Modelltransformation von *Statecharts*

Formale Regeln zur Übersetzung verschiedener Semantiken

Steffen Prochnow

Technische Universität Carolo-Wilhelmina zu Braunschweig
Institut für Software
Abteilung Programmierung
Prof. Dr. Ursula Goltz

Diplomarbeit

Modelltransformation von *Statecharts*

Formale Regeln zur Übersetzung verschiedener Semantiken

Steffen Prochnow

23. März 2003

Betreuung: Dipl.-Ing. Martin Mutz

Diese Arbeit wurde mit Hilfe von LaTeX und METAPOST gesetzt.
23. März 2003, 16:30 Uhr

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Diplomarbeit selbstständig verfasst und ohne Benutzung anderer als den angegebenen Hilfsmitteln angefertigt, nur die angegebenen Quellen benutzt und die in den Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Vorwort

Diese Diplomarbeit entstand am Institut für Software, Abteilung Programmierung der Technischen Universität Braunschweig während meines Studiums der Informatik. Den Rahmen für sie gab das Projekt *STEP-X*, an dem die obige Abteilung unter der Leitung von Frau Prof. Dr. U. Goltz beteiligt ist.

Frau Prof. Dr. U. Goltz danke ich dafür, mir die Mitwirkung an diesem Projekt ermöglicht zu haben, und für die engagierte Übernahme des Referats. Meinem Betreuer Dipl.-Ing. Martin Mutz danke ich für die zahlreichen fruchtbaren Gespräche. Ein besonderer Dank geht an Sylvia Moenickes.

Inhaltsverzeichnis

Tafelverzeichnis

Abbildungsverzeichnis

Verzeichnis der Definitionen

Verzeichnis der Transformationsregeln

Verzeichnis der *Flattening*-Regeln

Kapitel 1

Ziel und Aufbau

„The only way to get rid of a temptation is to yield to it.“
Oscar Wilde

1.1 Motivation

Elektronische Systeme sind stark anwendungsspezifische Mess-, Regel- und Steuersysteme, die durch intensive Interaktion mit der Umgebung und fest definierbare Aufgaben mit zeitlichen Randbedingungen gekennzeichnet sind. Mit zunehmender Komplexität und der Zahl der Aufgaben wird ihr Entwurf zunehmend anspruchsvoller. Zusätzlich wird gefordert, die Entwicklung innerhalb einer sehr kurzen Zeit zur Marktreife zu führen und zu vermeiden, dass am Endprodukt aufwändige nachträgliche Korrekturen durchgeführt werden müssen. Die schnell wachsende Komplexität der Entwurfsaufgaben ist durch den hohen Anteil an Wechselwirkungen für den Entwickler nur noch schwer nachzuvollziehen und somit gestaltet sich auch die Qualitätssicherung als zunehmendes Problem. Um Entwurfsfehler in frühen Phasen zu vermeiden, werden verstärkt formale Hilfsmittel zur Spezifikation und zur Modellierung verwendet, um in späteren Phasen auch Simulation und formale Verifikation der Modellierung zu nutzen. So wird im Projekt *STEP-X* (STrukturierter Entwicklungs-Prozess für Automobil-Anwendungen) im Rahmen des Zentrums für Verkehr der *Technischen Universität Braunschweig* untersucht, mit welchen Methoden, Vorgehensmodellen und Werkzeugen ein durchgängiger Entwicklungsprozess von elektronischen Systemen geschaffen werden kann. Im Teilprojekt *Digitales Lastenheft* des Instituts Software, Abteilung Programmierung, werden dabei Werkzeuge eingesetzt wie *Telelogic DOORS*® [Doo] zur Einhaltung von Konsistenzbedingungen für funktionale Anforderungen beim Systementwurf und -design, *ARTiSAN Software Tools Real-time Studio*® (*RtS*) unter anderem zur Modellierung ausführbarer und animierbarer *Statecharts*, *ETAS Ascet-SD*® zur Modellierung diskreter Modelle und *Matlab/Simulink* zur Integration kontinuierlicher Modelle.

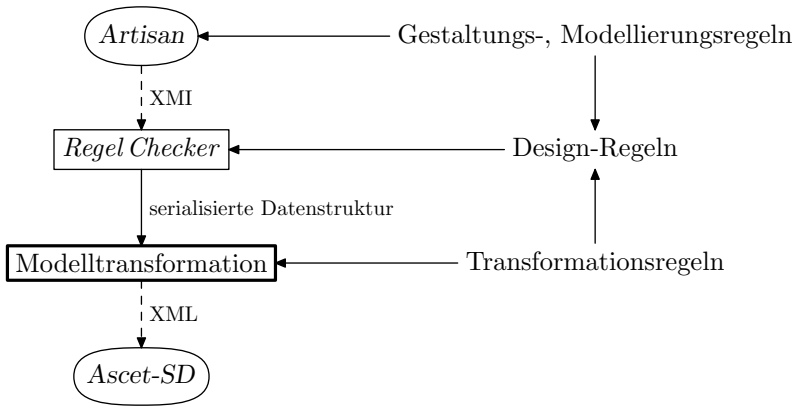


Abbildung 1.1: Schema der Modelltransformation

1.2 Beitrag dieser Arbeit

Bezüglich des Einsatzes der *Software*-Produkte wurde festgestellt, dass *Artisan RtS* vorteilhaft bei der Modellierung von *Statecharts* eingesetzt werden kann, *Ascet-SD* jedoch den effizienteren Programmcode erzeugt, der höchsten Ansprüchen beim Einsatz in elektronischen Steuergeräten genügt. Diese Tatsache macht demnach die Verwendung der unter *Artisan RtS* erstellten Modelle in Kombination mit einer Transformation der Strukturelemente und des Verhaltens der *Statecharts* höchst wünschenswert. Abb. 1.1 zeigt die dafür notwendige Werkzeugkette. Der Transformation vorangestellt ist der *Regel Checker*, der unter anderem nicht zu übersetzende Komponenten der *Statecharts* abfängt und aus den in *XMI* gespeicherten Modellen eine Datenstruktur für *Statecharts* erstellt. Auf dieser Struktur aufbauend sollte eine automatische Übersetzung der *Statecharts* erfolgen. Die Arbeiten zu einem *Regel Checker* wurden bereits durch Pietsch [Pie03] beendet. Es ist vorgesehen, an die Transformation die Wandlung der Objektstruktur in *XML*-Daten als Eingabeformat für *Ascet-SD* anzuschließen. Zwar existiert derzeit weder ein Export-Modul für *Statecharts* unter *Artisan RtS* noch ein Import-Modul für Zustandsautomaten unter *Ascet-SD*, jedoch kündigen beide Hersteller diese für künftige Versionen ihrer Produkte an.

Ausgehend von den genannten Erfordernissen soll es deshalb Aufgabe dieser Arbeit sein,

- einen geeigneten Formalismus für die Beschreibung der Transformation von *Statecharts* zu finden, der universell einsetzbar ist. Voraussetzung dafür ist die formale Beschreibung der Modelle, die an der Transformation beteiligt sind.

- zu prüfen, welche Bestandteile der *Statecharts* von *Artisan RtS* sich durch die in *Ascet-SD* darstellen lassen und ggf. Beschränkungen der *Statecharts* bzgl. der Transformation vorzunehmen. Dafür sind umfangreiche Analysearbeiten an beiden Pro-

dukten notwendig. Bei der Beschränkung ist darauf zu achten, dass so wenig Strukturelemente wie möglich von der Transformation ausgeschlossen werden.

- für die Syntax- und Semantik der *Statecharts* Transformationsregeln zu entwerfen, die ein *Statechart* von *Artisan RtS* nach *Ascet-SD* überführen. Dabei sollte darauf geachtet werden, dass die Regeln leicht auf die speziellen Gegebenheiten anderer Modellierungswerkzeuge (z. B. *Statemate*) angepasst und bei einer dortigen Transformation eingesetzt werden können.

- Transformationsregeln zu finden, die hierarchische Konstrukte in *Ascet-SD* zugunsten einer formalen Verifikation (z.B. zum Finden von Verklemmungen) in flache übersetzen (s. g. *Flattening*),

- Regeln zur Transformation in ein *Sun™ Java®*-Programm zu fassen, so dass mit *Artisan RtS* erstellte *Statecharts* in *Ascet-SD* benutzt werden können. Dabei ist die bestehende Datenstruktur zum Austausch der Modelle zwischen *Regel Checker* und Modelltransformation zu verwenden.

1.3 Gliederung der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut:

- Im anschließenden Kapitel wird ein Überblick über Ausprägungen von *Statecharts* gegeben. Zunächst wird aufgeschlüsselt, welchen Ursprung und welche Intention *Statecharts* besitzen. Danach wird speziell auf *Statecharts* von *Artisan RtS* und Zustandsautomaten von *Ascet-SD* eingegangen. Zustandsautomaten werden dabei als *Statecharts* mit beschränktem Umfang an Strukturelementen betrachtet. Die Notationen und Verhaltensweisen der Modelle werden eingehend beschrieben und jeweils formale Syntaxen notiert.

- In Kapitel 3 erfolgt die Vorstellung des eigenen entwickelten Ansatzes zur Überführung von Syntax und Semantik verschiedener *Statecharts*. Hier wird in Anlehnung an den mengentheoretischen Ansatz von Graphgrammatiken ein mengentheoretischer Ansatz für *Statechart*-Grammatiken entwickelt. Die Sprache dieser Grammatiken wird mit einer hier definierten Steuersprache erzeugt. Bei der Entwicklung des Überführungsmechanismus wurde der Anwendbarkeit auf alle *Statechart*-Konzepte große Aufmerksamkeit gezollt. Dafür wird eigens ein verallgemeinertes *Statechart*-Modell, das eine universelle Notation von *Statecharts* darstellt, entworfen.

- Aufbauend auf den Definitionen des Kapitels 3 schließt sich in Kapitel 4 die Anwendung bei der Überführung der *Statecharts* von *Artisan RtS* in Zustandsautomaten von *Ascet-SD* an. Dazu werden beide Modelle bzgl. Syntax und Semantik verglichen, die übersetzbaren Strukturelemente festgestellt und anschließend auf Basis

der *Statechart*-Grammatik ein spezielles Transformationssystem aufgestellt. Spezialitäten sind dort das s. g. *Flattening* (Einebnen serieller Hierarchiezustände) und die Sequentialisierung orthogonaler Zustände unter Erhalt des ursprünglichen Verhaltens. Bei der Konstruktion des Transformationssystems wurde großer Wert auf die Übersetzung möglichst aller Strukturelemente gelegt, so werden im Gegensatz zu Arbeiten wie [Bjö01, GPP98, Sim00] auch Aktivitäten, Bedingungen, Aktionen und verschiedene Arten von Ereignissen in die Übersetzung einbezogen.

- Das Fallbeispiel einer Ampelsteuerung in Kapitel 5 zeigt die Verwendbarkeit der *Statechart*-Grammatik als Transformationssystem für *Statecharts* als auch die Ableitbarkeit eines *Statecharts* von *Artisan RtS* mit dem Ergebnis eines Modells, das unter *Ascet-SD* mit gleichem Verhalten ausführbar ist.

- Kapitel 6 notiert und beschreibt die Konzeption sowie die Klassen und Methoden der zu dieser Arbeit entstandenen Software. Das *Java*-Programm dient der automatischen Transformation von *Statecharts* von *Artisan RtS* in Zustandsautomaten von *Ascet-SD*.

- In Kapitel 7 wird eine Zusammenfassung der Arbeit präsentiert und Grenzen der Arbeit beleuchtet. Die Arbeit wird durch einen Ausblick auf die möglichen zukünftigen Entwicklungen in diesem Bereich abgeschlossen.

Kapitel 2

Statecharts

2.1 Überblick

Statecharts wurden im Jahre 1987 von D. HAREL als ein visueller Formalismus zur Beschreibung Reaktiver Systeme [Har87] vorgeschlagen. Wie in [BA77] beschrieben, stellen sie eine Weiterentwicklung von Zustandsautomaten dar, die es erlauben, Zustände zu schachteln, und die Möglichkeit bieten, Nebenläufigkeit, zusammengesetzte Transitionen, Prioritäten und *Broadcasting* zu modellieren. Mit ihnen lässt sich auf graphische Weise das Verhalten von dynamischen Systemen modellieren, die auf interne bzw. externe Ereignisse reagieren und deren Antwort von ihrem aktuellen Zustand abhängt. Die *Statecharts* von HAREL wurden zum Vorbild für viele heutige Ausprägungen von *Statecharts* (s. Abb. 2.1).

Eine prominente Stellung unter den *Statecharts* nehmen die der *Unified Modeling Language* (*UML*) ein. Die *Object Management Group* (*OMG*) entwickelte die *UML* mit dem Ziel der Standardisierung für objektorientierte Methoden und Techniken. Neben der Standardisierung der einzelnen Beschreibungstechniken, die in [OMG01] vorgestellt werden, wird die Integration der verschiedenen Sichtweisen auf ein objektorientiertes System und deren Beschreibungstechniken vorangetrieben. Zur Darstellung der verschiedenen Sichtweisen bietet die *UML* eine Vielzahl von Notationen und Diagrammen, die auf unter-

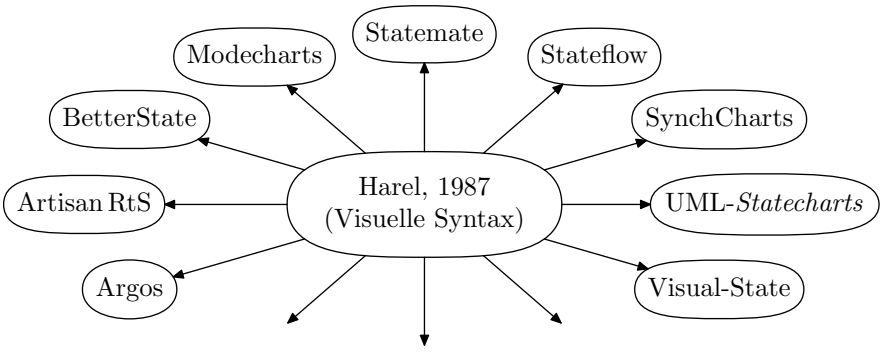


Abbildung 2.1: Ausprägungen von Statecharts

schiedlichsten Abstraktionsniveaus eingesetzt werden können. *Statecharts* sind dabei ein wesentlicher Bestandteil der Modellierung der dynamischen Sicht objektorientierter Systeme. Die *UML* stützt sich damit auch auf die Arbeiten von [HG96]. Der Ansatz der Standardisierung besteht in der Spezifikation eines Metamodells, das die Konzepte für die objektorientierte Modellierung beschreibt und ihre Semantik angibt.

In den folgenden Abschnitten soll auf zwei Ausprägungen von *Statecharts* eingegangen werden, die in dieser Arbeit zur Anwendung kommen und deren Aufbau Grundlage für darauffolgende Kapitel ist. Dabei wird bewusst der ähnliche Aufbau beider durch ähnliche Strukturen der Beschreibung unterstrichen. So können beide Abschnitte vergleichend oder jedes für sich beschreibend gelesen werden.

2.2 Das Werkzeug *Artisan RtS*

Artisan RtS ist ein Modellierungswerkzeug, das weitgehend nach den Richtlinien des *UML*-Standards in [OMG01] entwickelt wurd. Dabei legte der Hersteller großen Wert auf die Integrierbarkeit in einen bestehenden Entwicklungsprozess. In *Artisan RtS* ist es somit möglich, spezifiziertes System- und Softwareverhalten visuell zu validieren, indem Sequenzdiagramme animiert, Zustandsmodelle simuliert und graphische Anzeigen von *Altia® Design* [Alt] integriert werden können. Die Synchronisation mit *DOORS* bietet Konsistenz zu formulierten Anforderungen beim Entwurf und Design von Systemen. Mit dem Einsatz von Echtzeit-Sprachkonstrukten ist *Artisan RtS* prädestiniert für die Entwicklung von Echtzeit-Systemen.

Für die Beschreibung der *Statecharts* in *Artisan RtS* werden zunächst in Abschnitt 2.2.1 die Strukturelemente aufgezählt und informell beschrieben, um deren Verhalten in Abschnitt 2.2.2 ebenfalls informell zu erörtern. Abschnitt 2.2.3 zeigt eine – für diese Arbeit geeignete – Möglichkeit, die Syntax formal wiederzugeben.

2.2.1 *Statecharts* in *Artisan RtS*

Statecharts in *Artisan RtS* eignen sich dazu, das dynamische Verhalten eines Systems zu simulieren. Ein Benutzer kann während der Ausführung eines solchen *Statechart* Ereignisse aktivieren und die dynamischen Folgen in einer Animation beobachten.

Ein *Statechart* besteht aus einer endlichen Anzahl von Zuständen und Übergängen (s. g. Transitionen) zwischen diesen. Dabei sind Übergänge beschriftete gerichtete Kanten. Abb. 2.2 zeigt schematisch die Komponenten eines *Statecharts* mit rekursivem Ansatz. Als charakteristisches Beispiel für den Sprachumfang stellt Abb. 2.3 die wesentlichen graphischen Komponenten der *Statecharts* in *Artisan RtS* dar. Sie sollen im folgenden einzeln erläutert werden.

Zustand: Jeder Zustand wird graphisch als abgerundetes Rechteck dargestellt und erhält eine Namensmarkierung, einige Zustände erhalten zusätzlich eine Markierung als *Startzustand*; dies ist der Zustand, in dem sich der Automat zu Beginn befindet.

Die Diagramme erlauben es, Zustände zu gruppieren und s. g. *Hierarchiezuständen* zuzuordnen. Zustände, die keine weiteren Zustände enthalten, werden als *Basiszustände* bezeichnet. Zustände, die eine *Geschichte* haben, dienen dazu, zukünftige Aktivitäten auf vorherigen aufzubauen.

Aktivitäten: Zustände können optional *Eintritts-*, *statische* und *Austrittsaktivitäten*[1] haben. Eintrittsaktivitäten werden mit ‚Entry/‘, statische mit ‚do:‘ und Austrittsaktivitäten mit ‚Exit/‘ vor der Aktivität im Zustand gekennzeichnet.

[1] Zur genauen Verhaltensbeschreibung wird zwischen Aktivität und Aktion unterschieden. Eine Unterscheidung wird wegen des speziellen Verhaltens bei Aktivitäten notwendig: Eine Aktivität ist im Gegensatz zur Aktion durch Ereignisse abbrechbar, noch bevor sie vollständig beendet werden kann.

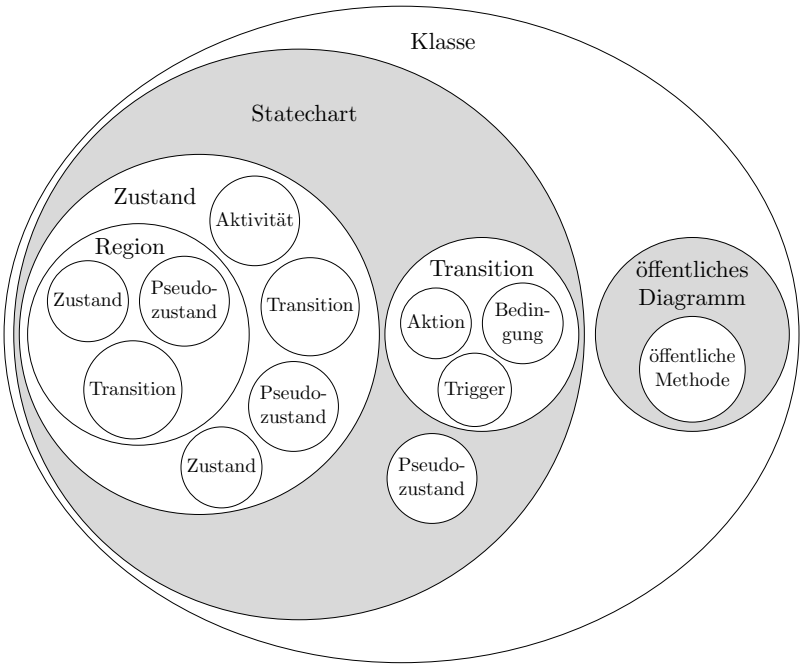


Abbildung 2.2: Schema eines Statecharts in Artisan RtS

interne Transition: Ein Zustand kann eine oder mehrere interne Transitionen besitzen. Diese können *Trigger*, Bedingungen und Aktionen haben. Sie werden mit $e[c]/a$ im Zustand beschrieben, wobei e der *Trigger*, c die Bedingung und a die Aktion sein soll. Quelle und Ziel der internen Transition sind ein und derselbe Zustand.

