassen sich EPKs automatisch in State-Charts übersetzen. Dies kann nützlich sein, wenn Anwender beispielsweise mit EPKs zunächst grob modellieren, dann schrittweise verfeinern und präzisieren, bis sie schließlich eine übersetzbare Spezifikation erhalten, die sie schlußendlich dem StateChart-Interpreter eines Workflow-Management-Systems zur Ausführung übergeben können. EPKs können über die Beschreibung des Kontroll- und Datenflusses hinaus auch organisatorische Zuordnungen der Aktivitäten (Funktionen) zu Abteilungen, Sachbearbeiterrollen, etc. in Form von Annotationen darstellen.

Die wichtigsten grafischen Elemente von EPKs sind im folgenden kurz zusammengestellt:



Einfaches Beispiel eines EPK-Modells (Kreditantragsprüfung) unter Weglassung von Ausnahmebehandlungen, Datenfluß, und Aspekten der Arbeitsorganisation:

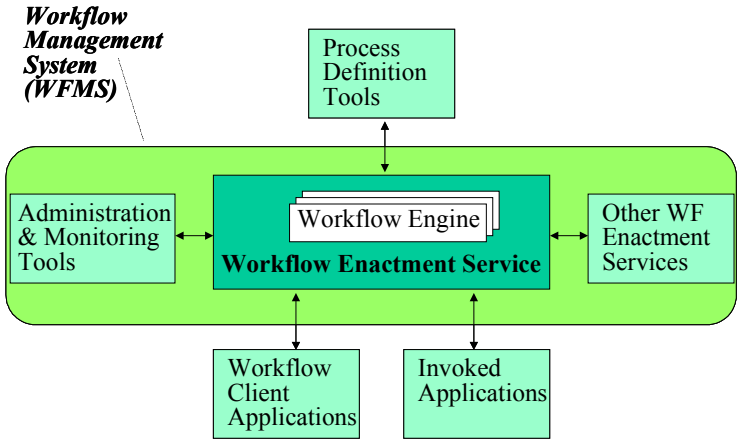


## 11.2 Workflow-Management für Geschäftsprozesse

Heute verfügbare *Workflow-Management-Systeme (WFMS)* unterstützen die automatische Steuerung von Geschäftsprozessen auf drei Ebenen:

- durch eine Methode zur *Spezifikation des Kontroll- und Datenflusses* zwischen den Aktivitäten eines Workflows,
- durch eine *Workflow-Engine zur Ausführung von Workflows* (im wesentlichen durch Interpretation der Workflow-Spezifikation) und damit systemunterstützten Koordination der Aktivitäten sowie
- durch eine Palette von *Administrationswerkzeugen* zur Simulation, Analyse und Visualisierung von Workflows sowie zur Festlegung und Überwachung von Regeln für die Zuteilung von Arbeitsschritten zu den Ausführungsorganen (dem sog. Worklist-Management zur dynamischen Rollenauflösung).

Die typische Architektur eines WFMS sieht folgendermaßen aus:




Dies ist eine Referenzarchitektur, die von der sog. Workflow Management Coalition (WfMC), einem Industriekonsortium, entwickelt wurde. Kommerzielle Produkte entsprechen mehr oder weniger diesem Framework. Führende WFMS-Produkte sind z.B. IBM MQ Series Workflow (Teil von IBM WebSphere), Staffware, MS BizTalk, HP E-Speak, BEA WebLogic, SAP Workflow, SNI WorkParty, Dialogika MultiDesk, etc.

## 11.3 Semantik von Statecharts

### 11.3.1 Operationale Semantik

Im folgenden wird eine operationale Semantik einer simplifizierten Variante von Statecharts formal entwickelt, also das dynamische Verhalten einer Statechart-Spezifikation mathematisch beschrieben. Zur Vereinfachung werden Events nicht betrachtet, und es wird angenommen, daß die einem Zustand entsprechende Aktivität jeweils mit Betreten des Zustands gestartet und mit dem Verlassen des Zustands beendet wird.

Die formale Semantik bildet die Grundlage für die Analyse und Verifikation von Prozeßspezifikationen. Dabei ist man an Invarianten des Verhaltens eines Prozesses unabhängig von seinen Eingaben interessiert. Diese Invarianten gehören in der Regel zu einer der folgenden Kategorien:

- *Sicherheitseigenschaften (safety properties):* zu jedem Zeitpunkt gelten bestimmte wünschenswerte Eigenschaften; unzulässige Prozeßzustände werden immer vermieden („es passiert nie etwas Schlimmes“).
- *Lebendigkeitseigenschaften (liveness properties):* ab einem bestimmten Zeitpunkt wird eine wünschenswerte Eigenschaft erreicht; Zielzustände des Prozesses werden irgendwann erreicht („es passiert schlußendlich etwas Gutes“).

Im Konferenzreiseplanungsbeispiel etwa wäre das Nichtüberschreiten des Budgets eine Sicherheitseigenschaft und das Terminieren des Entscheidungsprozesses (also Erreichen eines Endzustands) eine Lebendigkeitseigenschaft.

**Definition:**

Ein ***Statechart*** ist ein Tripel (S, V, T), wobei

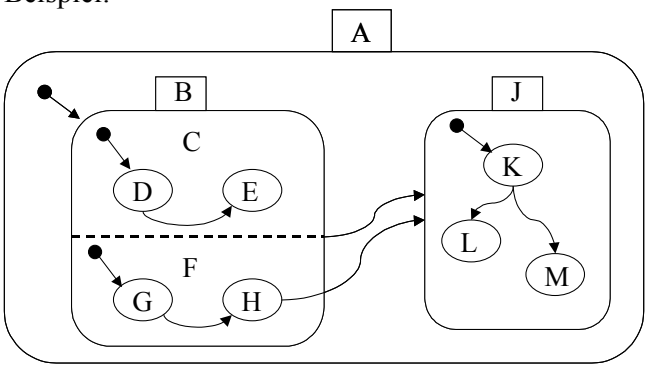
- S eine endliche Menge von ***Zuständen*** ist,
- V eine endliche Menge von typisierten ***Variablen*** ist (mit den Typen Boolean, Integer, etc.),
- T eine endliche Menge von ***Transitionen*** ist, wobei jede Transition ein Paar von Zuständen aus S verbindet oder eine initiale Transition in einen Zustand ist und mit einer Condition-Action-Regel der Form [C] / A annotiert ist. Für $t \in T$ bezeichnet source(t) den Ausgangszustand und target(t) den Zielzustand. Bei einer ***initialen Transition*** ist source(t) undefiniert.