Startzustand: Auf jeder Hierarchieebene zeigt ein *Startzustand* an, welcher Zustand als erster aktiviert wird. Die Auszeichnung erfolgt mit dem Pseudozustand *Startverknüpfung* (s. u.).

Hierarchie: Innerhalb eines Zustandes einer Hierarchieebene bilden die Unterzustände ihrerseits ein *Statechart*.

Man unterscheidet in den *Statecharts* von *Artisan RtS* zwei Arten von Hierarchiezuständen: *serielle* (*OR*-) Zustände, wobei jeweils nur ein Unterzustand aktiv ist, und *orthogonale* (*AND*-) Zustände, in denen dies eine Menge von Unterzuständen sein kann.

serieller Hierarchiezustand: Zu Beginn ist auf jeder Unterhierarchiestufe eines seriellen Zustandes immer genau ein Zustand, der *Startzustand*, aktiv. Wenn ein Unterzustand aktiv ist, ist dessen übergeordneter Hierarchiezustand ebenfalls aktiv. Ein Übergang heraus aus letzterem erzwingt das Verlassen des gerade aktiven Unterzustands. Unterzustände können Übergänge zu Zuständen haben, welche sich außerhalb des übergeordneten Hierarchiezustandes befinden, insbesondere auch zu Unterzuständen

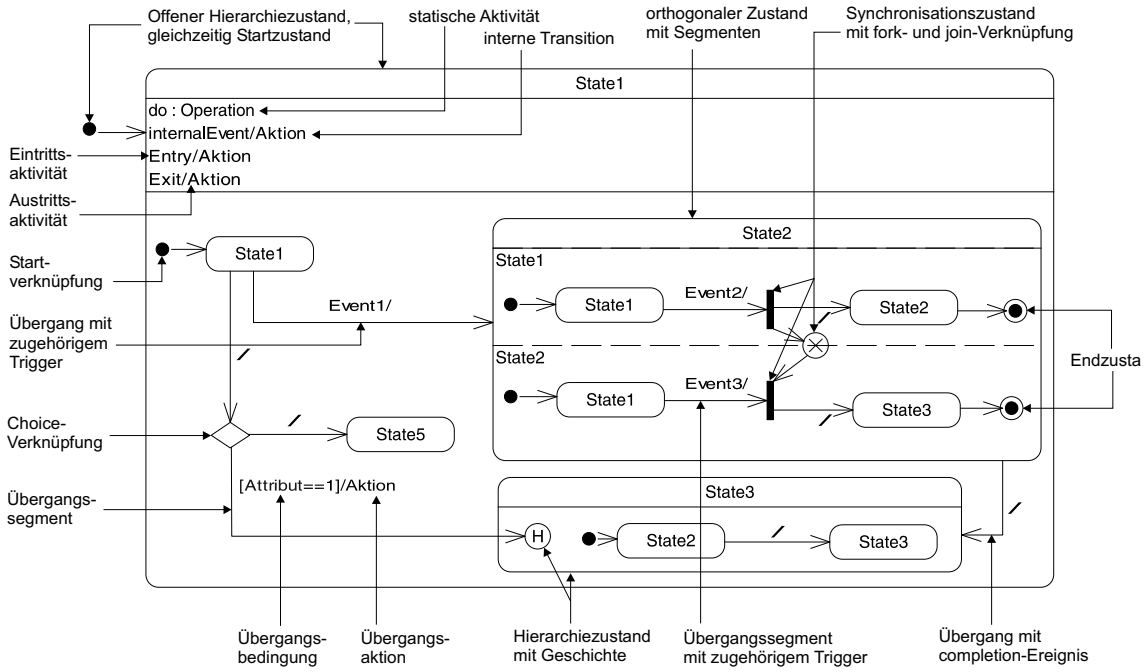


Abbildung 2.3: Komponenten der Statecharts in Artisan RtS

anderer Hierarchiezustände.
Zum Zwecke einer besseren Übersichtlichkeit lassen sich in *Artisan RtS* Unterzustände in einem separaten *Statechart* anlegen. Eine solche Konstruktion wird als *geschlossene* Hierarchie bezeichnet und ein entsprechender Hierarchiezustand als *submachine*-Zustand. Sie werden mit einem ‚STD : *Zustandsname*‘ als zusätzlicher Eintrag des Hierarchiezustandes formatiert. Eine Konstruktion, die aus der *UML* als s. g. *stub-state* bekannt ist, existiert in *Artisan RtS* nicht.

orthogonaler Hierarchiezustand: Orthogonale Zustände werden durch gestrichelte horizontale Linien in s. g. *Regionen* unterteilt. Diese Regionen können als serielle Zustände (ohne Aktivitäten) interpretiert werden. So hat jede Region einen *Startzustand*, der bei Aktivierung des orthogonalen Hierarchiezustandes aktiviert wird. Zustandsänderungen und Ausführen von Aktivitäten und Aktionen finden in den Regionen völlig unabhängig von denen anderer Regionen statt.

Geschichte: Soll der Zustand, der beim Verlassen eines Oberzustandes aktiv war, bei Wiedereintritt erneut aktiviert werden, wird der übergeordnete Zustand oder die Region mit einer Geschichte belegt. In einem orthogonalen Zustand kann jede Region mit einer Geschichte versehen werden. Ein Hierarchiezustand oder eine Region mit einer solchen (*einfachen*) Geschichte wird mit einer *Shallowhistory*-Verknüpfung gekennzeichnet. Darüber hinaus gibt es noch

Zustände mit einer *vollständigen* Geschichte. Dabei gilt die Eigenschaft der Geschichte nicht nur für die direkten Unterzustände, sondern auch für deren Unterzustände, usw. Solche Zustände werden mit der *Deephistory*-Verknüpfung versehen.

Neben den o. g. Zuständen gibt es noch spezielle Zustände, die das Verhalten eines *Statecharts* in *Artisan RtS* steuern. Dazu gehören:

Finalstate oder *Endzustand:* Ein weiterer besonderer Zustand ist der Endzustand, der als gefüllter Kreis umzirkelt dargestellt wird. Dieser namenlose Zustand darf weder Unterzustände noch ausgehende Transitionen besitzen. Ein aktiver Endzustand signalisiert die Beendigung des zugehörigen Oberzustandes.

synch-state oder *Synchronisationszustand:* Mit dem Synchronisationszustand lässt sich unter Zuhilfenahme von *Join*- und *Fork*-Verknüpfung eine Synchronisation zwischen Zuständen in orthogonalen Regionen realisieren. Er wird mit einem umzirkelten ‚×‘ oder einer umzirkelten positiven ganzen Zahl dargestellt.

Pseudozustand: In den *Statecharts* von *Artisan RtS* werden Zustände und Pseudozustände unterschieden. Dabei können Zustände während des Ablaufs eines *Statecharts* betreten werden, während Pseudozustände das nicht werden können. Sie dienen der Steuerung oder der Übersichtlichkeit eines *Statecharts*. Folgende Pseudozustände sind in *Artisan RtS* bekannt:

Initial- oder *Startverknüpfung:* Die Startverknüpfung zeigt an, welcher Zustand ein Startzustand ist. Die Darstellung erfolgt mittels eines gefüllten Kreises und einer davon ausgehenden Transition, die auf den Startzustand weist. Die Transition kann mit einer Übergangsaktion beschriftet sein. Eine Transition hin zur Startverknüpfung existiert nicht.

Deep-/Shallowhistory-Verknüpfung: Diese Pseudozustände dienen der Auszeichnung von Zuständen mit vollständiger/einfacher Geschichte. Es kann Transitionen von und/oder zu diesen Pseudozuständen geben. Die *Deephistory*-Verknüpfung wird mit einem umzirkelten ‚H*‘ gekennzeichnet, die *Shallowhistory* mit einem umzirkelten ‚H‘.

Join-Verknüpfung: Join-Verknüpfungen führen mehrere Transitionssegmente, die von den Unterzuständen in Regionen orthogonaler Zustände ausgehen, zu einem zusammen. Sie werden als kurze dicke vertikale Balken dargestellt, wobei mehrere Transitionssegmente in diese hineinführen, aber nur eine sie wieder verlässt.

Fork-Verknüpfung: Im Gegensatz zur *Join*-Verknüpfung hilft die *Fork*-Verknüpfung beim Verteilen von Zustandsüberführungen auf die Unterzustände von

Tafel 2.1: Beschränkungen der Beschriftungen von Transitionen in Statechartsvon Artisan RtS

Pseudozustand	Beschriftung der Transition	
	zum Pseudozustand	vom Pseudozustand
Startverknüpfung	–	Aktion
Deep-/*Shallowhistory*-Verknüpfung	*Trigger*, Bedingung, Aktion	Aktion
Fork-Verknüpfung	*Trigger*, Bedingung, Aktion	Aktion
Join-Verknüpfung	Aktion	*Trigger*, Bedingung, Aktion
Choice-Verknüpfung	*Trigger*, Bedingung, Aktion	Bedingung, Aktion

Regionen orthogonaler Zustände. Sie wird gleich der *Join*-Verknüpfung gezeichnet, hat aber ein eingehendes und mehrere ausgehende Transitionssegmente.

Choice-Verknüpfung: *Choice*-Verknüpfungen[2] repräsentieren Verzweigungsstellen im *Statechart*. Sie erleichtern die Darstellung verschiedener Übergänge, indem sie diese in einzelne Segmente zerlegen. *Choice*-Verknüpfungen werden durch einen Rhombus im *Statechart* dargestellt. Sie haben mehrere eingehende und ausgehende Transitionssegmente.

Transition: Eine Transition (auch: Zustandsübergang) verbindet zwei (Pseudo-) Zustände. Sie wird durch eine gerichtete Kante vom Ausgangszustand zum Zielzustand dargestellt.

Übergänge von Zustand zu Zustand können mit *Trigger*, *Bedingungen* und *Übergangsaktionen* beschriftet sein. *Bedingungen* werden im *Statechart* eingeschlossen in eckigen Klammern dargestellt, *Übergangsaktionen* mit einem vorangestellten ‚/‘. Zu einem Übergang kommt es, wenn sein Anfangszustand aktiv ist, ein *Trigger*-Ereignis auftritt und seine Bedingung erfüllt ist. Dann werden Übergangsaktionen ausgeführt, wodurch *Variablen* geändert und neue Ereignisse ausgelöst werden können. Abschließend ist der Anfangszustand inaktiv und der Zielzustand aktiv.

Die Beschriftungen von Transitionen, die von Pseudozuständen ausgehen oder zu ihnen hinführen unterliegen den Beschränkungen, wie sie in Taf. 2.1 aufgelistet sind. Alle *Trigger*, Bedingungen und Aktionen sind optional.

Bedingung: Ein Übergang von einem Zustand in einen anderen findet nur statt, wenn der Boolesche Ausdruck einer Bedingung wahr ist.

Übergangsaktion: Optional können Übergänge wie auch Übergangssegmente *Übergangsaktionen* haben. Übergangsaktionen können Ereignisse auslösen.

[2] Entgegen der *online*-Dokumentation der *Statecharts* in *Artisan RtS* soll hier die Bezeichnung *choice* statt *junction* verwendet werden, da das Konstruktionselement in der Darstellung *Choice*-Element der *UML* nahekommt.

Broadcast: Aktionen, die wieder Ereignisse senden, heißen *Broadcasts*. Die Eigenschaft eines *Statecharts*, Ereignisse in den Aktionen unterzubringen, wird *Broadcasting* genannt.

Trigger: Trigger aktivieren die Ausführung des *Statechart* immer dann, wenn ein bestimmtes Ereignis eintritt. Bei der Zustandsüberführung wird das Ereignis konsumiert, d. h. es steht für keinen weiteren Übergang mehr zur Verfügung. In *Artisan RtS* sind eine Reihe solcher Ereignisse bekannt:

Call-Event: Ein *Call-Event* ist gleichzusetzen mit dem Aufruf einer Methode einer Klasse.

Change-Event: Dieses Ereignis tritt immer dann ein, wenn der Boolesche Wert seines Parameters von *false* nach *true* wechselt. Es wird mit einem ‚*when*' als Ereignis-Beschriftung einer Transition gekennzeichnet. Tritt ein *Change-Event* auf und ist eine Aktivität des Zustandes noch nicht beendet, wird die Aktivität des Zustandes abgebrochen, die Zustandsüberführung findet nicht statt, und das *Change-Event* wird konsumiert.

Completion-Event: Ein besonderes Ereignis ist das *Completion*-Ereignis. Mit diesem Ereignis werden alle diejenigen Zustandsübergänge ausgelöst, die nicht mit einem *Trigger* versehen sind. Dabei handelt es sich um s. g. *Completion*-Transitionen. Dieses Ereignis löst genau dann eine Transition aus, wenn die Aktivitäten des Quellzustandes der Transition beendet sind.

Signal-Event: Dabei handelt es sich um ein Ereignis, das in Form eines expliziten Signals von anderen Objekten als dem *Statechart* selbst gesendet wird.

Time-Event: Dieses Ereignis tritt nach einer Zeit *t* ein, nachdem ein dazugehöriger Zähler gestartet wurde. Falls nicht anders deklariert, wird der Zähler beim Betreten des Quellzustandes einer Transition gestartet. Die Transition ist dann mit ‚*after*(*t*)' beschriftet.

Variablen: Variablen dienen der Speicherung und Verarbeitung von numerischen und nichtnumerischen Werten. In *Statecharts* von *Artisan RtS* werden die Datentypen, wie sie in Taf. 2.2 aufgeführt sind, unterschieden. Sie entsprechen denen der Definition von Programmiersprachen wie *Java* oder *C/C++*.

2.2.2 Semantik von *Statecharts* in *Artisan RtS*

Um zu beschreiben, wie ein Zustandsdiagramm interpretiert und ausgeführt wird und in welcher Reihenfolge dabei die Übergangsaktionen und Aktivitäten durchgeführt werden, sind Regeln für die *Abarbeitung von Zuständen*, *Auswahl* und *Abarbeitung von Übergängen* vonnöten, die Semantik. Ihr widmet sich dieser Abschnitt.

Tafel 2.2: Datentypen der Statecharts in Artisan RtS

Datentyp	Wert
any	typenlos
boolean	Boolescher Wert
char	Zeichen
double	Gleitkommazahl mit doppelter Genauigkeit
float	Gleitkommazahl mit einfacher Genauigkeit
long	große ganze Zahl
octal	Zeichenkette für eine ganze Zahl in Oktal-Darstellung mit führendem ‚O‘
short	kleine ganze Zahl
unsigned long	*long*, vorzeichenlos
unsigned short	*short*, vorzeichenlos

Grundlagen

Zusammenfassend besteht ein *Statechart* aus einer endlichen Anzahl von Zuständen. Unter bestimmten Umständen ändert sich der Gesamtzustand des Systems, dargestellt durch Übergänge zwischen den verschiedenen Zuständen. Diese Übergänge unterliegen Bedingungen.

Um ein *Statechart* zu aktivieren, bedarf es eines äußeren Ereignisses, des s. g. *Trigger*-Ereignisses. Dies ist eine öffentliche Methode des *Statechart*. Tritt ein *Trigger*-Ereignis auf, so reagiert das System mit der Ausführung von Übergangsaktionen (z. B. Erzeugung eines Signals, Ändern eines Variablenwerts oder dem Übergang in einen anderen Zustand).

Bei der *Initialisierung* des *Statecharts*, die vor jeglicher Zustandsüberführung stattfindet, wird der Startzustand des Systems aktiviert. Handelt es sich bei diesem Startzustand um einen Hierarchiezustand, wird auch der Startzustand innerhalb der Hierarchie aktiviert. Dabei werden Eintrittsaktivitäten in festgelegter Reihenfolge (s. u.) abgearbeitet.

Eintrittsaktivität: Eine Eintrittsaktivität eines Zustands wird ausgeführt, wenn ein Übergang in diesen stattfindet. Bevor die Eintrittsaktivität ausgeführt wird, wird der Zustand aktiviert.

statische Aktivität: Für die statische Aktivität ist das Eintreten eines *Trigger*-Ereignisses der Auslöser, das weder Zustandsüberführung noch interne Transition zur Folge hat.

Austrittsaktivität: Die Austrittsaktivität eines Zustands wird abgearbeitet, wenn ein Übergang aus diesem Zustand heraus stattfindet. Der Zustand wird erst inaktiv, wenn die Ausführung der Austrittsaktivität beendet ist.

Übergangsaktion: Die Übergangsaktion wird erledigt, nachdem der Ausgangszustand verlassen und deaktiviert wurde und bevor der Zielzustand aktiviert wird.

interne Transition: Die interne Transition eines Zustands wird ausgelöst, wenn dieser Zustand aktiv ist und ein *Trigger*-Ereignis eintritt, das das Auslösen der internen Transition zur Folge hat aber nicht den Übergang aus dem Zustand heraus. Das Auslösen hängt von einer Bedingung ab und eine Aktion wird ausgeführt. Die interne Transition wird gegenüber den anderen Transitionen als die mit der kleinsten Priorität behandelt. Sie wird direkt im Anschluss an die statische Aktivität ausgeführt. Beim Ausführen einer internen Transition wird ein Zustand nicht verlassen.

Trigger-Ereignisse können die statische Aktivität eines Zustandes abbrechen, falls diese noch nicht abgeschlossen ist. Allgemein benötigt eine Zustandsüberführung keine Zeit. Die Aktivitäten des Zustandes benötigen soviel Zeit, wie zur Abarbeitung ihrer Teilaktivitäten nötig ist.

Bei einem Übergang zwischen zwei Unterzuständen desselben Hierarchiezustands arbeitet der Hierarchiezustand (der ja nicht verlassen wird) nachdem der Ausgangszustand verlassen wurde zunächst seine statische Aktivität und danach die internen Transitionen ab, bevor die Übergangsaktion ausgeführt wird. Dabei wird der Hierarchiezustand nicht erneut betreten, d. h. es werden keine Eintritts- und Austrittsaktivitäten ausgeführt.

Konkurrierende Transitionen

Von jedem Zustand können mehrere Transitionen ausgehen. Für die Reihenfolge der Auswertung dieser Zustandsüberführungen schlägt die *UML* eine nichtdeterministische Abarbeitung der Transitionen vor. Da diese Eigenschaft von *Statecharts* im Detail also unklare Handlungsanweisungen zur Folge haben, obliegt es nun den Entwicklern von Modellimplementierungen von *Statecharts*, eine Umsetzung für diesen Nichtdeterminismus zu realisieren. Einige vom Hersteller des Produktes nicht deklarierte deterministische Umsetzungen lauten: Gehen zwei Transitionen von einem Zustand aus und

- eine besitzt ein *Completion*-Ereignis und die andere ist mit einem vom Benutzer definierten Ereignis ausgestattet, dann hat die Transition mit dem *Completion*-Ereignis die höhere Priorität, d. h. sind beide Transitionen zum Schalten bereit, wird diejenige mit dem *Completion*-Ereignis zuerst ausgewertet werden.
- beide sind mit unterschiedlichen vom Benutzer definierten Ereignissen ausgestattet, dann hat die Transition mit dem Ereignis, welches zuerst definiert wurde, die höhere Priorität.
- beide sind mit dem *Completion*-Ereignis ausgestattet und eine Transition benutzt in seiner Aktion eine vom Benutzer definierte Variable, während die andere keine solche benutzt, dann hat letztere die höhere Priorität.
- beide sind mit dem *Completion*-Ereignis ausgestattet und beide benutzen unterschiedliche vom Benutzer definierte Variablen in den Aktionen, dann hat die Transition, die die zuerst definierte Variable in der Aktion beinhaltet, die höhere Priorität.