In einer ***Condition-Action-Regel*** [C] / A ist

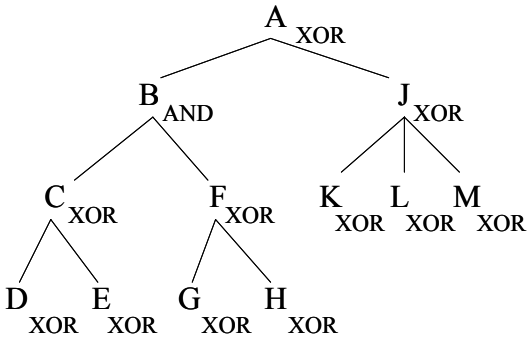
- C eine aussagenlogische Formel (mit den Junktoren and, or, not) über elementaren Vergleichen der Form Var1 θ Wert oder Var1 θ Var2 (Var1, Var2 $\in$ V) mit einem Vergleichsoperator θ aus $\{=, \neq, <, >, \leq, \geq\}$ oder der Form in(s) mit einem Zustand $s \in S$ und
- A eine Liste von Zuweisungen der Form Var := expr (Var $\in$ V) mit einem – je nach Typ von Var – arithmetischen oder Booleschen (oder sonstigem) Ausdruck expr.

Die Zustandsmenge S bildet einen Baum, bei dem $s1 \in S$ Vater von $s2 \in S$ ist, wenn s2 ein direkter Unterzustand von s1 ist. Ein Zustand ist vom Typ AND, wenn seine Kinder orthogonale Komponenten sind, ansonsten vom Typ XOR. Für jeden XOR-Zustand wird verlangt, daß unter seinen Kindern genau eines Ziel einer initialen Transition ist.

Beispiel:




Zustandsbaum:




**Definition:**

Ein ***Ausführungszustand*** states eines Statecharts ist eine Teilmenge von S, die Menge der aktuell betretenen Zustände. Dabei muß gelten:

- Die Wurzel von S ist in jedem Ausführungszustand states enthalten.
- Wenn s in states enthalten ist und s ist vom Typ AND, dann sind alle Kinder von s in states.
- Wenn s in states enthalten ist und s ist vom Typ XOR, dann ist genau ein Kind von s in states.

Ein ***Ausführungskontext*** eines Statecharts ist eine Wertebelegung der Variablen in V, d.h. eine (typgerechte) Abbildung val: V $\rightarrow$ Dom, wobei Dom die Menge aller möglichen Werte aller vorgesehenen Typen ist.

Eine (Ausführungs-) ***Konfiguration*** conf eines Statecharts ist ein Paar (states, val) von Ausführungszustand und Ausführungskontext.

Die initiale Konfiguration $conf_0$ hat die Zustandsmenge states, die

- die Wurzel von S enthält,
- für jeden AND-Zustand s in $conf_0$ alle Kinder von s und
- für jeden XOR-Zustand s in $conf_0$ das Kind von s, das Ziel einer initalen Transition ist,

und den Ausführungskontext mit val(Var1) = false für alle Var1 $\in$ V vom Typ Boolean und val(Var2) = 0 für alle Var2 $\in$ V vom Typ Integer usw.

**Definition:**

Die Auswertung eval(expr, conf) eines Ausdrucks expr in einer Konfiguration conf = (states, val) ist induktiv wie folgt definiert:

- eval (c, conf) = c für alle Konstanten c
- eval (not expr, conf) = not eval(expr, conf),
- eval (expr1 and expr2, conf) = eval(expr1, conf) and eval(expr2, conf); analog für or
- eval (expr1 $\theta$ expr2, conf) = eval(expr1, conf) $\theta$ eval(expr2, conf)
- eval (in(s), conf) = true falls s $\in$ states, false sonst

Die Wirkung einer Zuweisung Var1 := expr im Kontext val ist eine Kontextveränderung mit neuem Kontext val', so daß val'(Var1) = eval(expr, conf) und val'(Var) = val(Var) für alle anderen Variablen Var.

**Definition:**

Sei conf = (states, val) eine Konfiguration eines Statecharts (S,V,T), und sei t eine Transition mit Annotation [cond] / action.

t ***kann feuern***, wenn source(t) $\in$ states und eval(cond, conf) = true ist. Die Menge aller Transitionen, die in Konfiguration conf feuern können, wird mit fire(conf) bezeichnet.

Sei a = lca(source(t), target(t)) der kleinste gemeinsame Vorfahre von source(t) und target(t) im Zustandsbaum von S, und seien src(t) und tgt(t) diejenigen Kinder von a, die in den Teilbäumen source(t) bzw. target(t) liegen. Die Mengen source*(t) und target*(t) der durch das Feuern von t ***verlassenen*** bzw. ***neu betretenen Zustände*** sind wie folgt induktiv definiert:

- src(t) ist in source*(t)
- falls s in source*(t), dann sind auch alle Kinder von s in source*(t)
- tgt(t) und target(t) sind in target*(t)
- falls s in target*(t) und vom Typ AND ist, dann sind auch alle Kinder von s in target*(t)
- falls s in target*(t) und vom Typ XOR ist und nicht Vater von target(t) ist, dann ist das Kind von s, das Ziel einer initialen Transition ist, auch in target*(t)

Eine ***Folgekonfiguration*** conf' = (states', val') von conf entsteht durch (nichtdeterministische) Auswahl einer Transition t aus fire(conf) wie folgt:

- states' = states – source*(t) $\cup$ target*(t)
- val' unterscheidet sich von val dadurch, daß alle Wirkungen der Zuweisungen in action berücksichtigt werden.

Die operationale Semantik eines Statecharts (S,V,T) ist die Menge seiner möglichen Ausführungen mit Konfigurationen $conf_0$, $conf_1$, $conf_2$, ..., wobei $conf_0$ die initiale Konfiguration ist und $conf_{i+1}$ eine Folgekonfiguration von $conf_i$ sein muß.