Weitere deterministische Umsetzungen konnten nicht statuiert werden. Die Auswertung der Zustandsüberführungen beginnt immer mit der höchsten (impliziten) Transition oder zufällig. Sobald die Auswertung einer Bedingung den Wert *wahr* ergibt, findet der zugehörige Übergang statt und alle anderen Bedingungen, die zu Übergängen mit niedrigeren Prioritäten gehören, werden nicht geprüft. Wenn die Auswertung bei keiner Bedingung *wahr* ergibt, bleibt der Zustand unverändert und es werden nach der statischen Aktivität die internen Transitionen entsprechend den gleichen soeben genannten Regeln ausgewertet.

Choice-Verknüpfung

Choice-Verknüpfungen repräsentieren Verzweigungsstellen im *Statechart*. Sie erleichtern so die Darstellung verschiedener Übergänge, indem sie diese in einzelne Segmente zerlegen. Trifft eine Transition in der Verknüpfung ein, werden an dieser Stelle die Bedingungen der ausgehenden Transitionen geprüft und die mit der passenden Bedingung für den Zustandsübergang benutzt. Treffen mehrere Bedingungen zu, wird wie im Abschnitt 2.2.2 (konkurrierende Transitionen) vorgegangen. Die Bedingungen und Aktionen der von der Verknüpfung ausgehenden Transitionssegmente können auf Variablenänderungen in den Aktionen der eingehenden Transitionssegmente nicht reagieren. Dieses Verhalten findet man bei der *Junction*-Verknüpfung der *UML*.

Deep-/Shallowhistory-Verknüpfung

Eine *Shallowhistory*-Verknüpfung als Unter- (Pseudo-) Zustand eines hierarchischen Zustandes verändert dessen Verhalten so, dass bei Wiedereintritt der letzte aktive Unterzustand aktiviert wird. Beim ersten Betreten wird, wie bei Hierarchiezuständen ohne Geschichte, der Unterzustand (die Unterzustände) mit der Startverknüpfung aktiviert. Die *Deephistory*-Verknüpfung erweitert dieses Verhalten rekursiv auf die Unterzustände der unteren Hierarchieebenen.

Es können Transitionen in diese Verknüpfungen und aus ihnen heraus existieren. Mit der letzteren Konstruktion kann eine Startverknüpfung ersetzt werden. Dabei wird der Zustand, auf den eine Transition ausgehend von einer *History*-Verknüpfung verweist, beim ersten Betreten des Hierarchiezustandes als Startzustand aktiviert. Existiert nur eine Transition hin zur *History*-Verknüpfung, so ist diese Konstruktion gleichzusetzen mit einer Transition in den hierarchischen Zustand, der die *History*-Verknüpfung enthält.

Serielle Hierarchiezustände

Ist das *Statechart* aktiviert, werden die Bedingungen der Übergänge untersucht. Die hierarchische Reihenfolge legt die Priorisierung fest. Dabei hat die niedrigste hierarchische Ebene die höchste Priorität, d. h. die Bedingungen für die Übergänge auf den untersten hierarchischen Ebenen werden zuerst untersucht. Kommt es dazu, dass ein Hierarchiezustand verlassen wird, so werden ebenfalls die aktuellen Unterzustände verlassen. Der

innerste Unterzustand wird auch hier zuerst verlassen, der äußerste Hierarchiezustand zuletzt. Im Gegensatz dazu ist die Reihenfolge bei Eintritt in einen Hierarchiezustand vom äußersten Hierarchiezustand zum innersten (Unter-) Zustand, d. h. zuerst erfolgt der Eintritt in den äußersten Zustand und zuletzt der Eintritt in den innersten Zustand. Für den Fall, dass kein Übergang stattfindet und statische Aktivität und interne Transitionen ausgeführt werden müssen, so erfolgt diese Auswertung in der Reihenfolge von innen nach außen.

serieller Hierarchiezustand: Beim Eintritt in einen seriellen Hierarchiezustand gibt es zwei Möglichkeiten:

Hierarchiezustand hat keine Geschichte: Beim Betreten des Hierarchiezustandes erfolgt der Eintritt immer in den Startzustand des Hierarchiezustands.

Hierarchiezustand hat Geschichte: Beim erstmaligen Eintritt in einen Hierarchiezustand mit Geschichte erfolgt der Eintritt in den Startzustand dieses Hierarchiezustands. Bei erneutem Eintritt wird der bei Austritt aktive Unterzustand wieder aktiviert.

Für jeden seriellen Hierarchiezustand kann bestimmt werden, ob er eine Geschichte hat oder nicht.

Übergang innerhalb eines Hierarchiezustands: Falls ein Übergang innerhalb eines seriellen Hierarchiezustands stattfindet, verbleibt das *Statechart* in diesem Hierarchiezustand. Daher werden sowohl die statische Aktivität als auch die internen Transitionen dieses Hierarchiezustands ausgeführt, ebenso wie die aller Hierarchiezustände, in denen der betreffende Zustand enthalten ist. Diese werden nach allen Austrittsaktivitäten und vor der Übergangsaktion in der Reihenfolge vom innersten zum äußersten Hierarchiezustand ausgeführt.

Übergang zwischen Hierarchiezuständen: Findet ein Übergang zwischen Hierarchiezuständen statt, werden vor dem Übergang alle aktiven Unterzustände des Quellzustandes verlassen und danach die entsprechenden Zustände des Zielzustandes betreten.

Übergang zwischen Unterzuständen verschiedener Hierarchien: Darüberhinaus können Übergänge direkt vom Unterzustand eines Hierarchiezustands in den Unterzustand eines anderen Hierarchiezustands gehen. Dabei werden beim Verlassen/Betreten der Unterzustände auch die dazugehörigen Hierarchiezustände betreten/verlassen.

Übergang von einem Unterzustand in einen Hierarchiezustand: Wenn der Übergang von einem Unterzustand nicht in einen anderen Unterzustand geht, sondern in den Hierarchiezustand, ist die Vorgehensweise fast dieselbe wie beim Übergang in einen anderen Unterzustand. Der Unterzustand wird verlassen, der Hierarchiezustand nicht.

Abhängig davon, ob der Hierarchiezustand eine Geschichte hat oder nicht, wird entweder der zuletzt aktive Unterzustand aktiviert oder der Startzustand innerhalb der Hierarchie.

Kein Übergang: Findet kein Übergang statt, bleibt der aktivierte Zustand aktiv und die statische Aktivität sowie die interne Transition werden ausgeführt. Ist der Zustand ein Unterzustand, werden statische Aktivitäten und interne Transitionen der dazugehörigen übergeordneten Hierarchiezustände ebenfalls ausgeführt.

Orthogonale Hierarchiezustände

Orthogonale Hierarchiezustände sind in mindestens zwei Regionen unterteilt, die jede für sich als serieller Hierarchiezustand ohne Eintritts-, statische und Austrittsaktivität und ohne interne Transition gesehen werden kann. Ein orthogonaler Hierarchiezustand funktioniert im Prinzip genau wie ein serieller Hierarchiezustand, wenn man davon ausgeht, dass, wenn der orthogonale Zustand aktiv ist, genau ein direkter Unterzustand einer jeden Region aktiv ist.

Für jede Region kann entschieden werden, ob sie eine Geschichte hat oder nicht. Beim erstmaligen Aktivieren eines orthogonalen Hierarchiezustandes werden die Startzustände aller Regionen aktiviert und die dazugehörigen Eintrittsaktionen ausgeführt. Dabei ist bei *Artisan RtS* die Reihenfolge der Regionen einzuhalten: der Startzustand der obersten Region wird als erster aktiviert, danach der der zweiten usw. Wird der orthogonale Hierarchiezustand ein weiteres Mal aktiviert, werden auch die entsprechenden Unterzustände der einzelnen Regionen aktiviert. Das Betreten der Regionen erfolgt nach den Regeln der seriellen Hierarchiezustände.

Tritt der Fall ein, dass zwei Transitionen verschiedener Regionen gleichzeitig zum Auslösen bereit sind, dann wird diejenige bevorzugt, die der weiter oben stehenden Region angehört[3]. Dabei wird der gesamte Zustandsüberführungsprozess wie oben beschrieben vollzogen und zwar einschließlich dem Ausführen der Eintrittsaktion des Zielzustandes (die auch schaltbereite Transition unterbricht nicht etwa diesen Prozess).

Findet eine Transition zwischen den Zuständen verschiedener Regionen statt, wird der Startzustand der verlassenen Region ohne Geschichte aktiviert. Verfügt die Region über Geschichte, wird der soeben verlassene Zustand wieder aktiviert. Wird eine Transition von einem Zustand außerhalb des orthogonalen Hierarchiezustandes zu einem Zustand, der Unterzustand einer Region ist, ausgelöst, dann wird der Zielzustand aktiviert, sowie auch die entsprechenden Zustände der anderen Region (entsprechend der Geschichte). Beim Betreten ist die schon erwähnte Reihenfolge der Regionen zu beachten.

Soll eine Zustandsüberführung direkt in mehrere Unterzustände eines orthogonalen Hierarchiezustandes stattfinden, so kann das im *Statechart* durch Aufteilen einer Tran-

[3] Das bedeutet z. B. auch, dass ein orthogonaler Hierarchiezustand mit einer Transition, die einer oberen Region angehört, verlassen werden kann, auch wenn eine Transition in einer niedrigeren Region, gleichzeitig zum Auslösen bereit ist, die den orthogonalen Hierarchiezustand nicht verlassen würde.

sition durch eine *Fork*-Verknüpfung erfolgen. Die Transitionssegmente, die von der Verknüpfung wegführen, zeigen auf die Unterzustände, die zu aktivieren sind. Dabei wird vor Betreten einer Region die entsprechende Übergangsaktion des Segmentes ausgeführt. Nach abgeschlossener Zustandsüberführung sind alle Zustände, in die die Segmente verweisen, aktiv.

Die o. g. Reihenfolge für das Betreten der Zustände betrifft auch das Verlassen eines orthogonalen Zustandes durch eine Zustandsüberführung einer seiner Unterzustände. Dabei wird nach Deaktivieren aller Unterzustände der Regionen auch der orthogonale Zustand deaktiviert. Gleiches gilt, wenn es direkte Zustandsüberführungen aus mehreren Unterzuständen gibt. Diese werden mit einer *Join*-Verknüpfung zu einer Transition zusammengeführt. Die Übergangsaktionen der zur *Join*-Verknüpfung hinführenden Transitionssegmente werden nach Verlassen der Unterzustände der Regionen und damit des orthogonalen Zustandes ausgeführt.

Mit einem Synchronisationszustand lassen sich konkurrierende Regionen synchronisieren. Er wird zusammen mit *Fork*- und *Join*-Verknüpfung dazu benutzt, um festzulegen, dass ein Zustand einer Region verlassen wird, bevor ein Zustand einer anderen Region betreten wird. Eine *Fork*-Verknüpfung modelliert die Voraussetzung für die *Join*-Verknüpfung, d. h. sobald die Zustandsüberführung in die *Fork*-Verknüpfung stattgefunden hat, ist die Voraussetzung für das Schalten der Transitionen der *Join*-Verknüpfung gegeben. Die Anzahl der Transitionen, die von einem Synchronisationszustand ausgehen und schalten sollen, kann mit der Differenz zwischen der Anzahl an Schaltvorgängen von einkommenenden und ausgehenden Transitionen angegeben werden. Vereinfachend soll nachfolgend jedoch auf die Betrachtung von Synchronisationszuständen mit begrenzter Anzahl von Schaltvorgängen verzichtet werden. In *Statecharts* von *Artisan RtS* werden die Transitionen in der Reihenfolge der Beschriftungen in Abb. 2.4 geprüft und ggf. ausgeführt. Die Reihenfolge der Auswertung der Transitionen ist eine implementationsspezifische Eigenschaft von *Artisan RtS*.

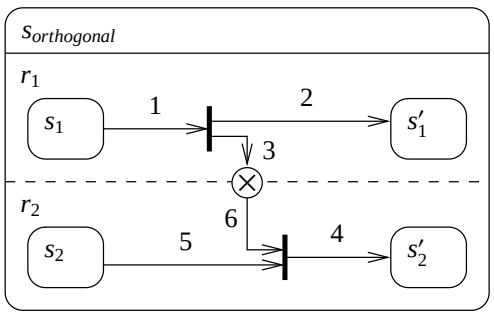


Abbildung 2.4: Reihenfolge der Auswertung von Transitionen an einem Synchronisationszustand

Zusammenfassung

In diesem Abschnitt soll das oben aufgeführte Verhalten als präziser Algorithmus wiedergegeben werden. Wenn dabei von Hierarchiezuständen die Rede ist, sind sowohl die seriellen als auch die orthogonalen gemeint. Bei orthogonalen Zuständen wirkt sich das Betreten/Verlassen von Unterzuständen, das Ausführen von Aktivitäten und das Überführen der Unterzustände immer auf alle Regionen in der Reihenfolge von oben nach unten aus. Wenn also im Folgenden die Einzahl für die Beschreibung von Konstrukten bei Unterzuständen benutzt wird, gilt für orthogonale Zustände die Mehrzahl, nämlich für die entsprechenden Konstrukte der Regionen.

Initialisierung des Zustandsdiagramms: Der Startzustand des Systems wird aktiviert. Ist der Startzustand ein Hierarchiezustand, wird auch der Startzustand innerhalb der Hierarchie aktiviert. Die Eintrittsaktivitäten werden dabei beginnend bei der höchsten Hierarchiestufe bis zur niedrigsten ausgeführt.

Eintritt in einen Zustand:

1. Wenn der Zustand einen nicht aktiven übergeordneten Zustand besitzt, werden zunächst für letzteren die Schritte 1 bis 4 ausgeführt.
2. Der Zustand wird aktiviert.
3. Die Eintrittsaktivität wird ausgeführt.
4. Bei Bedarf werden implizite (geschichtlich bedingte) Eintrittsaktivitäten ausgeführt:
   a) Wenn der Zustand einen untergeordneten Zustand ohne Geschichte enthält, wird dessen Startzustand aktiviert und seine Eintrittsaktivität ausgeführt.
   b) Wenn der Zustand einen untergeordneten Zustand mit Geschichte enthält, und wenn einer der Unterzustände nach der Initialisierung des *Statechart* aktiv war, wird dieser Unterzustand aktiviert und seine Eintrittsaktivität ausgeführt. Andernfalls wird wie unter 4a verfahren.

Ausführen eines (Basis-) Zustandes:

1. Die vom Zustand wegführenden Übergänge sowie Übergänge, die aus übergeordneten Zuständen wegführen, werden in der Reihenfolge ihrer impliziten Priorität (s. Abschnitt konkurrierende Transitionen) ausgewertet.
2. Wird ein gültiger Übergang gefunden, wird dieser ausgeführt. Damit ist die Ausführung des Zustands beendet.
3. Ist kein gültiger Übergang aus dem Zustand vorhanden, werden die statische Aktivitäten und internen Transitionen ausgeführt.

4. Wenn der Zustand übergeordnete Zustände hat, werden deren statische Aktivitäten und interne Transitionen ausgeführt.

Verlassen eines Zustandes:

1. Wenn der Zustand aktive Unterzustände enthält, werden deren Austrittsaktivitäten ausgeführt. Die Austrittsaktivität des innersten (Basis-) Zustands wird zuerst ausgeführt.
2. Die Austrittsaktivität des Zustands wird ausgeführt.
3. Der Zustand wird deaktiviert.

Ausführen eines Übergangs: Übergänge werden in der Reihenfolge ihrer impliziten Priorität ausgewertet. Im Falle einer *Join*-Verknüpfung gilt das für alle in sie hineinführenden Segmente. Übergänge aus einem Unterzustand eines Hierarchiezustandes haben immer eine höhere Priorität als Übergänge aus dem Hierarchiezustand selbst.

1. Ein Übergang oder Übergangssegment wird geprüft.
2. Ist der Übergang/das Segment ungültig, wird der Übergang/das Segment mit der nächstniedrigeren impliziten Priorität geprüft.
3. Ist der Übergang/das Segment gültig, hängt der nächste Schritt davon ab, wo der Übergang/das Segment endet.

   Übergang/Segment endet in einem Zustand:

   a) Keine weiteren Übergänge oder Übergangssegmente werden geprüft. Im Fall eines Übergangssegments aus einem Pseudozustand wird das Segment in den Pseudozustand ebenfalls berücksichtigt, um einen vollständigen Übergang zu erhalten. Für die Transitionssegmente in eine *Fork*-Verknüpfung gilt das für alle diese Segmente.
   b) Die Unterzustände des Ausgangszustandes werden verlassen (s. *Verlassen eines Zustandes*).
   c) Der Ausgangszustand wird verlassen.
   d) Die Übergangsaktion wird ausgeführt.
   e) Das System tritt in den Zielzustand ein (s. *Eintritt in einen Zustand*).

   Übergang/Segment endet in einem Pseudozustand:

   a) Die von der *Choice*-Verknüpfung wegführenden Übergangssegmente werden, wie in den Schritten 1 bis 3 beschrieben, ausgewertet.

4. Wenn alle von einem Pseudozustand wegführenden Übergangssegmente ungültig sind, geht das System zurück in den Ausgangszustand, von dem aus der Pseudozustand erreicht wurde. Da das Segment in den Pseudozustand zu keinem gültigen Übergang gehört, werden die Schritte 1 bis 4 für den Übergang/das Segment mit der nächstniedrigeren impliziten Priorität ausgeführt.

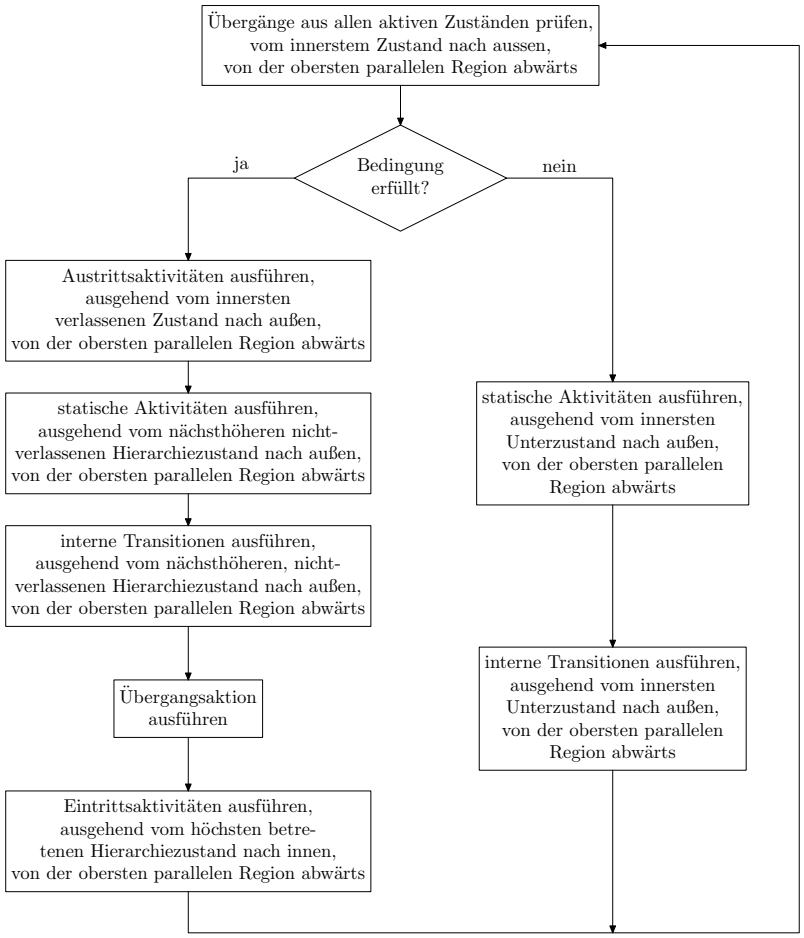


Abbildung 2.5: Auswertung eines Statecharts in Artisan RtS

5. Wenn alle von einem Zustand wegführenden Übergänge/Segmente ungültig sind, findet kein Übergang statt, das System bleibt in dem Zustand.

Der gesamte Ablauf ist zusammenfassend in Abb. 2.5 schematisch dargestellt.

2.2.3 Formale Syntax von *Statecharts* in *Artisan RtS*

Frei nach Lilius und Paltor [LP99] soll nun die Syntax eines *Statecharts* in *Artisan RtS* definiert werden. Darüberhinaus wurden dafür auch bei Pnueli und Shalev [PS91] und John [Joh99] Anleihen genommen. In Ähnlichkeit zur formalen Definition von Zustandsautomaten in *Ascet-SD* finden verallgemeinerte *Statecharts* der Definition 31 des Abschnittes 3.3 eine Anwendung. Auf deren Bedeutung wird in einem späteren Kapitel noch gesondert eingegangen werden. Dieser Abschnitt ist als formale Ergänzung zu den Aussagen in Abschnitt 2.2.1 zu sehen.

Definition 1 (*Statechart* in *Artisan RtS*) Ein Tupel $SC_{Artisan\,RtS} = (\Sigma, \Pi, T, s, l)$ über drei Mengen S, L_Σ, L_T heißt *Statechart*, wenn

(a) Σ eine endliche nichtleere Menge von Zuständen (Zustandsmenge),

(b) Π eine endliche Menge von Pseudozuständen,

(c) $T \subseteq V \times L_T \times V$ eine endliche Menge von Transitionen, wobei L_T eine (endliche) Menge von Transitionsmarkierungen ist. Die Menge der Knoten $V = \Sigma \uplus \Pi$ ist die disjunkte Vereinigung der Zustände Σ und der Pseudozustände Π.

(d) $s : V \to S$ ist die Symbolfunktion; s ordnet jedem Knoten aus V ein seiner Aufgabe entsprechendes Symbol aus der Symbolmenge S zu.

(e) $l : \Sigma \to L_\Sigma$ ist die Zustandsmarkierungsfunktion; dabei ist L_Σ eine (endliche) Menge von Zustandsmarkierungen. □

Folgend soll nur noch von *Statecharts* die Rede sein, wenn damit die *Statecharts* in *Artisan RtS* gemeint sind. Eine (formale) Sprache L, die über einem Alphabet A gebildet wird, soll im Folgenden immer mit $L \subseteq A^*$ ausgezeichnet werden und gründet sich auf den Beschreibungen von SALOMAA [Sal78]. Eine solche Sprache kann leer sein und enthält immer das leere Wort ε.

Zustände

Die Menge der Zustände Σ eines *Statecharts* $SC_{Artisan\,RtS}$ kann aufgrund ihrer Funktionalität in Teilmengen aufgeteilt und benannt werden:

Definition 2 (Zustände eines *Statecharts*) Teilmengen der Menge der Zustände eines *Statecharts* sind

(a) Σ_{is} die Menge der *Initial-* oder *Startzustände*,

(b) Σ_{ss} die Menge der *simple states* oder *Basiszustände*,

(c) Σ_{cs} die Menge der *compound states* oder *Hierarchiezustände*,

(d) Σ_{his} die Menge der *history states* oder *Zustände mit Geschichte*,

(e) Σ_{subm} die Menge der *submachine-Zustände* oder *geschlossenen Hierarchiezustände*,

(f) Σ_{synch} die Menge der *synch states* oder *Synchronisationszustände* und

(g) Σ_{fs} die Menge der *final states* oder *Endzustände*. □

Die Menge der Startzustände Σ_{is} ist eine Teilmenge von $\Sigma_{cs} \cup \Sigma_{ss}$. Die Menge der Hierarchiezustände wird unterteilt in die Menge der orthogonalen Zustände $\Sigma_{orthogonal}$ und die der seriellen Σ_{serial}. Es gilt:

$$\Sigma_{cs} = \Sigma_{orthogonal} \cup \Sigma_{serial},\ \Sigma_{orthogonal} \cap \Sigma_{serial} = \emptyset.$$

In einem seriellen Hierarchiezustand kann jeweils nur ein direkter Unterzustand aktiv sein, in einem orthogonalen Zustand müssen das mehrere sein, und zwar in jeder Region einer. Des Weiteren wird definiert, dass die Menge der Hierarchiezustände Σ_{cs} die Teilmenge der geschlossenen Hierarchie, die mit Σ_{subm} bezeichnet wird, beinhaltet:

$$\Sigma_{subm} \subseteq \Sigma_{cs} \subseteq \Sigma.$$

Die Unterzustände der Zustände der Menge Σ_{subm} werden in einem separaten *Statechart* modelliert.

Zur besseren Beschreibbarkeit der Hierarchiezustände und ihrer Unterzustände sollen folgende Definitionen beitragen:

Definition 3 (Unterzustände eines Hierarchiezustandes im *Statechart*) Die Menge aller direkten Unterzustände eines Zustandes $s \in \Sigma$ werden durch eine Funktion $children : \Sigma \rightarrow 2^{\Sigma}$ definiert. Ein Zustand $s \in \Sigma$ wird Basiszustand genannt, falls $children(s) = \emptyset$, sonst heißt er Hierarchiezustand. Für genau einen Zustand $root \in \Sigma$ gilt: $\forall s \in \Sigma, root \notin children(s)$. Der transitive und der reflexive Abschluss von *children* sind gegeben durch

$$children^*(s) = \bigcup_{i \geq 0} children^i(s) \quad \text{und} \quad children^+(s) = \bigcup_{i \geq 1} children^i(s),$$

$$\text{wobei } children^0(s) = \{s\},\ children^1(s) = children(s) \text{ und für jedes } i \geq 1,$$

$$children^{i+1}(s) = \bigcup_{s' \in children(s)} children^i(s').$$

Die Menge $children^1(s)$ sei mit *direkte Unterzustände* von s benannt. Die Funktion $children^*$ kann zu einer Menge von Zuständen erweitert werden, wenn man für eine Menge $\Omega \subseteq \Sigma$ folgendes definiert:

$$children^*(\Omega) = \bigcup_{s \in \Omega} children^*(s).$$

□

(nach [PS91])

Definition 4 (Hierarchiezustand eines Unterzustandes im *Statechart*) Es sei $v \in children(s)$, $s \in \Sigma_{cs}, v \in V$, dann ist s Hierarchiezustand von v, d. h. $parent(v) = s$. □

Als Kurzform für hierarchische Abhängigkeiten soll die in [LP99] benutzte Schreibweise verwendet werden. Ein serieller Hierarchiezustand $s_{serial} \in \Sigma_{serial}$, der den direkten

Unterzustand $v_{sub} \in V$, hat, wird demnach folgend nur noch mit $s_{serial}(v_{sub})$ bezeichnet werden. Da LILIUS und PALTOR keine brauchbare Schreibweise für Unterzustände orthogonaler Zustände mit Regionen anbieten, soll darüberhinaus folgendes abgekürzt werden: Hat ein orthogonaler Hierarchiezustand $s_{orthogonal} \in \Sigma_{orthogonal}$, in der Region $r \in \mathcal{R}(s_{orthogonal})$, einen direkten Unterzustand $v_{sub} \in V$, dann soll nur noch $s_{orthogonal}[r(v_{sub})]$ geschrieben werden.

Definition 5 (Regionen in orthogonalen Zuständen) Allen orthogonalen Hierarchiezuständen aus $\Sigma_{orthogonal}$ wird eine nichtleere geordnete Menge von *Regionen* $\mathcal{R}(s_{orthogonal})$, $s_{orthogonal} \in \Sigma_{orthogonal}$, zugeordnet, in denen Zustände unabhängig voneinander aktiv sein können. Die Ordnung auf den Regionen resultiert aus der Anordnung der Regionen $r_i \in \mathcal{R},\ i = 1, \ldots, n$, im orthogonalen Zustand. Die oberste Region des Zustandes ist r_1 und die unterste r_n. Dabei soll n die Anzahl der Regionen $n = |\mathcal{R}(s_{orthogonal})|$ sein. Eine spezielle Menge $\mathcal{R}_{his}(s_{orthogonal})$ sei die Menge aller Regionen über $s_{orthogonal}$, die eine *History*-Verknüpfung enthalten ($c_{history^{(*)}}$ benennt eine *Shallowhistory*- (*Deephistory*-) Verknüpfung und wird weiter unten definiert):

$$\mathcal{R}_{his}(s_{orthogonal}) \stackrel{\text{def}}{=} \bigcup_{\exists s_{orthogonal}[r(c_{history^{(*)}})]} s_{orthogonal}[r]$$

□

Pseudozustände

Eigenschaften von Zuständen und Transitionen eines *Statecharts* können durch Pseudozustände spezifiziert oder verändert werden. Pseudozustände können dabei als funktionale Abbildungen von Zuständen oder Transitionen verstanden werden. Folgend bezeichnet die Menge T_{pre} die Transitionen, die zum Pseudozustand hin-, und T_{post} diejenigen, die vom Pseudozustand wegführen. Für die Klärung der Funktionen *source* und *target* sei der Leser auf Definition 8 verwiesen.

Definition 6 (Pseudozustände eines *Statecharts*) In *Statecharts* existieren die Folgenden Pseudozustände aus Π mit den ihnen zugeordneten Abbildungen:

(a) $c_{initial}$ (*Initial*- oder *Startverknüpfung*), mit der Abbildung

$$c_{initial} : T_1 \times \Sigma_1 \rightarrow \Sigma_{is}$$

mit $\Sigma_1 = \Sigma \backslash \Sigma_{is}$ und $T_1 \subseteq T$, wobei $source(t_1 \in T_1) = c_{initial}$ und $target(t_1 \in T_1) \in \Sigma_1$,

(b) $c_{history}$ (*Shallowhistory*-Verknüpfung), mit den Abbildungen

$$c_{history} : \Sigma_1 \rightarrow \Sigma_{his}$$

mit $\Sigma_1 = \Sigma \setminus \Sigma_{his}$ wobei $c_{history} \in children^1(s_1 \in \Sigma_1)$ und

$$c_{history} : T_1 \times \Sigma_1 \rightarrow \Sigma_{is}$$

mit $\Sigma_1 = \Sigma \setminus \Sigma_{is}$ und $T_1 \subseteq T$ wobei $c_{history^{(*)}}$, $s_1(\in \Sigma_1) \in children^1(s_2 \in \Sigma_{cs} \setminus \Sigma_1)$,

(c) $c_{history^*}$ (*Deephistory*-Verknüpfung), mit den Abbildungen

$$c_{history^*} : \Sigma_1 \rightarrow \Sigma_{his}$$

mit $\Sigma_1 = \Sigma \setminus \Sigma_{his}$ und $\Sigma_1 = \Sigma_1 \cup children^*(\Sigma_1)$ wobei $c_{history^*} \in children^1(s_1 \in \Sigma_1)$ und

$$c_{history^*} : T_1 \times \Sigma_1 \rightarrow \Sigma_{is}$$

mit $\Sigma_1 = \Sigma \setminus \Sigma_{is}$ und $T_1 \subseteq T$ wobei $c_{history^{(*)}}, s_1(\in \Sigma_1) \in children^1(s_2 \in \Sigma_{cs} \setminus \Sigma_1)$,

(d) c_{fork} (*Fork*-Verknüpfung), mit der Abbildung

$$c_{fork} : T_0 \times T_1 \times \ldots \times T_n \rightarrow T_{ct}$$

mit $T_0 \subseteq T_{pre} \subset T$, $T_{1,\ldots,n} \subseteq T_{post} \subset T$, $target(t_0 \in T_{pre}) = c_{fork}$, $source(t_1 \in T_{post}) = c_{fork}$, $\bigcap_{i=0,\ldots,n} T_i = \emptyset$,

(e) c_{join} (*Join*-Verknüpfung), mit der Abbildung

$$c_{join} : T_0 \times T_1 \times \ldots \times T_n \rightarrow T_{ct}$$

mit $T_{1,\ldots,n} \subseteq T_{pre} \subset T$, $T_0 \subseteq T_{post} \subset T$, $target(t_0 \in T_{pre}) = c_{join}$, $source(t_1 \in T_{post}) = c_{join}$, $\bigcap_{i=0,\ldots,n} T_i = \emptyset$ und

(f) c_{choice} (*Choice*-Verknüpfung), mit der Abbildung

$$c_{choice} : T_{pre} \times T_{post} \rightarrow T_{ct}$$

mit $target(t_0 \in T_{pre}) = c_{choice}$, $source(t_1 \in T_{post}) = c_{choice}$, $T_{pre}, T_{post} \subset T$ und $T_{pre} \cap T_{post} = \emptyset$. □

Transitionen

Zustandsüberführungen finden mit Transitionen statt. Zweckmäßig für die Transformation in Abschnitt 3.3 sollen deshalb Transitionen in Verbindung mit Zuständen nach [LP99] definiert werden:

Definition 7 (Transitionen eines *Statecharts*) Eine Transition der Menge T ist ein Tripel (v, l_T, v'), wobei die Transition einen Knoten $v \in V$ zusammen mit einer Markierung $l_T \in L_T$ in einen Knoten $v' \in V$ überführt. Mindestens zwei Transitionen können mit einem Pseudozustand zu einer *compound-* oder *zusammengesetzten Transition* verknüpft werden. Die Menge der zusammengesetzten Transitionen sei mit T_{ct} benannt. Eine Teiltransition einer zusammengesetzten Transition wird auch mit (Transitions-) *Segment* bezeichnet. □

Definition 8 (Quelle und Ziel einer Transition im *Statechart*) Es seien $t = (v, l_T, v') \in T$ und $v, v' \in V$, dann kann eine Funktion $source(t) = v$ definiert werden, die die Quelle v der Transition findet. Äquivalent dazu sei $target(t) = v'$ definiert, die das Ziel v' liefert. □

Transitionsmarkierungen

Definition 9 (Transitionsmarkierung im *Statechart*) Jede Transition $t \in T$ besitzt eine Markierung $l_T \in L_T$, die sich aus folgenden Komponenten zusammensetzt:

(a) eine partielle Abbildung $trigger : T \to E$ (*Trigger*),

(b) eine partielle Abbildung $condition : T \to L_p$ (Bedingungen) und

(c) eine partielle Abbildung $action : T \to L_a$ (Übergangsaktion).

Für Transitionsmarkierungen gilt folgende Bildungsregel:

$$l_T(t) = \begin{cases} trigger(t)\, condition(t)\, /action(t) & \text{falls } condition(t) = \varepsilon, \\ trigger(t)\, [condition(t)]/\, action(t) & \text{sonst} \end{cases}$$

□

Definition 10 (*Trigger* eines *Statecharts*) Die Menge E der *Trigger* eines *Statecharts* ist in disjunkte Teilmengen zerlegbar:

(a) E_{call} heißt die Menge der *Trigger*, die mit einem Methodenaufruf verbunden sind,

(b) E_{change} heißt die Menge der *Trigger*, die bei Änderung ihres Parameters, der einen Booleschen Wert darstellt, bereitgestellt werden,

(c) E_{signal} heißt die Menge der *Trigger*, die ein explizites Signal erfordern,

(d) E_{time} heißt die Menge der *Trigger*, die nach Ablauf einer bestimmten Zeit erfolgen und

(e) $E_{completion}$ heißt die Menge der *Trigger*, die nach Beendigung einer Aktivität erfolgen.

□

Die Sprache L_p ist ein zusammengesetzter Boolescher Ausdruck und sei über $C_p = C \cup \{\&\&, ||\}$ gebildet, d. h. $L_p \subseteq C_p^*$. Die Menge C sei dabei eine Menge von Primitivbedingungen, aus denen komplexe Bedingungsausdrücke gebildet werden und $\{\&\&, ||\}$ Verknüpfungen zwischen den einzelnen Termen. Primitivbedingungen sind minimale Boolesche Ausdrücke und beinhalten Elemente der Menge W. Diese besteht aus Variablen und Methoden, die im *Statechart* definiert werden können, sowie Werten, die den Variablen zugewiesen werden können. Diese Elemente werden in Primitivbedingungen mit Operanden aus $\{==, !=, <>, >=, <=\}$ zu minimalen Booleschen Ausdrücken verknüpft. Für die Sprache L_p gelten die gleichen Bildungsregeln wie für Bedingungsausdrücke in der Programmiersprache *C*.

Die Sprache L_a unterliegt ebenso wie L_p den Konstruktionsprinzipien der Programmiersprache *C*. Sie wird über der Menge *A*, der Menge der Primitivaktionen, gebildet

Tafel 2.3: Beschränkungen der Beschriftungen von zusammengesetzten Transitionen in Statecharts

Pseudozustand	Eigenschaften der Transitionssegmente des Pseudozustandes
Startverknüpfung	t_1 existiert nicht, $trigger(t_2) = \emptyset,\ condition(t_2) = \emptyset,\ action(t_2) \in L_a$
Deep-/Shallowhistory-Verknüpfung	$trigger(t_1) \in E,\ condition(t_1) \in L_p,\ action(t_1) \in L_a$ $trigger(t_2) = \emptyset,\ condition(t_2) = \varepsilon,\ action(t_2) \in L_a$
Fork-Verknüpfung	$trigger(t_1) \in E,\ condition(t_1) \in L_p,\ action(t_1) \in L_a$ $trigger(t_2) = \emptyset,\ condition(t_2) = \varepsilon,\ action(t_2) \in L_a$
Join-Verknüpfung	$trigger(t_1) = \emptyset,\ condition(t_1) = \varepsilon,\ action(t_1) \in L_a$ $trigger(t_2) \in E,\ condition(t_2) \in L_p,\ action(t_2) \in L_a$
Choice-Verknüpfung	$trigger(t_1) \in E,\ condition(t_1) \in L_p,\ action(t_1) \in L_a$ $trigger(t_2) = \emptyset,\ condition(t_2) \in L_p,\ action(t_2) \in L_a$

($L_a \subseteq A^*$). Die Menge A besteht aus Elementen aus W, die mit einfachen Operanden aus $\{+, -, *, /, \%\}$ verknüpft und mit einem Semikolon (;) abgeschlossen sind.

Für die Markierungen der Segmente von zusammengesetzten Transitionen seien die Einschränkungen in Taf. 2.3 aufgeführt, die nach Taf. 2.1 erstellt wurde. Dabei sei $t_1 \in T_{pre}$ eine Transition, die zum Pseudozustand hin-, und $t_2 \in T_{post}$ diejenige, die vom Pseudozustand wegführt. Andere Möglichkeiten der Markierung für die aufgeführten Transitionen als die gezeigten sind nicht erlaubt.