### 11.3.2 Abbildung von Statecharts auf endliche Automaten

Ein alternativer und komplementärer Ansatz, die Semantik von Statecharts formal zu definieren, ist die Abbildung auf endliche Automaten. Endliche Automaten sind ein Standardmodell der Informatik mit wohldefinierter Semantik und einer reichen Theorie; insbesondere im Hinblick auf Verifikationsfragen ist die Automatentheorie ein nützliches Instrument.

**Definition:**

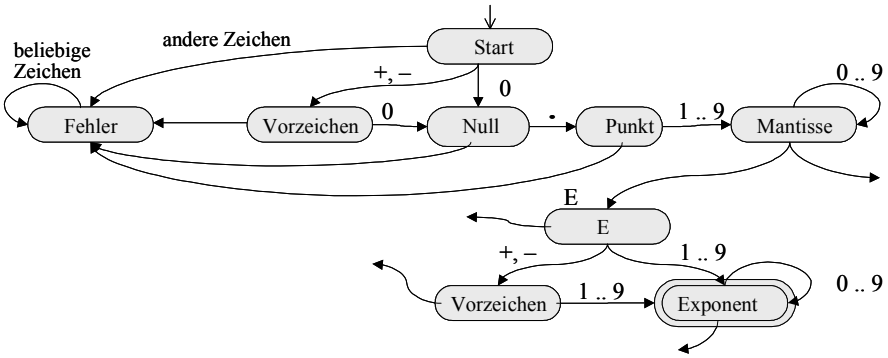
Ein endlicher Automat (finite state automaton) ist ein 5-Tupel M = (Z, $\Sigma$, $\delta$, z0, E) mit

- einer endlichen Zustandsmenge Z
- einem Alphabet (d.h. einer endlichen Menge von Zeichen) $\Sigma$
- einer Transitionsfunktion $\delta$: $Z \times \Sigma \rightarrow Z$
- einem Startzustand z0
- einer Menge von Endzuständen $E \subseteq Z$

Wir sagen, der Automat geht im Zustand $z \in Z$ mit dem Lesen des Eingabezeichens $x \in \Sigma$ in den Zustand $\delta(z,x) \in Z$ über. Die Transitionsfunktion $\delta$ wird von einzelnen Zeichen homomorph auf ganze Zeichenketten $w \in \Sigma^*$ zur Funktion $\delta^*$: $Z \times \Sigma^* \rightarrow Z$ erweitert: $\delta^*(z, au) = \delta^*(\delta(z,a),u)$ mit $z \in Z$, $a \in \Sigma$, $u \in \Sigma^*$. Die Menge $L(M) = \{w \in \Sigma^* \mid \delta^*(z0,w) \in E\} \subseteq \Sigma^*$ ist die vom Automat M akzeptierte Sprache.

Endliche Automaten sind ein in der Informatik weit verbreitetes Modellierungs- und Spezifikationsinstrument.

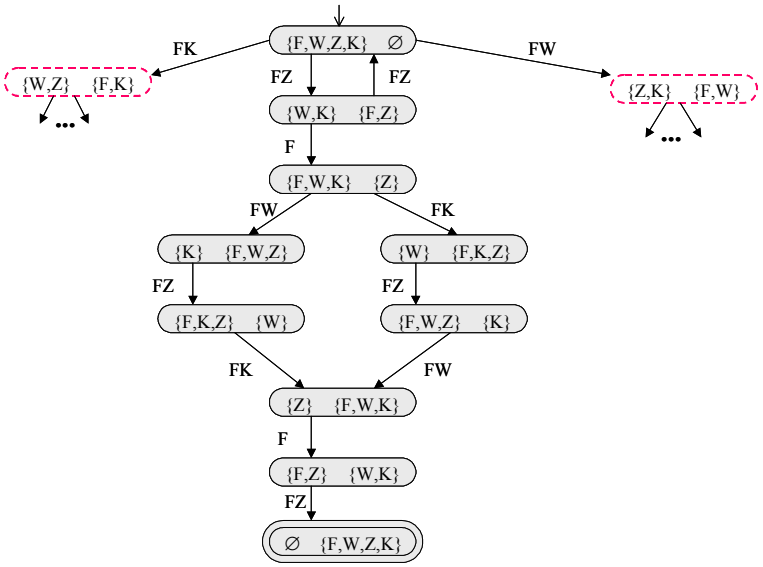
Beispiel 1:
Wir können die Menge der Gleitkommazahlen – in korrekter wissenschaftlicher Notation, also mit normalisierter Mantisse und Exponent – durch einen Automaten beschreiben. Beispielsweise sind 0.901E-42, -0.123E+1 und +0.909E10 korrekte Angaben, die der Automat akzeptieren soll, und 0.011E9, 15E-4, AB12XYZ sind inkorrekte Eingaben, für die der Automat in einen Fehlerzustand übergehen soll. Im folgenden Automatendiagramm ist der Anfangszustand durch einen kleinen Pfeil dargestellt und Endzustände sind doppelt umrandet. Zur Übersichtlichkeit sind nicht alle Übergänge in den Fehlerzustand angegeben.




Allgemeiner können beliebige sog. reguläre Sprachen durch endliche Automaten spezifiziert werden, und der einer solchen Sprache entsprechende Automat kann verwendet werden, um ein Eingabewort auf Zugehörigkeit zur Sprache zu testen. Dies hat Anwendungen u.a. bei der Erkennung von Mustern, wie man sie etwa im bekannten Unix-Tool grep angeben kann.

Beispiel 2:
Wir können die bekannte Knobelei Wolf-Ziege-Kohl durch einen endlichen Automaten modellieren. Bei dieser Knobelei muss ein Fährmann (F) Wolf (W), Ziege (Z) und Kohl (K) von einem Ufer zum anderen übersetzen, kann aber immer nur eines dieser drei „Objekte“ auf einmal mitnehmen. Erschwerend dabei ist, dass die Ziege den Kohl und der Wolf die Ziege fressen würde, wenn sie ohne Aufsicht gelassen werden. Die Zustände des Automaten modellieren jeweils Paare (L,R), wobei L angibt, welche „Objekte“ am linken Ufer sind und R, welche am rechten Ufer sind. Die Transitionen entsprechen also dem Übersetzen von einem Ufer ans andere; die Transitionsbeschriftungen geben an, welche zwei der insgesamt vier „Objekte“ jeweils übersetzen. Die Transitionsbeschriftungen haben hier aber nur mehr oder weniger Kommentarcharakter; es gibt es keine eigentlichen Eingabewörter, vielmehr kommt es nur auf die Zustandsfolge vom Startzustand zum Endzustand an (man könnte z.B. eine triviale Eingabe wählen und einfach alle Transitionen mit demselben Zeichen eines einelementigen Alphabets beschriften). Die Abbildung des Automaten lässt „uninteressante“, weil nicht zum Ziel führende, Zustände und Übergänge weg.