Zustandsmarkierungen

Definition 11 (Zustandsmarkierung im *Statechart*) Gibt es eine Sprache L_n über einer Menge $CHAR = \{ a, \ldots, z,\ A, \ldots, Z,\ 0, \ldots, 9,\ \_ \}$ mit $L_n \subseteq CHAR^* - \{\varepsilon\}$ für die Beschreibung der Zustandsnamen und eine Sprache L_a über die Menge aller (Primitiv-) Aktionen A mit $L_a \subset A^*$ für die Beschreibung von Aktivitäten und $s \in \Sigma$ ist ein Zustand eines *Statecharts*, so gibt es folgende Funktionen auf s, die eine Zustandsmarkierung bilden:

(a) $name : \Sigma \rightarrow L_n$ ist der Zustandsname,

(b) $entry : \Sigma \rightarrow L_a$ ist die Eintrittsaktivität,

(c) $exit : \Sigma \rightarrow L_a$ die Austrittsaktivität,

(d) $do : \Sigma \rightarrow L_a$ ist statische Aktivität und

(e) $internalTransition : \Sigma \rightarrow \{L_T\}$ ist die *interne Transition* von s, die aus mehreren Markierungen aus L_T bestehen kann. □

Die Zustandsmarkierungen werden durch Voranstellen von ‚Entry/' vor die Eintrittsaktivität, von ‚Exit/' vor die Austrittsaktivität und von ‚do:' vor die statische Aktivität komponiert.

Symbole

Definition 12 (Symbole eines *Statecharts*) Jedem Knoten $v \in V$ ist mit der Symbolfunktion s entsprechend seiner Funktion ein eindeutiges Symbol $s(v) \in S$ zugeordnet:

- $s : \Sigma_{ss} \cup \Sigma_{serial} \setminus \Sigma_{subm} \rightarrow$ 
- $s : \Sigma_{subm} \rightarrow$ 

- $s : \Sigma_{orthogonal} \rightarrow$ 
- $s : \Sigma_{fs} \rightarrow$ 
- $s : \Sigma_{synch} \rightarrow$ 
- $s : c_{initial} \mapsto$
- $s : c_{history^*} \mapsto$ 

- $s : c_{history} \mapsto$ 

- $s : c_{fork} \mapsto$ 
- $s : c_{join} \mapsto$
- $s : c_{choice} \mapsto$ 

□

2.3 Das Werkzeug *Ascet-SD*

Ascet-SD (*Advanced Simulation and Control Engineering Tool-Software Development*) der Firma *ETAS* ist ein *CASE-Tool* für die Entwicklung von Software für Steuergeräte im Kraftfahrzeugbereich. Die Modellbildung erfolgt mit Hilfe von Blockdiagrammen, hierarchischen Zustandsautomaten, *Java*-ähnlichem Pseudo-Code oder *C*-Code. Die Steuersoftware für eingebettete Systeme kann automatisch aus den Diagrammen generiert werden. Der Hersteller bietet eine Export-Funktion von Modellen nach *XML* zum Austausch mit anderen Werkzeugen; ein Import für diese Sprache existiert nicht.

Für die Beschreibung der Zustandsautomaten in *Ascet-SD* werden, wie bereits bei der Beschreibung der *Statecharts* in *Artisan RtS*, zunächst in Abschnitt 2.3.1 die Strukturelemente aufgezählt und informell beschrieben. Daran schließt sich die informelle Semantik in Abschnitt 2.3.2 und die formale Syntax in Abschnitt 2.3.3 an.

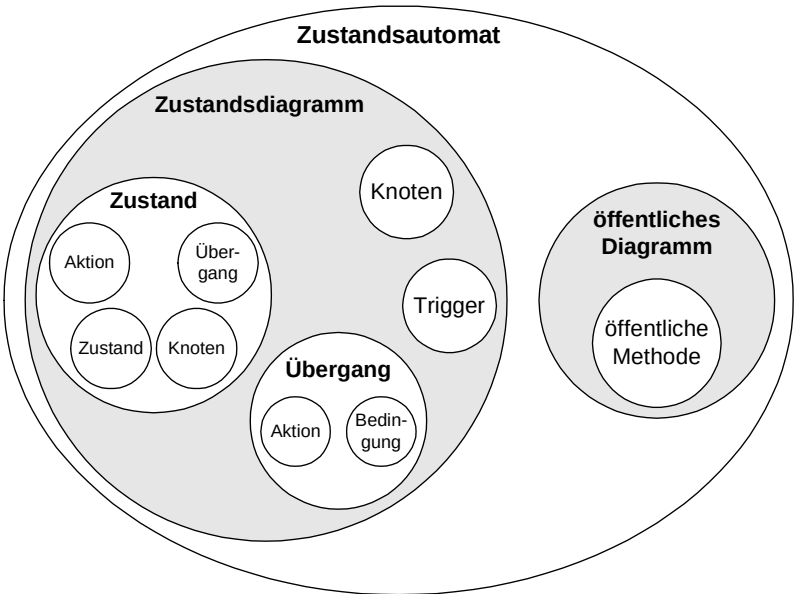


Abbildung 2.6: Schema des Zustandsautomaten in Ascet-SD (Quelle: [ETA02])

2.3.1 Hierarchische Zustandsautomaten von *Ascet-SD*

Komplexe dynamische Anwendungen können in *Ascet-SD* durch Zustandsautomaten beschrieben werden. Im Zustandsautomaten-Editor von *Ascet-SD* definiert man dazu graphisch Zustände und Übergänge. Beim Übergang zwischen Zuständen können Bedingungen und Übergangsaktionen, beim Eintritt, Verweilen oder Verlassen eines Zustands dagegen Bedingungen und Aktivitäten[4] definiert werden. Eindeutigkeit wird durch Attributierung der Übergänge erreicht, d. h. durch Belegung der Übergänge mit Ausführungsprioritäten. Die Übergänge werden dabei in der Reihenfolge ihrer Prioritäten geprüft und gegebenenfalls durchgeführt.

Ein Zustandsautomat besteht bei *Ascet-SD* aus einer endlichen Anzahl von Zuständen und Übergängen zwischen diesen. Beim Entwurf der Automaten stützte sich der Hersteller auf die Beschreibungen in [Har87] und [HP88]. In der Abb. 2.6 werden schematisch die Komponenten eines Zustandsautomaten gezeigt. Die graphischen Komponenten von *Ascet-SD*, die in Abb. 2.7 dargestellt sind, charakterisieren den wesentlichen Sprachumfang von Zustandsautomaten. Auf sie soll im Folgenden einzeln eingegangen werden, dabei wird möglichst die Nomenklatur des Handbuches [ETA02] benutzt.

Zustand: Jeder Zustand wird graphisch als abgerundetes Rechteck dargestellt und enthält einen eindeutigen Namen. Einer von mehreren Zuständen muss stets als *Anfangszu-*

[4] Anders als im Handbuch zu *Ascet-SD* [ETA02] wird in dieser Arbeit bei Zuständen nicht von Aktionen sondern von Aktivitäten zu lesen sein. Eine Unterscheidung wird wegen des speziellen Verhaltens bei Aktivitäten notwendig: Eine Aktivität ist im Gegensatz zur Aktion durch Ereignisse abbrechbar, noch bevor sie vollständig beendet werden kann.

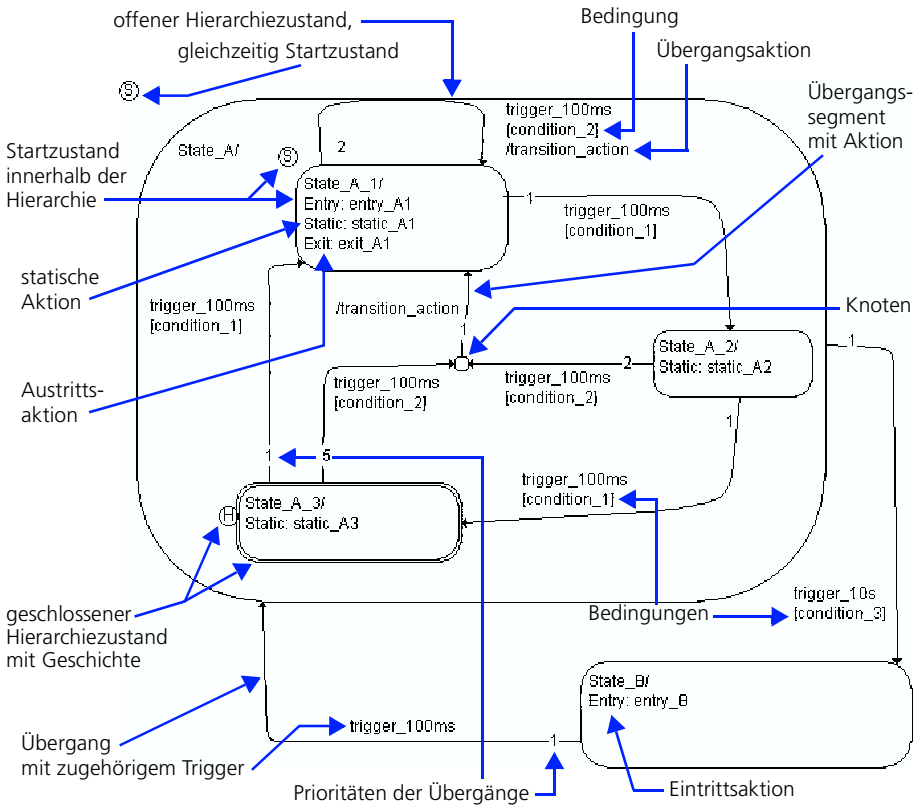


Abbildung 2.7: Komponenten der Zustandsautomaten in Ascet-SD (Quelle: [ETA02])

stand gekennzeichnet sein; dies ist der Zustand, in dem sich der Automat zu Beginn befindet.

Die Zustandsautomaten erlauben es, Zustände zu gruppieren und als s. g. *Hierarchiezustände* zusammenzufassen. *Basiszustände* werden solche Zustände genannt, die keine weiteren Zustände enthalten. Zustände, die eine *Geschichte* haben, dienen dazu, zukünftige Aktivitäten auf vorherigen aufzubauen.

Die Zustandsautomaten in *Ascet-SD* erlauben im Gegensatz zu den in [Har87] definierten keine Parallelität von Zuständen, d. h. es kann immer nur ein Zustand aktiv sein.

Aktivitäten: Zustände können *Eintritts-*, *statische* und *Austrittsaktivitäten* aufweisen. Alle Aktivitäten sind optional.

Hierarchie: Innerhalb eines Hierarchiezustandes bilden die Unterzustände ihrerseits einen Zustandsautomaten. Zustände innerhalb eines Hierarchiezustandes können Übergänge zu anderen Zuständen haben, welche sich nicht innerhalb desselben Hierarchiezustandes befinden. Zu Beginn befindet sich ein Hierarchiezustand immer in einem *Startzustand*. Ein Übergang heraus aus einem

Hierarchiezustand beinhaltet automatisch das Verlassen des gerade aktiven Unterzustands. Ein Übergang von einem Unterzustand kann über die Grenzen von Hierarchiezuständen hinweg zu einem anderen Unterzustand führen. Wenn ein Unterzustand aktiv ist, ist dessen übergeordneter Hierarchiezustand ebenfalls aktiv.

Um bessere Übersichtlichkeit zu gewährleisten, bietet *Ascet-SD* die Möglichkeit, Unterzustände in einem separaten Zustandsautomaten anzulegen. Eine solche Konstruktion wird bei *Ascet-SD* als geschlossene Hierarchie bezeichnet. Die Hierarchiezustände, die solche Unterzustände besitzen, werden mit doppelter Begrenzungslinie ausgezeichnet und sollen hier in Anlehnung an die *UML* als *submachine*-Zustände bezeichnet werden. Transitionen, die von oder zu einem Zustandsautomaten mit niedrigerer (höherer) Ebene verlaufen sollen, werden an *Pins* (o. a. *Anschlüsse*) angeschlossen und werden *stubbed* Transition genannt. Dabei kann entweder nur die Teiltransition vor dem *Pin* oder die nach dem *Pin* eine Beschriftung haben. *Pins* werden im Zustandsautomaten mit einem rechteckig umrahmten alphanumerischen Bezeichner des *Pins* auf der Begrenzungslinie des dazugehörigen Hierarchiezustandes gekennzeichnet. Diese Art der Konstruktion ähnelt stark den *Stub-States* der *Statecharts* der *UML*.

Startzustand: Ein *Startzustand* zeigt an, welcher von mehreren Zuständen auf derselben Hierarchieebene als erster aktiviert wird. Von einer Menge von Unterzuständen eines Hierarchiezustandes muss mindestens einer als Startzustand ausgezeichnet sein. Die Auszeichnung erfolgt mittels einem umzirkelten ‚S' direkt vor einem Startzustand.

Geschichte: Ein Hierarchiezustand wird mit einer Geschichte belegt, wenn die zuletzt gültige Konfiguration innerhalb dieses Zustandes wiederhergestellt werden soll, d. h. der Zustand, der beim Verlassen des Oberzustandes aktiv war, wird wieder aktiviert. Ein Hierarchiezustand mit einer Geschichte wird mit einem umzirkelten ‚H' vor dem Zustand ausgezeichnet. Eine Konstruktion, die aus der *UML* als *deephistory* bekannt ist, existiert in *Ascet-SD* nicht.

Übergang: Ein Übergang (auch: Transition) verbindet zwei Zustände oder einen Zustand und eine *Junction-Verknüpfung*. Er wird durch eine gerichtete Kante vom Ausgangszustand zum Zielzustand dargestellt. Jeder Übergang ist mit einer Priorität markiert, wobei die höhere den Übergang angibt, der zuerst ausgewertet wird. Zwei Übergänge von ein und demselben Zustand mit derselben Priorität dürfen nicht existieren.

Übergänge von Zustand zu Zustand müssen unbedingt mit einem *Trigger* und können optional mit *Bedingungen* und *Übergangsaktionen* beschriftet sein. *Bedingungen* werden im Zustandsautomaten in eckigen Klammern dargestellt, *Übergangsaktionen* mit einem vorangestellten ‚/'. Ein Übergang ist gültig, wenn sein Anfangs-

zustand aktiv ist, ein *Trigger*-Ereignis auftritt und seine Bedingung erfüllt ist. Beim Übergang werden Übergangsaktionen ausgeführt, wodurch *Daten* geändert werden können. Nach dem Übergang ist der Anfangszustand inaktiv und der Zielzustand aktiv.

Bedingung: Ein Übergang von einem Zustand in einen anderen findet genau dann statt, wenn der Boolesche Ausdruck einer Bedingung, die einem Übergang zugeteilt ist, wahr ist.

Übergangsaktion: Übergänge und die von einer *Junction*-Verknüpfung wegführenden Übergangssegmente können optionale *Übergangsaktionen* haben.

Junction-Verknüpfung: Ein Übergang kann durch eine *Junction*-Verknüpfung[5] unterbrochen und in zwei Segmente geteilt werden. In diesem Fall verbindet ein Segment den Ausgangszustand mit der Verknüpfung und kann mit einer Bedingung assoziiert sein, das andere verbindet die Verknüpfung mit dem Zielzustand und kann mit Bedingung und Übergangsaktion belegt sein. *Junction*-Verknüpfungen repräsentieren somit Verzweigungsstellen im Zustandsautomaten. Sie erleichtern die Darstellung verschiedener Übergänge, indem sie diese in einzelne Segmente zerlegen. Falls die eingehenden Segmente mit einem *Trigger* beschriftet sind, kann das bei den ausgehenden Segmenten nicht der Fall sein und umgekehrt. Die Verknüpfungen werden durch eine Kreislinie im Automaten dargestellt.

Trigger: Trigger aktivieren die Ausführung des Zustandsautomaten. Alle Übergänge aus dem aktuellen Zustand werden in der Reihenfolge ihrer Priorität geprüft und ggf. wird ein Übergang ausgeführt. *Ascet-SD* bietet zum einen *Timeout-Trigger*, die den Zustandsautomaten jeweils nach Verstreichen einer bestimmten, einstellbaren Zeit aktivieren. Darüber hinaus ist es möglich, in Abhängigkeit einer externen Variable zu *triggern*. Diese *Trigger* sind aus der *UML* als *Change-Event* bekannt. Der zweite erwähnte *Trigger* ist für diese Arbeit unwesentlich und kommt nicht zur Anwendung, da dafür Zustandsautomaten-externe Informationen notwendig sind.

Daten: Daten (oder *Variablen*) dienen der Speicherung und Verarbeitung von numerischen und nichtnumerischen Werten. In Zustandsautomaten von *Ascet-SD* werden folgende Datentypen unterschieden:

Basismodelltyp: Basismodelltypen werden zum Modellieren physischen Verhaltens verwendet. *Ascet-SD* stellt die *skalaren* Basistypen, wie sie in Taf. 2.4 aufgeführt sind, zur Verfügung. Für sie sind Operationen wie z. B. Addition oder Vergleich definiert. Skalare Basistypen können zu *zusammengesetzten Typen* kombiniert werden (z. B. *Array*). Weitere Datentypen sollen hier aufgrund ihrer Irrelevanz für diese Arbeit nicht aufgeführt werden.

[5] Der Leser wird im Handbuch [ETA02] den Begriff *Knoten* finden. In Anlehnung an die *UML* soll hier aber stattdessen der Begriff *Junction*-Verknüpfung verwendet werden.

Tafel 2.4: Skalare Basismodelltypen in Ascet-SD und ihre Standard-Implementierungen

Datentyp	Wert	Implementierungstyp
cont	beliebige Zahl	*real64*
udisc	positive *Integer*-Zahl	*unsigned int32*
sdisc	negative *Integer*-Zahl	*signed int32*
log	Boolescher Wert	*int16*

Es werden Arten von Basismodelltypen unterschieden: *Variablen* (Wert lässt sich durch Berechnungen im Zustandsautomaten verändern), *Parameter* (Wert wird mit dem Aufruf eines Zustandsautomaten übergeben und lässt sich durch Berechnungen im Zustandsautomaten nicht verändern) und *Konstanten* (fester Wert).

Implementierungstyp: Implementierungstypen werden die Datentypen des spezifischen unterliegenden *C-Compilers* genannt. Diese sind z. B. *unsigned byte* (*uint8*), *signed word* (*sint16*), *float*. Im Gegensatz zu Basismodelltypen stellen Implementierungstypen keine physikalischen Größen dar. Die Implementierung (automatisierte Umsetzung der physikalischen Aufgabenstellung (des Modells) in ausführbarem Festkomma-Code durch *Ascet-SD*) dient der Umwandlung der Basismodelltypen in Implementierungstypen.

Die Standardzuordnungen von Implementierungstypen zu ihren Basistypen, wie sie Taf. 2.4 zeigt, können vom Benutzer vor dem automatischen Erstellen des ausführbaren Codes verändert werden.

ESDL: ESDL (*Embedded Software Description Language*) wurde speziell als Modelliersprache im Umfeld der Kraftfahrzeugproduktion entwickelt. In *Ascet-SD* wird *ESDL* verwendet, um Aktionen, Aktivitäten und Bedingungen zu spezifizieren. Im Gegensatz zur Entwicklung von *C*-Code ist die Textspezifikation von Steuer- und Regelalgorithmen in *ESDL* eine portable physikalische Beschreibung. Die Sprache *ESDL* orientiert sich stark an der Programmiersprache *Java*, deshalb sind auch beide Syntaxen größtenteils identisch.

2.3.2 Semantik von Zustandsautomaten in *Ascet-SD*

Der folgende Abschnitt wurde in Anlehnung an das Handbuch von *Ascet-SD* [ETA02] erstellt:

Grundlagen

Die Semantik beschreibt, wie ein Zustandsautomat interpretiert und ausgeführt wird und in welcher Reihenfolge dabei die Übergangsaktionen und Aktivitäten durchgeführt wer-

den. Die Kenntnis der Semantik von Zustandsautomaten ist für die Erstellung geeigneter Zustandsautomaten, die Transformation in andere Modellierungsimplementationen und die Generierung von effizientem Code unerlässlich. Verschiedene Möglichkeiten der Realisierung ergeben unterschiedliches Verhalten bei der Simulation und im ausführbaren Code. Die Semantik von Zustandsautomaten enthält Regeln für die *Abarbeitung von Zuständen*, die *Auswahl von Übergängen* und die *Abarbeitung von Übergängen*.