Allgemeiner spielen endliche Automaten in der Modellierung und Analyse kombinatorischer Spiele eine wichtige Rolle.




Bei der **Abbildung von Statecharts auf endliche Automaten** müssen drei Aspekte betrachtet werden:

1) die Abstraktion von Variablen mit unendlichen Wertebereichen (z.B. ganze Zahlen) durch einen endlichen Zustandsraum,

2) die Abbildung der aktuell aktiven Zustandsmenge eines Statecharts auf einen Zustand eines Automaten,

3) die Codierung der Übergangsannotationen (d.h. der ECA-Regeln) eines Statecharts in Zustände eines Automaten.

*Zu Aspekt 1:*

Die Aspekte 2 und 3 sind eher technischer Natur, während Aspekt 1 fundamentalen Charakter hat: offensichtlich können unendliche Strukturen nicht ohne weiteres auf endliche Strukturen ohne Informationsverlust abgebildet werden. Wir lösen dies, indem wir jede Bedingung, die sich auf Variablen mit unendlichem Wertebereich bezieht, durch eine Boolesche Variable ersetzen; die Variable ist auf True gesetzt, wenn die Originalbedingung wahr ist. Zuweisungen zu den Originalvariablen mit unendlichem Wertebereich werden in dieser Abstraktion durch Zuweisungen zu den Booleschen Variablen ersetzt; wenn dabei offen ist, ob die Boolesche Variable den Wert True oder False erhalten soll (z.B. weil die Originalbedingungen so komplex sind, dass sich dies gar nicht entscheiden lässt, oder weil der Wahrheitswert von Eingabeparametern abhängt), müssen beide Varianten berücksichtigt werden.

Im Reiseplanungsbeispiel von Abschnitt 11.1 etwa wird die Bedingung Cost > Budget durch eine Boolesche Variable BudgetOK ersetzt. Nach Zuweisungen wie Cost := ConfFee + TravelCost müssen beide möglichen Wahrheitswerte für BudgetOK berücksichtigt werden (d.h. wir kennen nach einer solchen Zuweisung den Wert von BudgetOK nicht).

Formal handelt es sich bei diesem Abbildungsschritt also um eine nichtinjektive Abbildung $\psi 1$: val $\rightarrow B1 \times B2 \times ... \times Bm$ des Ausführungskontextes eines Statecharts auf eine endliche Menge Boolescher Variablen B1, ..., Bm.

*Zu Aspekt 2:*

In einem Statechart mit Zustandsmenge S können wegen der Hierarchisierung einerseits und vor allem wegen der orthogonalen Komponenten andererseits mehrere Zustände gleichzeitig aktiv sein; in einem endlichen Automaten ist zu einer Zeit jeweils nur ein Zustand betreten.

Die Abbildung des Statecharts auf endlichen Automaten mit Zustandsmenge Z muss also Teilmengen von S „codieren“.

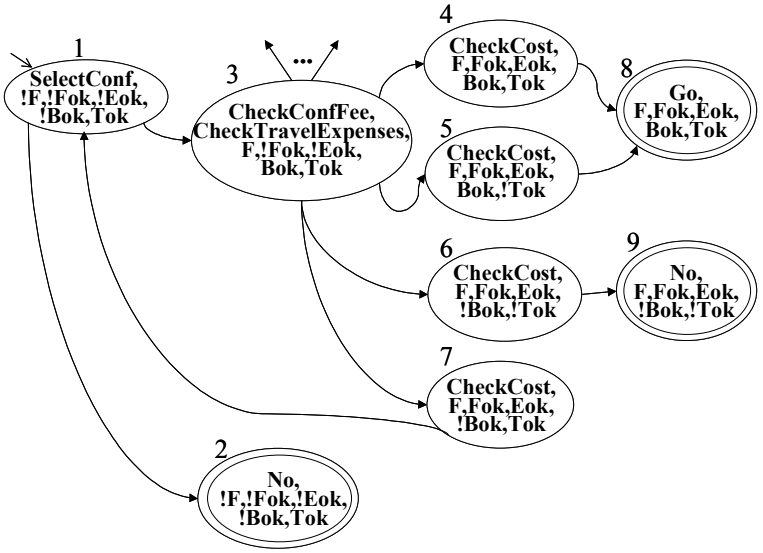
Im Reiseplanungsbeispiel von Abschnitt 11.1 etwa ist die Zustandsmenge {CheckConfFee, SelectTutorials, CheckTravelExpenses, CheckAirfare, CheckHotel} ein einziger Zustand des entsprechenden Automaten.

Formal entspricht diese Art der Zustandscodierung einer Abbildung $\psi 2$: states $\rightarrow 2^S =: Z$ der Ausführungszustände des Statecharts auf Zustände eines „Potenzmengenautomaten“. Wenn im Statechart Zustände s1, ..., sk aktiv sind, ist der entsprechende Automat gerade im Zustand $\{s1, ..., sk\} \in Z$. Startzustand des Automaten ist der Zustand $\{sj \mid sj \in conf_0.states\}$, wobei $conf_0$ die initiale Konfiguration des Statecharts ist. Man sieht sofort, dass der überwiegende Teil der Zustände in Z nie erreichbar ist und bei weiteren Analysen des Automaten weggelassen werden kann.