Ein Zustandsautomat besteht aus einer endlichen Anzahl von Zuständen. Jeder Zustand repräsentiert einen Zustand, in dem sich ein System befinden kann, zum Beispiel ob eine Tür verriegelt, geöffnet oder geschlossen ist. Der Zustandsautomat muss sich immer in einem seiner Zustände befinden. Unter bestimmten Umständen ändert sich der Zustand des Systems. Diese Zustandsänderungen werden durch Übergänge zwischen den verschiedenen Zuständen modelliert. Für jeden möglichen Übergang, der stattfinden soll, muss eine Bedingung erfüllt werden. Wird eine Transition durch einen *Pin* unterbrochen, dann wird bei der Auswertung der *Pin* entfernt und beide Teilsegmente der Transition zu einer zusammengefügt.

Ein Zustandsautomat wird durch ein Ereignis aktiviert, das *Trigger*-Ereignis. Ein *Trigger* ist eine öffentliche Methode des Zustandsautomaten. Tritt ein *Trigger*-Ereignis auf, so reagiert das System mit der Ausführung von Übergangsaktionen (z. B. Erzeugung eines Signals, Ändern eines Variablenwertes oder dem Übergehen in einen anderen Zustand).

Bevor Zustandsänderungen stattfinden können, wird der Zustandsautomat *initialisiert*. Dabei wird der Startzustand des Systems aktiviert. Ist der Startzustand ein Hierarchiezustand, wird auch der Startzustand innerhalb der Hierarchie aktiviert. Dabei wird im Gegensatz zu *Statecharts* von *Artisan RtS* keine Eintrittsaktivität ausgeführt.

Eintrittsaktivität: Die Eintrittsaktivität eines Zustands wird ausgeführt, wenn ein Übergang in diesen Zustand stattfindet. Der Zustand wird aktiviert, ehe die Ausführung der Eintrittsaktivität beginnt.

statische Aktivität: Die statische Aktivität eines Zustands wird ausgeführt, wenn der Zustand aktiv ist und ein *Trigger*-Ereignis eintritt, das keinen Übergang aus dem Zustand heraus zur Folge hat. Bei einem Übergang zwischen zwei Unterzuständen desselben Hierarchiezustands führt der Hierarchiezustand (der ja nicht verlassen wird) seine statische Aktivität aus und beendet sie, nachdem der Ausgangszustand verlassen wurde, aber bevor die Übergangsaktion ausgeführt wird.

Austrittsaktivität: Die Austrittsaktivität eines Zustands wird ausgeführt, wenn ein Übergang aus diesem Zustand heraus stattfindet. Der Zustand wird inaktiv, wenn die Ausführung der Austrittsaktivität beendet ist.

Übergangsaktion: Die Übergangsaktion eines Übergangs wird ausgeführt, nachdem der Ausgangszustand verlassen und deaktiviert wurde und bevor der Zielzustand aktiviert wird.

Trigger-Ereignisse können die Eintritts-, statische oder Austrittsaktivität eines Zustandes abbrechen, falls diese noch nicht abgeschlossen ist. Eine Zustandsüberführung benötigt keine Zeit. Die Aktivitäten des Zustandes benötigen soviel Zeit, bis die Abarbeitung der Teilaktivitäten abgeschlossen ist.

Konkurrierende Transitionen

Jeder Zustand kann Übergänge zu mehr als einem anderen Zustand haben. Um das Verhalten des Zustandsautomaten deterministisch zu machen, muss jedem Übergang eine Priorität zugeordnet werden. Durch die Priorität wird die Reihenfolge festgelegt, in der die zu den Übergängen gehörenden Bedingungen geprüft werden. Sobald die Auswertung einer Bedingung den Wert *wahr* ergibt, findet der zugehörige Übergang statt und alle anderen Bedingungen, die zu Übergängen mit niedrigeren Prioritäten gehören, werden nicht geprüft. Wenn die Auswertung bei keiner Bedingung *wahr* ergibt, bleibt der Zustand unverändert und es wird die statische Aktivität ausgeführt.

Junction-Verknüpfung

Junction-Verknüpfungen repräsentieren Verzweigungsstellen im Zustandsautomaten. Sie erleichtern so die Darstellung verschiedener Übergänge, indem sie diese in einzelne Segmente zerlegen. Trifft eine Transition in der Verknüpfung ein, werden an dieser Stelle die Bedingungen der ausgehenden Transition entsprechend der Reihenfolge ihrer Prioritätsbeschriftung geprüft und die mit der passenden Bedingung für den Zustandsübergang benutzt. Bedingungen und Aktionen der von der Verknüpfung ausgehenden Transitionssegmente können dabei nicht auf Variablenänderungen in den Aktionen der Segmente dynamisch reagieren.

Hierarchische Zustände

Nach Aktivierung des Zustandsautomaten werden die Bedingungen der Übergänge geprüft. Die Priorität wird durch die hierarchische Reihenfolge festgelegt. Die höchste hierarchische Ebene hat die höchste Priorität, d. h. die Bedingungen für die Übergänge auf den oberen hierarchischen Ebenen werden zuerst geprüft. Wenn ein Hierarchiezustand verlassen wird, werden die aktuellen Unterzustände gleichfalls verlassen. Der innerste Unterzustand wird zuerst verlassen, der äußerste Hierarchiezustand zuletzt. Beim Eintritt in einen Hierarchiezustand ist die Reihenfolge, in der die Eintrittsaktivitäten ausgeführt werden, vom äußersten Hierarchiezustand zum innersten (Unter-) Zustand, d. h. zuerst erfolgt der Eintritt in den äußersten Zustand und zuletzt der Eintritt in den innersten Zustand. Findet kein Übergang statt, werden die statischen Aktivitäten von innen nach außen ausgeführt, d. h. zuerst die statische Aktivität des innersten Unterzustands und zuletzt die statische Aktivität des äußersten Hierarchiezustands.

Übergang in einen Hierarchiezustand ohne Geschichte: Wird ein Hierarchiezustand betreten, gibt es zwei Möglichkeiten: Entweder erfolgt der Eintritt in den Anfangszustand des Hierarchiezustands. In diesem Falle hat der Hierarchiezustand den Unterzustand vergessen, in dem er sich befand, als er verlassen wurde. Die andere Möglichkeit ist, dass der Eintritt in den letzten aktiven Unterzustand erfolgt. In diesem Falle hat der Hierarchiezustand eine Geschichte. Jedem Hierarchiezustand kann eine Geschichte zugeordnet werden. Beim erstmaligen Eintritt in einen Hierarchiezustand mit Geschichte erfolgt der Eintritt in den Anfangszustand dieses Hierarchiezustands.

Übergang innerhalb eines Hierarchiezustands: Falls ein Übergang innerhalb eines Hierarchiezustands stattfindet, verbleibt der Zustandsautomat in diesem Hierarchiezustand. Daher wird auch die statische Aktivität dieses Hierarchiezustands ausgeführt, ebenso wie die aller Hierarchiezustände, in denen der betreffende Zustand enthalten ist. Diese werden nach allen Austrittsaktivitäten und vor der Übergangsaktion in der Reihenfolge vom innersten zum äußersten Hierarchiezustand ausgeführt.

Übergang zwischen Hierarchiezuständen: Findet ein Übergang zwischen Hierarchiezuständen statt, werden vor dem Übergang alle aktiven Unterzustände des Quellzustandes verlassen und danach die entsprechenden Zustände des Zielzustandes von außen nach innen betreten.

Übergang zwischen Unterzuständen verschiedener Hierarchien: Übergänge können darüberhinaus auch direkt vom Unterzustand eines Hierarchiezustands in den Unterzustand eines anderen Hierarchiezustands gehen. Dabei werden beim Verlassen/Betreten der Unterzustände auch die dazugehörigen Hierarchiezustände betreten/verlassen.

Übergang von einem Unterzustand in einen Hierarchiezustand: Wenn der Übergang von einem Unterzustand nicht in einen anderen Unterzustand führt, sondern in den Hierarchiezustand, ist die Vorgehensweise fast dieselbe. Der Unterzustand wird verlassen, der Hierarchiezustand nicht. Abhängig davon, ob der Hierarchiezustand eine Geschichte hat oder nicht, wird entweder der zuletzt aktive Unterzustand aktiviert oder der Startzustand innerhalb der Hierarchie.

Kein Übergang: Findet kein Übergang statt, bleibt der aktivierte Zustand aktiv und die statische Aktivität wird ausgeführt. Ist der Zustand ein Unterzustand, werden die statischen Aktivitäten der dazugehörigen Hierarchiezustände ebenfalls ausgeführt.

Zusammenfassung

Initialisierung des Zustandsautomaten: Der Startzustand des Systems wird aktiviert. Ist der Startzustand ein Hierarchiezustand, wird auch der Startzustand innerhalb der Hierarchie aktiviert. Dabei wird *keine* Eintrittsaktivität ausgeführt.

Eintritt in einen Zustand:

1. Wenn der Zustand einen nicht aktiven übergeordneten Zustand hat, werden zunächst für letzteren die Schritte 1 bis 4 ausgeführt.
2. Der Zustand wird aktiviert.
3. Die Eintrittsaktivität wird ausgeführt.
4. Bei Bedarf werden implizite (geschichtlich bedingte) Eintrittsaktivitäten ausgeführt:
   a) Enthält der Zustand einen untergeordneten Zustandsautomaten ohne Geschichte, wird dessen Startzustand aktiviert und seine Eintrittsaktivität ausgeführt.
   b) Wenn der Zustand einen untergeordneten Zustandsautomaten mit Geschichte enthält, und wenn einer der Unterzustände nach der Initialisierung des Zustandsautomaten aktiv war, wird dieser Unterzustand aktiviert und seine Eintrittsaktivität ausgeführt. Andernfalls wird wie unter 4a verfahren.

Ausführen eines (Basis-) Zustandes:

1. Die vom Zustand wegführenden Übergänge sowie Übergänge, die aus übergeordneten Zuständen wegführen, werden in der Reihenfolge ihrer Priorität ausgewertet.
2. Wird ein gültiger Übergang gefunden, wird dieser ausgeführt. Damit ist die Ausführung des Zustands beendet.
3. Ist kein gültiger Übergang aus dem Zustand vorhanden, wird die statische Aktivität ausgeführt.
4. Wenn der Zustand übergeordnete Zustände hat, werden deren statische Aktivitäten ausgeführt.

Verlassen eines Zustandes:

1. Wenn der Zustand aktive Unterzustände enthält, werden deren Austrittsaktivitäten ausgeführt. Die Austrittsaktivität des innersten (Basis-) Zustands wird zuerst ausgeführt.
2. Die Austrittsaktivität des Zustands wird ausgeführt.
3. Der Zustand wird deaktiviert.

Ausführen eines Übergangs: Übergänge werden in der Reihenfolge ihrer Priorität ausgewertet. Übergänge aus einem Hierarchiezustand haben immer eine höhere Priorität als Übergänge aus den Unterzuständen dieses Hierarchiezustands.

1. Ein Übergang oder Übergangssegment wird geprüft.
2. Ist der Übergang/das Segment ungültig, wird der Übergang/das Segment mit der nächstniedrigeren Priorität geprüft.
3. Ist der Übergang/das Segment gültig, hängt der nächste Schritt davon ab, wo der Übergang/das Segment endet.

   Übergang bzw. Segment endet in einem Zustand:

   a) Keine weiteren Übergänge oder Übergangssegmente werden geprüft. Im Fall eines Übergangssegments aus einer *Junction*-Verknüpfung wird das Segment in die fragliche Verknüpfung ebenfalls berücksichtigt, um einen vollständigen Übergang zu erhalten.
   b) Die Unterzustände des Ausgangszustandes werden verlassen (s. *Verlassen eines Zustandes*).
   c) Der Ausgangszustand wird verlassen.
   d) Die Übergangsaktion wird ausgeführt.
   e) Das System tritt in den Zielzustand ein (s. *Eintritt in einen Zustand*).

   Übergang bzw. Segment endet in einer Junction-Verknüpfung:

   a) Die von einer *Junction*-Verknüpfung wegführenden Übergangssegmente werden wie in den Schritten 1 bis 3 beschrieben ausgewertet.
4. Wenn alle von einer *Junction*-Verknüpfung wegführenden Übergangssegmente ungültig sind, geht das System zurück in den Ausgangszustand, von dem aus die Verknüpfung erreicht wurde. Da das Segment in die Verknüpfung zu keinem gültigen Übergang gehört, werden die Schritte 1 bis 4 für den Übergang/das Segment mit der nächstniedrigeren Priorität ausgeführt.
5. Wenn alle von einem Zustand wegführenden Übergänge bzw. Segmente ungültig sind, findet kein Übergang statt und das System bleibt in dem Zustand.

Der gesamte Ablauf ist zusammenfassend in Abb. 2.8 schematisch dargestellt.

2.3.3 Formale Syntax von Zustandsautomaten in *Ascet-SD*

Ähnlich der formalen Definition von *Statecharts* in *Artisan RtS* finden hier verallgemeinerte *Statecharts* der Definition 31 des Abschnittes 3.3 eine Anwendung. Auf deren Bedeutung wird in einem späteren Kapitel noch gesondert eingegangen werden. Eingang erfährt dabei die Arbeit von Lilius und Paltor [LP99]. Ebenfalls verwendet wurden Pnueli und Shalev [PS91] sowie John [Joh99]. Dieser Abschnitt ist als formale Ergänzung zu den Aussagen in Abschnitt 2.3.1 zu sehen.

Definition 13 (Zustandsautomat in *Ascet-SD*) Ein 5-Tupel $ZA_{Ascet\text{-}SD} = (\Sigma, \Pi, T, s, l)$ über drei Mengen S, L_Σ, L_T heißt Zustandsautomat, wenn

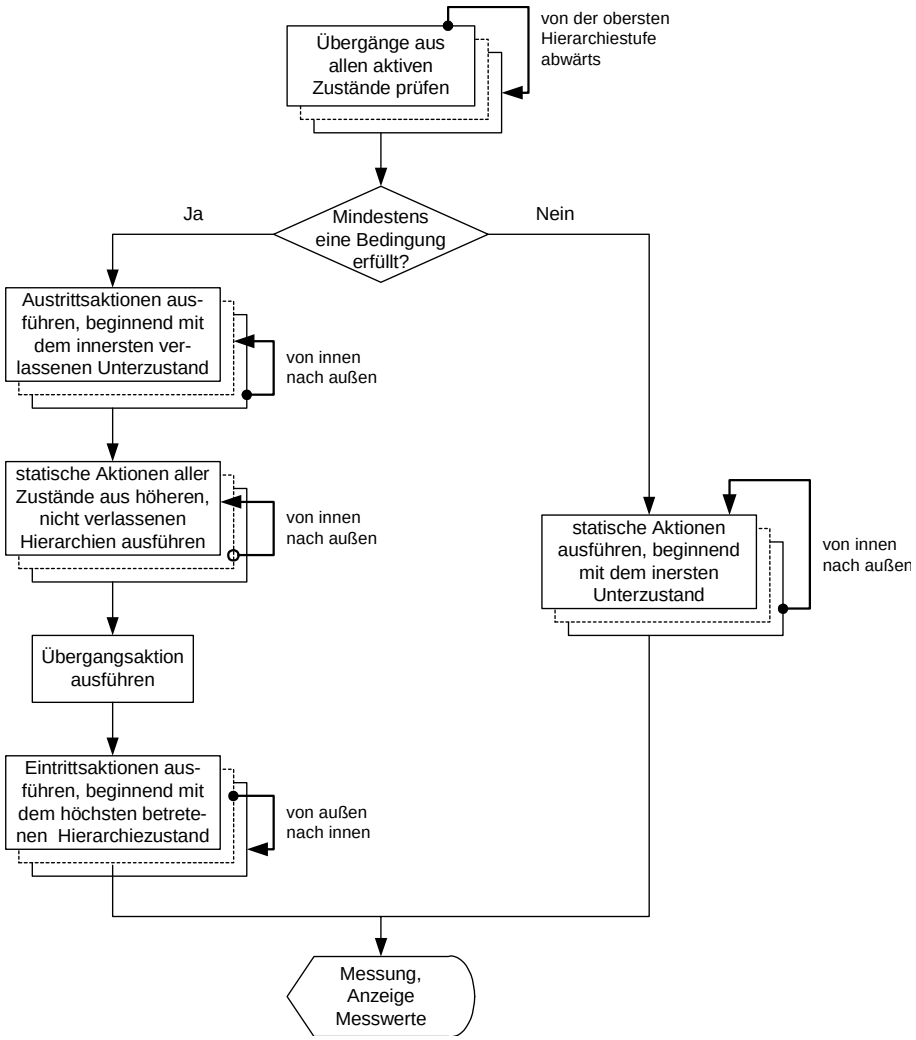


Abbildung 2.8: Auswertung eines Zustandsautomaten in Ascet-SD (Quelle: [ETA02])

(a) Σ eine endliche nichtleere Menge von Zuständen (Zustandsmenge),

(b) Π eine endliche Menge von Pseudozuständen,

(c) $T \subseteq V \times L_T \times V$ eine endliche Menge von Transitionen, wobei L_T eine (endliche) Menge von Transitionsmarkierungen ist. Die Menge der Knoten $V = \Sigma \uplus \Pi$ ist die disjunkte Vereinigung der Zustände Σ und der Pseudozustände Π.

(d) $s : V \to S$ ist die Symbolfunktion; s ordnet jedem Knoten aus V ein seiner Aufgabe entsprechendes Symbol aus der Symbolmenge S zu.

(e) $l : \Sigma \to L_\Sigma$ ist die Zustandsmarkierungsfunktion; dabei ist L_Σ eine (endliche) Menge von Zustandsmarkierungen. □

Folgend soll nur noch von Zustandsautomaten die Rede sein, wenn damit die Zustandsautomaten in *Ascet-SD* gemeint sind. Eine (formale) Sprache L die über einem Alphabet A gebildet wird, soll folgend immer mit $L \subseteq A*$ ausgezeichnet werden und gründet sich auf den Beschreibungen von Salomaa [Sal78]. Eine solche Sprache kann leer sein und enthält immer das leere Wort ε.

Zustände

Die Menge der Zustände Σ eines Zustandsautomaten $ZA_{Ascet\text{-}SD}$ kann aufgrund ihrer Funktionalität in Teilmengen aufgeteilt und benannt werden:

Definition 14 (Zustände eines Zustandsautomaten) Teilmengen der Menge der Zustände eines Zustandsautomaten sind

(a) Σ_{is} die Menge der *Initial-* oder *Startzustände*,

(b) Σ_{ss} die Menge der *simple states* oder *Basiszustände*,

(c) Σ_{cs} die Menge der *compound states* oder *Hierarchiezustände*,

(d) Σ_{his} die Menge der *history states* oder *Zustände mit Geschichte* und

(e) Σ_{subm} die Menge der *Submachine-Zustände*. □

Die Menge der Startzustände Σ_{is} ist eine Teilmenge von $\Sigma_{cs} \cup \Sigma_{ss}$. Die Menge der Hierarchiezustände umfasst nur Zustände serieller Natur: $\Sigma_{cs} \stackrel{\text{def}}{=} \Sigma_{serial}$, d. h. es kann immer nur ein Zustand des Zustandsautomaten aktiv sein.

$$\Sigma_{ss} \cup \Sigma_{cs} = \Sigma, \Sigma_{ss} \cap \Sigma_{cs} = \emptyset.$$

Des Weiteren wird definiert, dass die Menge der Hierarchiezustände Σ_{cs} die Teilmenge der geschlossenen Hierarchie, die mit Σ_{subm} bezeichnet wird, beinhaltet:

$$\Sigma_{subm} \subseteq \Sigma_{cs} \subseteq \Sigma.$$

Damit finden hier Hierarchiezustände eine Beschreibung, deren Unterzustände in einem separaten Zustandsautomaten modelliert werden.