*Zu Aspekt 3:*

Nach der Abstraktion zu Aspekt 1 sind alle Bedingungen in den Transitionsannotationen eines Statecharts aussagenlogische Formeln über Booleschen Variablen B1, ..., Bm und die Aktionen sind Zuweisungen zu den Booleschen Variablen. Um dies auf einen einfachen Automaten abzubilden, wird die Wertebelegung der Booleschen Variablen in den Zustandsraum des Automaten hineincodiert. Formal entspricht dies einer injektiven Abbildung $\psi 3$: $Z \times B1 \times B2 \times ... \times Bm \rightarrow Z'$, wobei Z' den so erweiterten Zustandsraum des Automaten bezeichnet. Wir erzeugen dann im Automaten eine – unbeschriftete – Transition von Zustand (z, b1, ..., bm) nach (z', b1', ..., bm'), wenn es im Statechart eine Transition von dem z entsprechenden Ausführungszustand mit Transitionsannotation [cond] / action in den z' entsprechenden Ausführungszustand gibt, so dass cond bei Auswertung mit der Variablenbelegung b1, ..., bm wahr wird und action die Variablenbelegung b1', ..., bm' erzeugt.

Ein Ausschnitt des aus diesen drei Abbildungsschritten resultierenden Automaten für eine vereinfachte Variante des Reiseplanungsbeispiels von Abschnitt 11.1 sieht wie folgt aus. Gegenüber dem vollständigen Statechart werden dabei die Zustände CheckConfFee und CheckTravelExpenses nicht weiter verfeinert. Die Bedingung Cost > Budget wird zur Booleschen Variablen Bok abstrahiert und die Bedingung Trials < 3 zu Tok.




Insgesamt repräsentiert der aus einem Statechart abgeleitete Automat die Menge der möglichen Konfigurationen bei der Ausführung des Statecharts, so dass die Zustandsübergänge den möglichen Übergängen von einer Konfiguration zu einer Folgekonfiguration entsprechen.

## 11.4 Eigenschaften von Statecharts und deren Verifikation

Die im Hinblick auf verlässliche Informationsdienste kritischen Eigenschaften des Verhaltens von Statecharts und verwandten Prozessspezifikationen, die wir formal verifizieren sollten, können in zwei Kategorien eingeteilt werden:

**Sicherheitseigenschaften (Safety Properties):** logische Invarianten, die in jedem Zustand während der Prozessausführung gelten sollen. Im Reiseplanungsbeispiel von Abschnitt 11.1 beispielsweise wäre die Forderung, dass die Reisekosten das vorgesehene Budget nicht überschreiten dürfen, eine Sicherheitseigenschaft. Intuitiv kann man Sicherheitseigenschaften auch mit der Forderung gleichsetzen, dass bestimmte unerwünschte Konfigurationen (in denen die Sicherheitsinvariante verletzt wäre) während der Prozessausführung nicht betreten werden.

**Lebendigkeitseigenschaften (Liveness Properties):** logische Bedingungen, die bei einer Prozessausführung schlussendlich gelten sollen und damit den Fortschritt des Prozesses (Termination, Fairness, etc.) sicherstellen. Im Reiseplanungsbeispiel von Abschnitt 11.1 beispielsweise wäre die Forderung, dass der Reiseantrag schlussendlich entschieden wird, der Prozess also in einem der Zustände Go oder No terminiert, eine Lebendigkeitseigenschaft. Intuitiv kann man Lebendigkeitseigenschaften auch mit der Forderung gleichsetzen, dass bestimmte erwünschte Konfigurationen während der Prozessausführung irgendwann definitiv erreicht werden.

Um Eigenschaften dieser Art, also Bedingungen über das dynamische Verhalten eines Prozesses, formal ausdrücken zu können, wird eine erweiterte Art von Logik benötigt, die sog. **temporalen Logiken**. Der temporale Aspekt bezieht sich dabei in der Regel auf Ausführungsschritte eines Programms, Prozesses oder allgemeiner Transitionssystems, also eine abstrakte und diskrete Zeit. Temporallogiken zählen zu der Familie der sog. **Modallogiken**, bei denen der Wahrheitswert von logischen Aussagen relativ zu den Umständen (Modalitäten) dieser Aussagen verstanden wird. Beispielsweise können Aussagen wie „es regnet“ von Ort und Zeit der Aussage abhängen, und Aussagen wie „Gott existiert“ oder „das Weltall wird in seiner Expansion irgendwann stoppen und danach kollabieren“ hängen vom subjektiven Glauben oder dem relativen Wissensstand der jeweiligen Person ab. Ungeachtet der relativen Wahrheit solcher Aussagen kann man aber streng logische Schlüsse über die Konsequenzen der Aussagen führen. Für die Präzisierung von Programm- bzw. Prozessausführungseigenschaften haben sich vor allem die temporale Logik CTL (Computational Tree Logic) sowie deren Verallgemeinerungen (z.B. CTL*) bewährt.

### 11.4.1 Die Temporale Logik CTL

Die temporale Logik CTL (Computational Tree Logic) betrachtet mögliche Ausführungspfade eines Programms bzw. Prozesses, also Folgen von aufeinanderfolgenden Programm- bzw. Prozesszuständen (z.B. Konfigurationen einer Statechart-Ausführung). Das gedankliche Verfolgen aller möglichen Zustandsfolgen führt auf einen Baum von Zuständen, den Berechnungsbaum, der bei dem Namen CTL Pate gestanden hat. Diese Art von Temporallogik wird auch als „Branching Time Logic“ bezeichnet.

CTL umfasst **aussagenlogische Formeln**, die sich auf Zustände einer Programmausführung beziehen. Diese Formeln können quantifiziert werden – mit einem **Existenzquantor E** oder einem **Allquantor A** -, wobei sich die Quantifizierung auf Ausführungspfade bezieht: für einen Pfad oder für alle Pfade, die von einem bestimmten Zustand ausgehen, soll eine bestimmte Formel gültig sein. In den meisten Fällen ist der Zustand, von dem aus die möglichen Ausführungspfade betrachtet werden, der initiale Zustand des gesamten

Ausführungspfade betrachtet werden, der initiale Zustand des gesamten Programms oder Prozesses.

Die beiden Quantoren E und A werden stets mit einem der folgenden **Modaloperatoren X (Next), G (Globally), F (Finally)** kombiniert. Dabei bedeutet:

EX P: eine Formel P soll in (mindestens) einem Nachfolgerzutand gültig sein.

AX P: eine Formel P soll in allen möglichen Nachfolgerzuständen gültig sein.

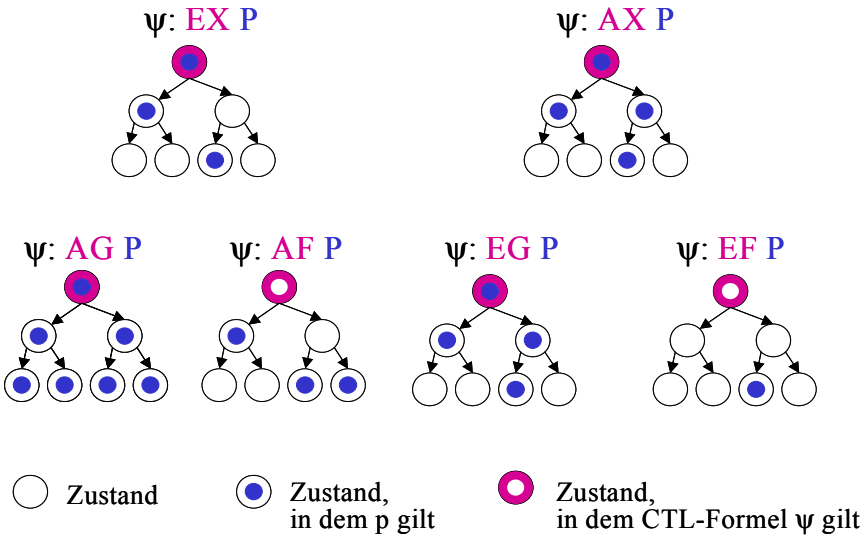
EG P: eine Formel P soll entlang (mindestens) eines Pfades global, d.h. in allen Zuständen des Pfades, gültig sein.

AG P: eine Formel P soll entlang aller möglichen Pfade global, d.h. in allen Zuständen der jeweiligen Pfade, gültig sein.

EF P: eine Formel P soll entlang (mindestens) eines Pfades irgendwann, d.h. in einem der Zustände des Pfades, gültig sein.

AF P: eine Formel P soll entlang aller möglichen Pfade irgendwann gültig sein.

Die Betrachtung der Wahrheitswerte aussagenlogischer Formeln in Zuständen und entlang von Pfaden eines Ausführungsbaums generell und die verschiedenen Kombinationen von Quantor und Modaloperator werden in der folgenden Abbildung visualisiert.




**Definition:**
Eine atomare CTL-Formel ist eine aussagenlogische Formel über elementaren Aussagen (bzw. Booleschen Variablen), inklusive Aussagen der Form „in(s)“, wobei s ein Zustand eines Transitionssystems ist.
Die Menge der in CTL erlaubten Formeln ist induktiv wie folgt definiert:

(i) Jede atomare CTL-Formel ist eine Formel.

(ii) Wenn P und Q Formeln sind, dann sind auch EX (P), AX (P), EG (P), AG (P), EF (P), AF (P), (P), $\neg$P, P$\wedge$Q, P$\vee$Q, P$\Rightarrow$Q und P$\Leftrightarrow$Q Formeln.

Formeln mit einer Quantor/Modalität-Kombination können also auch geschachtelt werden. Dabei dürfen jedoch Quantoren und Modaloperatoren immer nur in den genannten Kombinationen auftreten, niemals alleine. Diese Einschränkung fällt in der expressiveren Temporallogik CTL* weg, dafür hat diese aber eine höhere Komplexität.

Beispiele:

1) Die Sicherheitseigenschaft für das Reiseplanungsbeispiel, dass das Budget nicht überschritten werden darf, kann wie folgt in CTL formalisiert werden:
   AG ($\neg$Bok $\Rightarrow$ $\neg$in(Go))
   oder alternativ (äquivalent)
   $\neg$EF ($\neg$Bok $\wedge$ in(Go))
2) Die Lebendigkeitseigenschaft für das Reiseplanungsbeispiel, dass der Prozess in einem der Zustände Go oder No terminieren muss, kann in CTL folgendermaßen formalisiert werden:
   AF (in(Go) $\vee$ in(No))
3) Die Eigenschaft, dass wir nach einem ersten wegen Budgetüberschreitung gescheiterten Vorschlag immer noch die Reise genehmigen können, ist in CTL wie folgt ausdrückbar:
   EF ( (in(CheckCost) and !Bok) => ( EF (in(Go)) ) )

**Definition:**
Gegeben sei eine Menge P atomarer aussagenlogischer Formeln. Eine *Kripke-Struktur* M über P ist ein 4-Tupel (S, s0, R, L) mit

- einer endlichen Zustandsmenge S,
- einem Startzustand s0 $\in$ S,
- einer binären Transitionsrelation R $\subseteq$ S $\times$ S,
- einer Funktion L: S $\rightarrow$ $2^P$, die jedem Zustand die Menge der in dem Zustand wahren atomaren Aussagen zuordnet.

Die *Interpretation* $\psi$ einer CTL-Formel F mit atomaren Aussagen P ist eine Abbildung auf eine Kripke-Struktur M=(S, s0, R, L) über atomaren Aussagen P, so dass die Wahrheitswerte von Teilformeln p bzw. p1, p2 von F in den Zuständen s von M, in Zeichen: M,s |= p, wie folgt festgelegt sind:

(i) M,s |= p mit einer aussagenlogischen Formel p gilt genau dann, wenn p $\in$ L(s);
(ii) M,s |= $\neg$p genau dann, wenn nicht M,s |= p gilt;
(iii) M,s |= p1 $\wedge$ p2 genau dann, wenn M,s |= p1 und M,s |= p2;
(iv) M,s |= p1 $\vee$ p2 genau dann, wenn M,s |= p1 oder M,s |= p2;
(v) M,s |= EX p genau dann, wenn es t$\in$ S gibt mit (s,t)$\in$ R und M,t |= p;
(vi) M,s |= AX p genau dann, wenn für alle t$\in$ S mit (s,t)$\in$ R gilt: M,t |= p;
(vii) M,s |= EG p genau dann, wenn es t1, ..., tk $\in$ S gibt mit t1=s, (ti, ti+1)$\in$ R für alle i und tk=tj für ein j:1$\leq$j<k oder tk ohne Nachfolger bzgl. R, so dass M,ti |= p für alle i;
(viii) M,s |= AG p genau dann, wenn für alle t$\in$ S mit (s,t)$\in$ R+ gilt: M,t |= p;
(ix) M,s |= EF p genau dann, wenn es t$\in$ S gibt mit (s,t)$\in$ R+ und M,t |= p;
(x) M,s |= AF p genau dann, wenn es für alle t$\in$ S mit (s,t)$\in$ R+ einen Zustand t'$\in$ S gibt mit a) (t,t')$\in$ R+ oder b) (s,t')$\in$ R+ und (t',t)$\in$ R+, so dass M,t' |= p gilt.