Zur besseren Beschreibbarkeit der Hierarchiezustände und ihrer Unterzustände sollen folgende Definitionen beitragen:

Definition 15 (Unterzustände eines Hierarchiezustandes im Zustandsautomaten) Direkte Unterzustände eines Zustandes $s \in \Sigma$ werden durch eine Funktion $children : \Sigma \rightarrow 2^{\Sigma}$ definiert. Ein Zustand $s \in \Sigma$ wird Basiszustand genannt, falls $children(s) = \emptyset$, sonst heißt er Hierarchiezustand. Für genau einen Zustand $root \in \Sigma$ gilt: $\forall s \in \Sigma, root \notin$

children(*s*). Weiterhin sind der transitive und der reflexive Abschluss von *children* gegeben durch

$$children^*(s) = \bigcup_{i \geq 0} children^i(s) \quad \text{und} \quad children^+(s) = \bigcup_{i \geq 1} children^i(s),$$

$$\text{wobei } children^0(s) = \{s\},\ children^1(s) = children(s) \text{ und für jedes } i \geq 1,$$

$$children^{i+1}(s) = \bigcup_{s' \in children(s)} children^i(s').$$

Die Menge $children^1(s)$ sei mit *direkte Unterzustände* von *s* benannt. Die Funktion *children** kann zu einer Menge von Zuständen erweitert werden, wenn man für eine Menge $\Omega \subseteq \Sigma$ folgendes definiert:

$$children^*(\Omega) = \bigcup_{s \in \Omega} children^*(s).$$

□

(nach [PS91])

Definition 16 (Hierarchiezustand eines Unterzustandes im Zustandsautomaten) Es sei $v \in children(s)$, $s \in \Sigma_{cs}, v \in V$, dann ist *s* Hierarchiezustand von *v*, d. h. $parent(v) = s$.

Als Kurzform für hierarchische Abhängigkeiten soll die in [LP99] vorgestellte Schreibweise benutzt werden. Ein serieller Hierarchiezustand $s_{serial} \in \Sigma_{serial}$, der den direkten Unterzustand $v_{sub} \in V$ hat, wird demnach folgend nur noch mit $s_{serial}(v_{sub})$ bezeichnet werden.

Pseudozustände

Eigenschaften von Zuständen und Transitionen eines Zustandsautomaten können durch Pseudozustände spezifiziert oder verändert werden. Pseudozustände können dabei als funktionale Abbildungen von Zuständen oder Transitionen verstanden werden. Folgend sei die Menge T_{pre} die Transitionen, die zum Pseudozustand hin-, und T_{post} diejenigen, die vom Pseudozustand wegführen. Für die Klärung der Funktionen *source* und *target* sei der Leser auf Definition 19 verwiesen.

Definition 17 (Pseudozustände eines Zustandsautomaten) Es existieren in Zustandsautomaten folgende Pseudozustände aus Π mit den ihnen zugeordneten Abbildungen:

(a) $c_{initial}$ (*Initial-* oder *Startverknüpfung*), mit der Abbildung $c_{initial} : \Sigma_1 \rightarrow \Sigma_{is}$ mit $\Sigma_1 = \Sigma \setminus \Sigma_{is}$,

(b) $c_{history}$ (*History*-Verknüpfung), mit der Abbildung $c_{history} : \Sigma_1 \rightarrow \Sigma_{his}$ mit $\Sigma_1 = \Sigma \setminus \Sigma_{his}$,

(c) $c_{junction}$ (*Junction*-Verknüpfung), mit der Abbildung $c_{junction} : T_{pre} \times T_{post} \rightarrow T_{ct}$ mit $target(t_1 \in T_{pre}) = c_{junction}$, $source(t_2 \in T_{post}) = c_{junction}$, $T_{pre}, T_{post} \subset T$ und $T_{pre} \cap T_{post} = \emptyset$, □

Transitionen

Zustandswechsel finden mit Transitionen statt. Zweckmäßig für die Transformation in Abschnitt 3.3 sollen deshalb Transitionen in Verbindung mit Zuständen nach [LP99] definiert werden:

Definition 18 (Transition eines Zustandsautomaten) Eine Transition aus T ist ein Tripel (v, l_T, v'), wobei die Transition einen Knoten $v \in V$ zusammen mit einer Markierung $l_T \in L_T$ in einen Knoten $v' \in V$ überführt. Mindestens zwei Transitionen können mit einer *Junction*-Verknüpfung zu einer *Compound*- oder *zusammengesetzten Transition* verknüpft werden. Die Menge der zusammengesetzten Transitionen sei mit T_{ct} benannt. Eine Teiltransition einer zusammengesetzten Transition wird auch mit (Transitions-) *Segment* bezeichnet. In der Menge T der Transitionen kann noch die Teilmenge der *stubbed* oder *verstümmelten* Transitionen T_{st} festgestellt werden, die in Hierarchiezuständen mit geschlossener Hierarchie vorkommen. □

Definition 19 (Quelle und Ziel einer Transition im Zustandsautomaten) Es seien $t = (v, l_T, v') \in T$ und $v, v' \in V$, dann kann man eine Funktion $source(t) = v$ definieren, die die Quelle v der Transition findet. Äquivalent dazu sei $target(t) = v'$ definiert, die das Ziel v' liefert. □

Transitionsmarkierungen

Definition 20 (Transitionsmarkierung im Zustandsautomaten) Jede Transition $t \in T$ hat eine Markierung $l_T \in L_T$, die sich aus folgenden Komponenten zusammensetzt:

(a) eine totale Abbildung $trigger : T \to E$ (*Trigger*),

(b) eine partielle Abbildung $condition : T \to L_p$ (Bedingungen),

(c) eine partielle Abbildung $action : T \to L_a$ (Übergangsaktion) und

(d) eine totale Abbildung $priority : T \to \mathbb{N}$ (Priorität).

Für jede Transitionsmarkierung l_T gilt folgende Bildungsregel: Eine Teilmarkierung entsteht durch die Konkatenation:

$$trigger(t)\,[condition(t)]\,/action(t)$$

Die Zeichen ‚[' und ‚]' bzw. ‚/' werden genau dann nicht geschrieben, falls $condition(t) = \varepsilon$ bzw. $action(t) = \varepsilon$. Der andere Teil der Transitionsmarkierung, $priority(t)$, unterliegt keiner Komposition, d. h. er wird unabhängig von anderen Markierungen positioniert. □

Die Menge der *Trigger* besteht allein aus jenen, die bei Veränderung eines Booleschen Wertes von *false* nach *true* bei Eintreten des eingestellten Zeitabstandes ausgelöst werden. Diese sollen mit $E = E_{change}$ bezeichnet werden.

Die Sprache L_p ist ein zusammengesetzter Boolescher Ausdruck und sei über $C_p = C \cup \{\&\&, ||\}$ gebildet, d. h. $L_p \subseteq C_p^*$. Die Menge C sei dabei eine Menge von Primitivbedingungen, aus denen komplexe Bedingungsausdrücke gebildet werden und $\{\&\&, ||\}$ Verknüpfungen zwischen den einzelnen Termen. Die Menge der Primitivbedingungen besteht aus minimalen Booleschen Ausdrücken und beinhaltet Elemente der Menge W. Diese besteht aus Variablen und Methoden, die im Zustandsautomaten definiert werden können, sowie Werten, die den Variablen zugewiesen werden können. Diese Elemente werden in Primitivbedingungen mit Operanden aus $\{==, !=, <>, >=, <=\}$ verknüpft. Für die Sprache L_p gelten die gleichen Bildungsregeln wie für Bedingungsausdrücke in der Programmiersprache *ESDL*[6].

Die Sprache L_a unterliegt ebenso wie L_p den Konstruktionsprinzipien der Programmiersprache *ESDL*. Sie wird über der Menge A, der Menge der Primitivaktionen gebildet ($L_a \subseteq A^*$). Die Menge A besteht aus Elementen aus W, die mit einfachen Operanden aus $\{+, -, *, /, \%\}$ verknüpft und mit einem Semikolon (;) abgeschlossen sind.

Es gelten folgende Einschränkungen für die Markierungen von Transitionen aus T_{ct}: Es sei $t_1 \in T_{pre}$ die zu einem Knoten hinführende Transition und $t_2 \in T_{post}$ die von einem Knoten wegführende Transition, dann kann man für die Attribute von t_1 und t_2 allein zwei mögliche Fälle unterscheiden:

1. $trigger(t_1) \neq \varepsilon,\ condition(t_1) \in L_p,\ action(t_1) = \varepsilon$ *und*
   $trigger(t_2) = \varepsilon,\ condition(t_2) \in L_p,\ action(t_2) \in L_a$

2. $trigger(t_1) = \varepsilon,\ condition(t_1) \in L_p,\ action(t_1) = \varepsilon$ *und*
   $trigger(t_2) \neq \varepsilon,\ condition(t_2) \in L_p,\ action(t_2) \in L_a$

Andere Möglichkeiten der Markierungen solcher Transitionen sind nicht erlaubt.

Zustandsmarkierungen

Definition 21 (Zustandsmarkierung im Zustandsautomaten) Gibt es eine Sprache L_n über einer Menge $CHAR = \{ a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \_ \}$ mit $L_n \subseteq CHAR^* - \{\varepsilon\}$ für die Beschreibung der Zustandsnamen und eine Sprache L_a über die Menge alle (Primitiv-) Aktionen A für die Beschreibung von Aktivitäten und $s \in \Sigma$ ist ein Zustand eines Zustandsautomaten, dann gibt es folgende Funktionen auf s, aus der eine Zustandsmarkierung besteht:

(a) $name : \Sigma \rightarrow L_n$ ist der Zustandsname,

(b) $entry : \Sigma \rightarrow L_a$ ist die Eintrittsaktivität,

[6] An dieser Stelle soll und kann keine vollständige formale Beschreibung der Syntax der Sprache *ESDL* erfolgen, da sie für diese Arbeit nur insofern von Bedeutung ist, dass die Bildung von Ausdrücken entsprechend denen in der Sprache *C* erfolgt. Zur (informellen) Syntax von *ESDL* sei dem Leser das Handbuch [ETA02], S. 196ff. empfohlen.

(c) $exit : \Sigma \rightarrow L_a$ die Austrittsaktivität und

(d) $do : \Sigma \rightarrow L_a$ ist statische Aktivität. □

Die Zustandsmarkierungen werden durch Voranstellen von ‚Entry:' vor die Eintrittsaktivität, von ‚Exit:' vor die Austrittsaktivität und von ‚Static:' vor die statische Aktivität komponiert. Dem Zustandsnamen wird ein ‚/' nachgestellt. Ein *stubbed* Zustand aus Σ_{stub} hat nur den Zustandsnamen als Markierung ohne nachgestelltes ‚/'.

Symbole

Definition 22 (Symbole eines Zustandsautomaten) Jedem Knoten $v \in V$ ist mit der Symbolfunktion s entsprechend seiner Funktion ein eindeutiges Symbol $s(v) \in S$ zugeordnet:

- $s : \Sigma_{ss} \cup \Sigma_{cs} \setminus \Sigma_{subm} \rightarrow$ 
- $s : \Sigma_{subm} \rightarrow$ 
- $s : \Sigma_{stub} \rightarrow \square$
- $s : c_{initial} \mapsto$ 
- $s : c_{history} \mapsto$ 
- $s : c_{junction} \mapsto$  □

Kapitel 3

Transformation von *Statecharts*

Wie der Titel wissen lässt, sollen in dieser Arbeit speziell die Semantiken verschiedener *Statecharts* ineinander überführt werden. Deren visuelle Eigenschaften legen die Benutzung visueller Mechanismen für die Überführung nahe. Auf der Suche nach korrekten Beschreibungsmöglichkeiten der Überführung und motiviert, weiteren möglichen Arbeiten auf diesem Gebiet den Weg zu ebnen, entstand die Idee, Graphgrammatiken für die Definition visueller Sprachen zum Einsatz zu bringen. Graphgrammatiken sind bereits seit längerer Zeit im Rahmen der theoretischen Informatik und Softwareentwicklung untersucht worden. Graphgrammatiken dienen dazu, zum einen neue Strukturen, wie visuelle Sprachen zu erzeugen, zum anderen bestehende Strukturen zu verändern oder zu löschen. Diese Eigenschaft soll in dieser Arbeit genutzt werden. Die Idee dabei ist, ein beschriebenes Modell eines *Statechart* durch Graphtransformation in ein anderes Modell zu überführen. Dadurch wird, ausgehend von einer visuellen Sprache, eine *neue* (ebenfalls visuelle) Sprache konstruiert, die der Zielsprache entspricht. Zusammen mit einem Transformationssystem kann jetzt ein für das Quellsystem entwickeltes *Statechart* in dem Zielsystem mit gleichem Verhalten eingesetzt werden. Durch Vergleich der Semantik der überführten *Statecharts* mit denen des Zielsystems können Rückschlüsse auf die Güte der Transformation gezogen werden. Die Voraussetzungen für den Umgang mit Graphtransformationen sollen dem Leser in den folgenden Abschnitten nahegebracht werden.

3.1 Einführung in Graphtransformation

Mit formalen Graphbegriffen und Algorithmen auf Graphen beschäftigt sich die Wissenschaft schon seit mehreren hundert Jahren. Schon Leonhard Euler benutzte im Jahre 1736 Graphen bei der Lösung des „Königsberger Brückenproblems“ und gilt damit als Pionier. An syntaktischen Formalisierungen des Graphbegriffs besteht heute kein Mangel mehr. Vor ca. 30 Jahren begann dazu die systematische Erforschung dynamischer Aspekte von Graphen in Form von *Graphtransformation* [PR69, EPS73]. Die zugrundeliegende Idee ist dabei die regelbasierte Manipulation statischer Graphen, die sich am Vorbild der Veränderung von Symbolketten durch die Produktionsregeln einer *Chomsky*-Grammatik [Sal78] orientiert. Aufgrund dieser Analogie wird eine Menge von Graphregeln zusammen mit einem Arbeitsgraphen auch *Graphgrammatik* genannt.

So wie Zeichen- oder Baumgrammatiken [GS84, GS97] dazu verwendet werden, den Aufbau von Zeichenketten oder Bäumen zu beschreiben, lässt sich die Struktur komplexer Graphen formal mit Hilfe von Graphgrammatiken definieren. Eine Graphgrammatik ist ein System von Regeln (*Produktionen*), die aus einem Axiom (*Startgraph*) alle Graphen einer bestimmten Klasse erzeugen. Dabei beschreibt eine Produktion der Form (L, R) die Ersetzung einer linken Seite L durch eine rechte Seite R. Sowohl die linke als auch die rechte Seite sind Graphen. Eine Produktion ist genau dann auf einen Graphen anwendbar, wenn die linke Seite in ihm vorkommt.

Die Motivation für die formale Beschreibung von Graphen und Graphtransformationen liegt dabei immer in dem Anliegen, die intuitive und anschauliche Aussagekraft von Graphen mit der maschinellen Interpretierbarkeit zu vereinen. Außerdem schafft eine Formalisierung die Grundlage, bestimmte Eigenschaften der graphisch modellierten Systeme formal zu beweisen. So können Graphgrammatiken etwa auf Unabhängigkeits- und Nebenläufigkeitseigenschaften [Ehr79, Kre87, KW87] oder in Bezug auf Terminierung untersucht werden [Plu95] und es gibt Methoden der statischen und dynamischen Konsistenzanalyse und -sicherung [HW95].

Objekten in visuellen Sprachen sind häufig eine Menge von Namen und Zahlen zugeordnet. Mit diesem Konzept der *Attributierung* [LKW93] kann der Begriff des einfachen Graphen um die Möglichkeit, graphischen Objekten Daten, wie Texte und Zahlen zuzuordnen, erweitert werden. Die Transformation attributierter Graphen ermöglicht dann auch das Operieren auf Attributen, so dass z. B. Berechnungen auf Zahlen durchgeführt werden können, was durch einfache Graphtransformation nicht sinnvoll modelliert werden kann.

Für den Einsatz an einem realen Modell können Graphtransformationen in speziellen Systemen für das Programmieren mit Graphtransformation spezifiziert und zur Anwendung gebracht werden. Mit Systemen wie z. B. *PROGRES* (*PROgrammed Graph REplacement Systems*) [Zün95, SWZ97] oder *PAGG* (*Programm Attributed Graph Grammars*) [Göt83, Göt88, GGN91] existieren umfangreiche Entwicklungsumgebungen für Graphtransformationen. Der große Nutzen von solchen Graphtransformationssystemen kommt aber durch die mangelnde Werkzeugunterstützung der auf dem Markt angebotenen *CASE-Tools* leider äußerst selten zur Anwendung.

Hinsichtlich der Kalküle lassen sich drei Gruppen von Graphgrammatiken unterscheiden (wobei die Grenzen fließend sind), die hier nur in Kürze charakterisiert werden sollen:

Mengentheoretischer Ansatz: In ihm werden Graphen durch Mengen von Knoten und Kanten beschrieben, und die Wirkung von Graphregeln wird mit Hilfe von mengentheoretischen Operationen wie Differenz, Vereinigung etc. beschrieben.

Kategorientheoretischer Ansatz: Im Mittelpunkt der theoretischen Fundierung steht der Begriff der Kategorie, d. h. einer totalen und assoziativen zweistelligen Verknüpfung. Die Wirkung einer Graphregel wird mit Hilfe von kategorientheoretischen Begriffen [Ehr79, Löw93] definiert. Die Kategorientheorie [BW95, Gog89] stellt

eine Begriffswelt zur Verfügung, mit der eine Anzahl verschiedener Strukturen, insbes. auch Graphen, behandelt werden können. Ein großer Vorteil dieses Ansatzes ist seine gute theoretische Fundierung mit einer formalen Semantik, die auf der kategorientheoretischen Konstruktion des *Pushouts* beruht.

Logikorientierter Ansatz: Hier werden sowohl Graphen als auch Graphregeln mit Hilfe von Formelmengen dargestellt, und die Ausführung einer Graphregel wird als Manipulation einer Formelmenge aufgefasst.

Der aufgeführte mengentheoretische Ansatz ist zum einen intuitiv am leichtesten verständlich und gestattet es zum anderen, recht komplexe Graphregeln formal zu beschreiben. Aus diesem Grunde soll für die Transformation von *Statecharts* dieser Ansatz zur Anwendung kommen.

3.2 Theorie der Graph-Grammatiken

Zur Vorstellung des mengentheoretischen Ansatzes sollen in diesem Abschnitt Grundbegriffe nach [Nag79] eingeführt werden, die im Folgenden benötigt werden.

Definition 23 (Gerichtete, markierte Graphen) Ein *gerichteter, markierter Graph* (im Folgenden auch kurz als Graph bezeichnet) über zwei Mengen L_V, L_E ist ein Tripel $G = (V, E, l)$ mit folgenden Komponenten:

(a) V (*vertices*) ist eine (endliche) Menge von *Knoten*,

(b) $E \subseteq V \times L_E \times V$ ist eine Menge von *Kanten* (*edges*), wobei L_E eine (endliche) Menge von *Kantenmarkierungen* (*edge labels*) ist.

(c) $l : V \to L_V$ ist die *Knotenmarkierungsfunktion*; dabei ist L_V eine (endliche) Menge von *Knotenmarkierungen* (*vertex labels*). □

Im Folgenden werden mit $G,\ G',\ G''$ Graphen bezeichnet. Für die Benennung ihrer Komponenten soll dann gelten: $G = (V, E, l),\ G' = (V', E', l'),\ G'' = (V'', E'', l'')$.