Eine Kripke-Struktur M = (S, s0, R, L) ist ein *Modell* einer CTL-Formel F, wenn M,s0 |= F.
Eine Formel heißt erfüllbar, wenn sie mindestens ein Modell hat, ansonsten unerfüllbar. Eine Formel F heißt allgemeingültig (oder Tautologie), wenn jede Kripke-Struktur über den atomaren Aussagen von F ein Modell von F ist.

### 11.4.2 Verifikation durch Modellprüfen (Model Checking)

Deduktionsverfahren für CTL haben sehr hohen Rechenaufwand. Insbesondere das Testen einer CTL-Formel auf Allgemeingültigkeit, also das Theorembeweisen in einem CTL-Kalkül, ist für die meisten praktischen Belange zu aufwändig (mindestens exponentiell in der Formelgröße). Dieses Vorgehen ist jedoch zum Verifizieren oder Falsifizieren kritischer Eigenschaften von Transitionssystemen weder angemessen noch notwendig. Vielmehr genügt es zu testen, ob das vorliegende Transitionssystem ein Modell einer gegebenen CTL-Formel ist. Diese Technik nennt man Modellprüfen oder Model-Checking. Sie hat eine Laufzeit, die polynomiell in der Größe der Formel und des Transitionssystems ist.

Die Vorgehensweise beim Modellprüfen orientiert sich eng an der Definition der Interpretation einer CTL-Formel F, indem man prüft, ob das vorliegende Transitionssystem eine Kripke-Struktur ist, in der die Formel wahr ist. Der Algorithmus des Modellprüfens arbeitet induktiv über dem Aufbau der Formel F, indem er bottom-up für jede Teilformel q von F die Menge der Zustände des Transitionssystems mit markiert, in denen q gilt. Wenn dann am Ende der initiale Zustand des Transitionssystems mit F markiert ist, wissen wir, dass das Transitionssystem ein Modell von F ist.

Die Prozeduren für das Markieren der Zustände $Q \subseteq S$, in denen eine Teilformel q gilt, hängen von der Form von q ab und sehen wie folgt aus. Dabei seien p, p1, p2 Teilformeln von q, für die gemäß der strukturellen Induktion des Gesamtalgorithmus bereits markierte Zustandsmengen P, P1, P2 bestimmt wurden.

(i) q ist eine atomare Aussage (Boolesche Variable):
Markiere alle Zustände s mit $q \in L(s)$ mit q

(ii) q hat die Form $\neg p$:
Markiere S – P mit q

(iii) q hat die Form $p1 \wedge p2$:
Markiere $P1 \cap P2$ mit q

(iv) q hat die Form $p1 \vee p2$:
Markiere $P1 \cup P2$ mit q

(v) q hat die Form EX p:
Markiere alle Vorgänger von P mit q, also alle $s \in S$, für die es ein $x \in P$ gibt mit R(s,x)

(vi) q hat die Form AX p:
Markiere s mit q, wenn alle Nachfolger von s mit p markiert sind

(vii) q hat die Form EF p:
Dieser Fall wird auf die Rekursions- bzw. Fixpunktgleichung $Q = P \cup pred(Q)$ zurückgeführt. Dabei ist pred(Q) die Menge der Vorgänger der Zustände in Q.
Diese Fixpunktgleichung beruht auf der logischen Äquivalenz $EF\ p \Leftrightarrow p \vee EX\ (EF\ p)$.
Die Lösung der Fixpunktgleichung, d.h. die Berechnung des kleinsten Fixpunktes, erfolgt mit der Prozedur:

```
Q := P;
Qnew := Q ∪ pred(Q);
while not (Q = Qnew) do
   Q := Qnew;
   Qnew := Q ∪ pred(Q);
od;
```

(viii) q hat die Form EG p:

Die Äquivalenz EG p $\Leftrightarrow$ p $\wedge$ EX (EG p) weist den Weg zur iterativen Berechnung:

```
Q := P;
Qnew := Q ;
repeat
      for each s in Q do
          if s has successors and no successor of s is in Q
          then Qnew := Q - {s}; fi;
      od;
until (Q = Qnew);
```

(ix) q hat die Form AG p:

Die Äquivalenz AG p $\Leftrightarrow$ p $\wedge$ AX (AG p) weist den Weg zur iterativen Berechnung:

```
Q := P;
repeat
     Qnew := Q;
     for each s in Q do
          if s has successors and one successor of s is not in Q
          then Q := Q - {s} fi;
     od;
until (Q = Qnew);
```

Alternativ kann dieser Fall auf die Äquivalenz AG p $\Leftrightarrow \neg$ EF ($\neg$p) zurückgeführt werden.
Wir berechnen also zunächst die Zustandsmenge Q' zur Formel EF ($\neg$p) und markieren dann die Zustandsmenge S – Q' mit q.

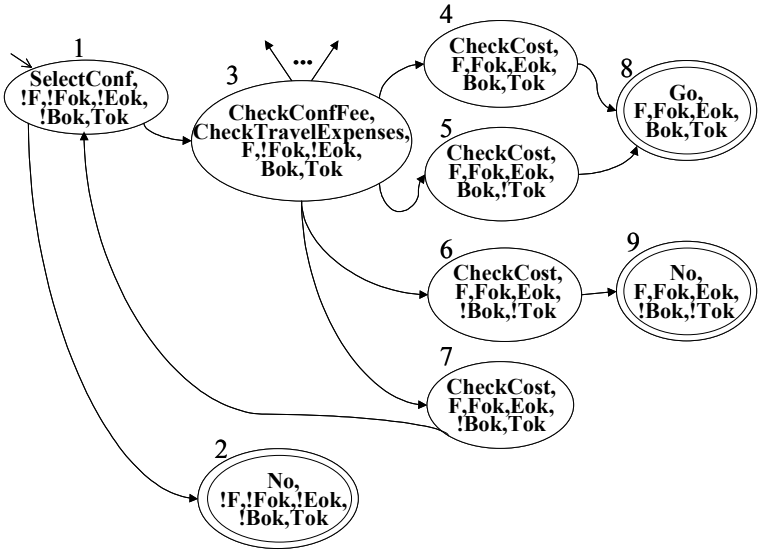
(x) q hat die Form AF p:

Die Äquivalenz AF p $\Leftrightarrow$ p $\vee$ AX (AF p) weist den Weg zur iterativen Berechnung:

```
Q := P;
repeat
     Qnew := Q;
     for each s in pred(Q) do
          if all successors of s are in Q
          then Q := Q ∪ {s}; fi;
     od;
until (Q = Qnew);
```

Alternativ kann dieser Fall auf die Äquivalenz AF p $\Leftrightarrow \neg$ EG ($\neg$p) zurückgeführt werden.
Wir berechnen also zunächst die Zustandsmenge Q' zur Formel EG ($\neg$p) und markieren dann die Zustandsmenge S – Q' mit q.

Beispiel:
Wir wollen Modellprüfen auf das Reiseplanungsbeispiel bzw. den daraus abgeleiteten endlichen Automaten anwenden. Der Automat ist hier – mit durchnummerierten Zuständen – nochmals abgebildet.




Zur Verifikation der Eigenschaft AG (¬Bok ⇒ ¬in(Go)) wird wie folgt markiert:
mit Bok die Zustände 3, 4, 5, 8
mit in(Go) der Zustand 8
mit ¬in(Go) die Zustände 1, 2, 3, 4, 5, 6, 7, 9
mit (¬Bok ⇒ ¬in(Go)) die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
mit AG (¬Bok ⇒ ¬in(Go))
  in der 0. Iteration die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
  in der 1. Iteration die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
Damit gilt die Eigenschaft im Startzustand 1.

Zur Verifikation der Eigenschaft AF (in(Go) ∨ in(No)) wird wie folgt markiert:
mit in(Go) der Zustand 8
mit in(No) der Zustand 2, 9
mit in(Go) ∨ in(No) die Zustände 2, 8, 9
mit AF (in(Go) ∨ in(No))
  in der 0. Iteration die Zustände 2, 8, 9
  in der 1. Iteration die Zustände 2, 4, 5, 8, 9
  in der 2. Iteration die Zustände 2, 4, 5, 8, 9
Damit kann die Eigenschaft für den Startzustand 1 nicht nachgewiesen werden.

Zur Verifikation der Eigenschaft EF ( (in(CheckCost) and !Bok) => ( EF (in(Go)) ) ) wird wie folgt markiert:
mit in(Go): der Zustand 8
mit EF (in(Go)): die Zustände 1, 3, 4, 5, 6, 7, 8, 9
mit not in(CheckCost) or Bok: die Zustände 1, 2, 3, 4, 5, 8, 9
mit (in(CheckCost) and !Bok) => ( EF (in(Go)): die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
mit EF ( (in(CheckCost) and !Bok) => ( EF (in(Go)) ) )
  in der 0. Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9
  in der 1. Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9
Damit gilt die Eigenschaft im Startzustand 1.

Diese grundsätzliche algorithmische Vorgehensweise kann in ihrer Effizienz noch deutlich verbessert werden, wenn das Modellprüfen statt auf dem eigentlichen Transitionssystem auf einer geeigneten symbolischen Repräsentation des Systems arbeitet; man spricht dann auch von symbolischem Modellprüfen. Die erfolgreichste Variante dieser Richtung, mit der sehr große Systeme verifiziert werden konnten, codiert die Zustände des Transitionssystems in Form von m binären Variablen, also praktisch als m-stellige Bitstrings, und die Transitionen als 2m-stellige Boolesche Funktion, deren Funktionswert 1 ist, wenn es eine mögliche Transition zwischen den entsprechenden Zuständen gibt, und 0 sonst. Diese Boolesche Funktion wiederum kann mit sog. BDDs (Binary Decision Diagrams) bzw. OBDDs (Ordered Binary Decision Diagrams) sehr kompakt repräsentiert werden, und symbolisches Modellprüfen arbeitet dann als Markierungsverfahren auf OBDDs.

## Ergänzende Literatur zu Kapitel 11:


D. Harel, M. Politi, Modeling Reactive Systems with Statecharts: the Statemate Approach, McGraw-Hill, 1998

D. Harel, E. Gery, Executable Object Modeling with Statecharts, IEEE Computer Vol.30 No.7, 1997

E.A. Emerson, Temporal and Modal Logic, in: J. van Leeuwen (Editor), Handbook of Theoretical Computer Science, Elsevier, 1990

E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999

G. Weikum, D. Wodtke, A. Kotz-Dittrich, P. Muth, J. Weißenfels, Spezifikation, Verifikation und verteilte Ausführung von Workflows in MENTOR, in: T. Härder (Hrsg.), Informatik Forschung und Entwicklung Band 12 Heft 2, Springer-Verlag, 1997, Themenheft über Workflow-Management

G. Weikum, Towards Guaranteed Quality and Dependability of Information Services, 8. GI-Fachtagung über Datenbanksysteme in Büro, Technik und Wissenschaft, Freiburg, 1999, Springer-Verlag, 1999.

Scheer, A.-W., ARIS - Business Process Modeling, 2nd Edition, Springer-Verlag, 1999

Object Management Group (OMG), Unified Modeling Language (UML) Documentation, http://www.omg.org/ oder http://www.rational.com/uml/

F. Leymann, D. Roller, Production Workflow – Concepts and Techniques, Prentice Hall, 2000

A. Dogac, L. Kalinichenko, T. Özsu, A. Sheth (Editors), Workflow Management Systems and Interoperability, Springer, 1998

G. Vossen, J. Becker (Hrsg.), Geschäftsprozeßmodelle und Workflow-Management - Modelle, Methoden, Werkzeuge, International Thomson Publishing, 1995

S. Jablonski, M. Böhm, W. Schulze (Hrsg.), Workflow-Management - Entwicklung von Anwendungen und Systemen, dpunkt-Verlag, 1997