Definition 24 (Menge gerichteter, markierter Graphen) Es seien L_V, L_E zwei endliche Mengen (Knoten- bzw. Kantenmarkierungen). Dann wird die *Menge aller markierten Graphen* über L_V und L_E mit $\mathcal{G}(L_V, L_E)$ bezeichnet. □

Definition 25 (Teilgraph) Es seien $G, G' \in \mathcal{G}(L_V, L_E)$. G heißt dann *Teilgraph* von G' (kurz: $G \subset G'$), wenn gilt

(a) $V \subseteq V'$

(b) $E \subseteq E'$

(c) $l = l'|_V$ (d. h. $\forall v \in V : l(v) = l'(v)$) □

Definition 26 (Untergraph) Es sei $G \subset G'$. Dann heißt G *Untergraph* von G' (kurz: $G \subseteq G'$), wenn G alle Kanten aus E' enthält, die Knoten aus V verbinden:

$$E = E'|_V \Leftrightarrow \forall e' \in E' : source(e') \in V \wedge target(e') \in V \Rightarrow e' \in E)$$

Dabei seien $source, target : E \to V$ Funktionen, die Quell- bzw. Zielknoten einer Kante liefern. □

Eine Graphproduktion beschreibt nun analog zu Produktionen für Zeichenketten die Ersetzung eines Teilgraphen durch einen anderen Teilgraphen.

Definition 27 (Graphproduktion) Ein Paar $p = (L, R)$ mit $L, R \in \mathcal{G}$ heißt *Graphersetzungsregel* oder *Graphproduktion*. Man schreibt auch $L \to R$. □

Besteht L aus einem Knoten, heißt die Produktion und damit die Graphgrammatik *kontextfrei*, bei mehrknotigen linken Seiten spricht man von *kontextsensitiven* Produktionen (Graphgrammatiken).

Die Anwendung einer Graphproduktion auf einen s. g. Wirtsgraphen erfolgt nach folgenden Schritten:

1. Der zu ersetzende Teilgraph wird identifiziert.
2. Der zu ersetzende Teilgraph wird gelöscht.
3. Ein neuer Untergraph wird eingefügt.
4. Der eingefügte Untergraph wird in den Wirtsgraphen eingebettet, d. h. es werden Kanten von den neuen Knoten des Untergraphen zu alten Knoten des Wirtsgraphen gezogen.

Definition 28 (Graphgrammatik) $GR = (L_V, L_E, G_S, P, EM)$ heißt *Graphgrammatik*, wenn gilt:

(a) $L_V = N_V \oplus T_V$ ist eine endliche Menge von *nichtterminalen* bzw. *terminalen* Knotenmarkierungen ($L_V = N_V \oplus T_V$ ist die disjunkte Zerlegung der Knotenmarkierungsmenge).

(b) L_E ist eine endliche Menge von (terminalen) *Kantenmarkierungen*.

(c) G_S ist der *Startgraph*, d. h. ein kantenloser Graph, der genau einen Knoten enthält, der mit dem Startsymbol $S \in N_V$ markiert ist.

(d) P ist eine endliche Menge von *Graphproduktionen*.

(e) EM ist eine globale *Einbettungsüberführungsregel*. □

Die Einbettungsüberführung wird anhand von Markierungen und Richtungen von Kanten definiert. Folgende Faktoren können dabei eine Rolle spielen:

- die Markierung des Kontextknotens,
- die Richtung und Markierung einer Kante, die zwischen einem Kontextknoten und dem zu ersetzenden Knoten verläuft,
- die Markierung des zu ersetzenden Knotens,
- die Markierung eines ersetzenden Knotens und
- die Richtung und Markierung einer Kante, die zwischen einem Kontextknoten und einem ersetzenden Knoten erzeugt wird.

Genauere Informationen zur Einbettungsüberführung findet der Leser in [Nag79].

Definition 29 (Ableitung eines Graphen) Es sei *GR* eine Graphgrammatik.

(a) $G \in \mathcal{G}$ *erzeugt direkt* $G' \in \mathcal{G}$ (kurz $G \Longrightarrow_{GR} G'$ oder $G \Longrightarrow G'$), wenn ein Untergraph in G existiert, der mit der linken Seite L einer Produktion $p = (L, R),\ p \in P$, übereinstimmt.

(b) $G \in \mathcal{G}$ *erzeugt* $G' \in \mathcal{G}$ (kurz $G \Longrightarrow^*_{GR} G'$ oder $G \Longrightarrow^* G'$), wenn $G_0, G_1, \ldots, G_k \in \mathcal{G}$ existieren mit $k \in \mathbb{N}_0,\ G_0 = G,\ G_k = G'$ und $G_i \Longrightarrow G_{i+1}$ für $0 \leq i \leq k-1$.

Die Folge $G_0, G_1, \ldots, G_k$ heißt *Ableitung* von G' aus G in GR. □

Definition 30 (Graphsprache) Es sei *GR* eine Graphgrammatik. Dann heißt

$$L(GR) = \{\ G \mid G \in \mathcal{G}(T_V, L_E), G_S \Longrightarrow^* G\ \}$$

die von GR erzeugte Sprache. □

3.3 Eine Grammatik für *Statecharts*

Mit Bezug zum mengentheoretischen Ansatz für Graph-Grammatiken [Nag79] soll hier eine Grammatik für *Statecharts* entworfen werden. Auch die kontextfreie Grammatik in [ER97] wurde dabei als Anregung verwendet. Zuerst wird ein verallgemeinertes *Statechart* entworfen, das als universelle formale Definition von *Statecharts* verstanden werden kann.

Definition 31 (Verallgemeinertes *Statechart*) Ein 5-Tupel $SC = (\Sigma, \Pi, T, s, l)$ über drei Mengen S, L_Σ und L_T heißt *verallgemeinertes Statechart*. Es gilt:

(a) Σ ist eine (endliche) Menge von *Zuständen*,

(b) Π ist eine (endliche) Menge von *Pseudozuständen*,

(c) $T \subseteq V \times L_T \times V$ ist eine Menge von *Transitionen*, wobei L_T eine (endliche) Menge von *Transitionsmarkierungen* ist. Die Menge der *Knoten* V sei die disjunkte Vereinigung der Zustände Σ und Pseudozustände Π (kurz: $V = \Sigma \uplus \Pi$).

(d) $s : V \to S$ ist die *Symbolfunktion*; s ordnet jedem Knoten aus V ein seiner Aufgabe entsprechendes Symbol aus der *Symbolmenge* S zu.

(e) $l : \Sigma \to L_\Sigma$ ist die *Zustandsmarkierungsfunktion*; dabei ist L_Σ eine (endliche) Menge von *Zustandsmarkierungen*. □

Im folgenden werden mit SC, SC', SC'' *Statecharts* bezeichnet. Für die Benennung ihrer Komponenten soll dann gelten: $SC = (\Sigma, T, s, l)$, $SC' = (\Sigma', T', s', l')$, $SC'' = (\Sigma'', T'', s'', l'')$.

Definition 32 (Menge von *Statecharts*) Es seien S, L_Σ, L_T drei endliche Mengen (Symbole, Zustands- bzw. Transitionsmarkierungen). Dann wird die *Menge aller Statecharts* über S, L_Σ und L_T mit $\mathcal{SC}(S, L_\Sigma, L_T)$ bezeichnet. □

Definition 33 (Teil-*Statechart*) Es seien $SC, SC' \in \mathcal{SC}(S, L_\Sigma, L_T)$. SC heißt dann *Teil-Statechart* von SC' (kurz: $SC \subset SC'$), wenn gilt

(a) $\Sigma \subseteq \Sigma'$

(b) $\Pi \subseteq \Pi'$

(c) $T \subseteq T'$

(d) $s = s'|_V \Leftrightarrow \forall v \in V : s(v) = l'(v)$

(e) $l = l'|_\Sigma \Leftrightarrow \forall \sigma \in \Sigma : l(\sigma) = l'(\sigma))$ □

Definition 34 (Unter-*Statechart*) Es sei $SC \subset SC'$. Dann heißt SC *Unter-Statechart* von SC' (kurz: $SC \subseteq SC'$), wenn SC alle Transitionen aus T' enthält, die Knoten aus V verbinden:

$$T = T'|_V \Leftrightarrow \forall t' \in T' : source(t') \in V \wedge target(t') \in V \Rightarrow t' \in T)$$

Dabei seien $source, target : T \to V$ Funktionen, die Quell- bzw. Zielknoten einer Transition liefern. □

Eine *Statechart*-Produktion beschreibt nun analog zu Produktionen für Zeichenketten die Ersetzung eines Teil-*Statecharts* durch ein anderes Teil-*Statechart*.

Definition 35 (*Statechart*-Produktion) Ein Tripel $p = (L, R, EM_l)$ mit $L, R \in \mathcal{SC}(S, L_\Sigma, L_T)$, heißt *Statechart-Ersetzungsregel* oder *Statechart-Produktion*. Man notiert dafür $L \to R$. Der Term EM_l ist die *lokale Einbettungsüberführungsregel*, die einer jeden Graphproduktion zugeordnet ist. □

Besteht L aus einem Knoten, heißt die Produktion und damit die *Statechart*-Grammatik *kontextfrei*, bei mehrknotigen linken Seiten spricht man von *kontextsensitiven* Produktionen (*Statechart*-Grammatiken).

Definition 36 (*Statechart*-Grammatik) $GR = (S, L_\Sigma, L_T, SC_0, P, EM_g)$ heißt *Statechart-Grammatik*, wenn gilt:

(a) $S = N_S \oplus T_S$ ist die disjunkte Zerlegung der Symbolmenge in eine endliche Menge von nichtterminalen bzw. terminalen Symbolen,

(b) $L_\Sigma = N_\Sigma \oplus T_\Sigma$ ist disjunkte Zerlegung der Zustandsmarkierungsmenge in die endliche Menge von nichtterminalen bzw. terminalen Zustandsmarkierungen,

(c) $L_T = N_T \oplus T_T$ ist disjunkte Zerlegung der Transitionsmarkierungsmenge in die endliche Menge von nichtterminalen bzw. terminalen Transitionsmarkierungen,

(d) $SC_0 \in \mathcal{SC}(S, L_\Sigma, L_T)$ ist das Start-*Statechart*, bei dem die Ableitung beginnt,

(e) P ist eine endliche Menge von *Statechart*-Produktionen,

(f) EM_g ist eine *globale Einbettungsüberführungsregel*, die für alle Produktionen einer *Statechart*-Grammatik gilt. □

Definition 37 (Ableitung eines *Statecharts*) Es sei GR eine *Statechart*-Grammatik.

(a) $SC \in \mathcal{SC}(S, L_\Sigma, L_T)$ *erzeugt direkt* $SC' \in \mathcal{SC}(S, L_\Sigma, L_T)$ (kurz $SC \Longrightarrow_{GR} SC'$ oder $SC \Longrightarrow SC'$), wenn ein Unter-*Statechart* in SC existiert, das mit der linken Seite L einer Produktion $p = (L, R, EM_l)$, $p \in P$ übereinstimmt.

(b) $SC \in \mathcal{SC}(S, L_\Sigma, L_T)$ *erzeugt* $SC' \in \mathcal{SC}(S, L_\Sigma, L_T)$ (kurz $SC \Longrightarrow^*_{GR} SC'$ oder $SC \Longrightarrow^* SC'$), wenn $SC_0, SC_1, \ldots, SC_k \in \mathcal{SC}(S, L_\Sigma, L_T)$ existieren mit $k \in \mathbb{N}_0$, $SC_0 = SC$, $SC_k = SC'$ und $SC_i \Longrightarrow SC_{i+1}$ für $0 \leq i \leq k-1$.

Die Folge $SC_0, SC_1, \ldots, SC_k$ heißt *Ableitung* von SC' aus SC in GR. □

Die Anwendung einer *Statechart*-Produktion auf ein so genannten Wirts-*Statechart* erfolgt mit folgenden Schritten:

1. Das zu ersetzende Teil-*Statechart* wird identifiziert.
2. Das zu ersetzende Teil-*Statechart* wird gelöscht.
3. Ein neues Unter-*Statechart* wird eingefügt.
4. Das eingefügte Unter-*Statechart* wird in das Wirts-*Statechart* entsprechend den lokalen und globalen Einbettungsüberführungsregeln mit dem Wirtsgraph verknüpft.

Mit Hilfe einer Einbettungsüberführungsregel wird beschrieben, wie die rechte Seite in den Wirtsgraphen einzubetten ist. Die Einbettungsüberführung wird anhand von Markierungen, Richtungen der Transitionen und hierarchischen Beziehungen definiert. Folgende Faktoren können dabei eine Rolle spielen:

- die Markierung des Kontextzustands,
- die Richtung und Markierung einer Transition, die zwischen einem Kontextknoten und dem zu ersetzenden Knoten verläuft,

- Hierarchiebeziehung des zu ersetzenden Zustands,
- die Markierung des zu ersetzenden Zustands,
- die Markierung des ersetzenden Zustands,
- Hierarchiebeziehung des ersetzenden Zustands und
- die Richtung und Markierung einer Transition, die zwischen einem Kontextknoten und einem ersetzenden Knoten erzeugt wird.

Um Einbettungsüberführungen zu definieren, gibt es laut den Definitionen zwei Möglichkeiten:

Globale Definition: Für die gesamte Graphgrammatik gibt es eine Einbettungsüberführungsregel, die für alle Graphproduktionen gilt.

Lokale Definition: Jeder Graphproduktion ist eine eigene Einbettungsüberführungsregel zugeordnet.

Die lokale Definition ist die mit der höheren Priorität bei der Einbettung und bietet dafür gegenüber der globalen differenziertere Möglichkeiten.

Definition 38 (*Statechart*-Sprache) Es sei *GR* eine *Statechart*-Grammatik. Dann heißt

$$L(GR) = \{\, SC \mid SC \in \mathcal{SC}(T_S, T_\Sigma, T_T),\ SC_0 \Longrightarrow^* SC \,\}$$

die von *GR* erzeugte (*visuelle*) *Sprache*. □

Im bisherigen Verlauf dieses Kapitels wurde der Leser mit einer *Statechart*-Grammatik bekanntgemacht, in der die Reihenfolge der Anwendung von *Statechart*-Produktionen nicht festgelegt ist. In vielen Fällen erweist es sich jedoch als notwendig oder zumindest nützlich, Kontrollstrukturen für *Statechart*-Produktionen einzuführen. Mit Hilfe von Kontrollstrukturen lässt sich beispielsweise ausdrücken, dass zwei Graphregeln nacheinander angewendet werden sollen, dass wahlweise eine von beiden angewendet werden soll, etc. Auf diese Weise entsteht eine gesteuerte *Statechart*-Sprache.

Natürlich kann man die linken und rechten Seiten von Graphregeln immer so erweitern, dass diese in der gewünschten Reihenfolge nacheinander ausgeführt werden. Diese Vorgehensweise ist jedoch aus softwaretechnischer Sicht abzulehnen. Dies gilt insbesondere dann, wenn man einige wenige Basisoperationen in immer neuen Abfolgen zu verschiedenen komplexen Graphtransformationen kombinieren will. In diesem Fall dürfte es zumindest für den Leser einer Spezifikation ohne explizite Kontrollstrukturen unmöglich sein, aus der Betrachtung einer ungeordneten Menge von Graphregeln das Wissen über ihre Ausführungsreihenfolge wiederzugewinnen.

Kontrollstrukturen, wie sie in Graphgrammatiken zum Einsatz kommen, sind meist komplexe graphische Notationen, die Graphersetzung von Erfolg oder Misserfolg einer

vorangegangenen Aktion abhängig machen. In dieser Arbeit soll jedoch die Steuerung der Ableitungen durch das aus [Sal78] bekannte Konzept der gesteuerten Sprache zur Anwendung kommen. Bei gesteuerten Sprachen werden die zugehörigen Ableitungen durch Sprachen über dem Alphabet von Marken der Produktionen einer Grammatik gesteuert.

Definition 39 (Steuersprache einer *Statechart*-Grammatik) Es sei $GR = (S, L_\Sigma, L_T, SC_0, P, EM_g)$ eine *Statechart*-Grammatik, $Lab(P)$ eine Menge von Marken für P und D eine Ableitung in GR. Dann heißt $c \in (Lab(P))^*$ *Steuerwort* von D, wenn

(a) $SC, SC' \in \mathcal{SC}(S, L_\Sigma, L_T)$ existieren, so dass D die Ableitung $SC \Longrightarrow_p SC'$ ist und die Produktion $p \in P$ mit $c \in Lab(P)$ markiert ist und

(b) die *Statecharts* SC, SC', SC'' und die Worte c_1, c_2 existieren, so dass D die Ableitung $SC \Longrightarrow^* SC' \Longrightarrow^* SC''$ ist und $c = c_1c_2$ gilt, wobei c_1 ein Steuerwort von $SC \Longrightarrow^* SC'$ und c_2 ein Steuerwort von $SC' \Longrightarrow^* SC''$ ist.

Für $C \subset (Lab(P))^*$ heißt

$$L(GR, C) = \{SC \in \mathcal{SC}(T_S, T_\Sigma, T_T) \mid D : SC_0 \Longrightarrow^* SC, c \in C \text{ ist Steuerwort von } D\}$$

die von GR mit *Steuersprache* C erzeugte Sprache. $L(GR)$ ist dann die durch C *gesteuerte Statechart-Sprache*. □

Damit ist ein universeller Ableitungsformalismus für visuelle Sprachen entwickelt, der die intuitive Kraft desjenigen für Zeichen um visuelle Komponenten erweitert und dabei die Anschaulichkeit mengentheoretischer Kalküle beibehält. Da in dieser Arbeit nicht der generative Aspekt einer Grammatik im Mittelpunkt stehen soll, sondern die anstehende Transformation von *Statecharts*, erfolgt eine Beschränkung auf kontextsensitive Grammatiken. Dabei wird als linke Seite einer Produktion ein Teilgraph angenommen, der aus mindestens einer nichtterminalen Komponente besteht. Als Kontext werden dann die Komponenten der linken Seite L einer Produktion bezeichnet, die in der rechten Seite R unverändert auftauchen.

Kapitel 4

Eine Modelltransformation

Basierend auf einer verallgemeinerten Definition von *Statecharts* wurde im Abschnitt 3.3 eine allgemeine Grammatik für *Statecharts* konstruiert. Darauf aufbauend soll diese nunmehr als ein Transformationssystem für eine exemplarische und konkrete Übersetzung von Zustandsmodellen in *Artisan RtS* und *Ascet-SD* adaptiert werden. Da die Grammatik als Werkzeug für Übersetzungen von *Statecharts* entworfen wurde, soll zunächst geklärt werden, ob damit auch die Zustandsautomaten in *Ascet-SD* behandelt werden können, bevor Strukturelemente von *Statecharts* auf ihre Übersetzbarkeit hin überprüft werden und schließlich die Einschränkungen bzgl. der Übersetzung formal festgehalten werden. Danach kann schließlich das Transformationssystem mit den Regeln für die Ableitung von *Statecharts* aus *Artisan RtS* konstruiert werden.

4.1 Vorbetrachtungen für die Modelltransformation

4.1.1 Vergleich von *Statecharts* mit hierarchischen Zustandsautomaten

Wie bereits in Kapitel 2 erwähnt, ist ein *Statechart* als Erweiterung von Zustandsautomaten durch hierarchische Zustände, Orthogonalität und *Broadcasting* definiert. Darin ist auch der auffälligste Unterschied beider Modelle begründet: während *Statecharts* von *Artisan RtS* strikt nach den Vorgaben für *Statecharts* entworfen wurden und die genannten Eigenschaften aufweisen, fehlen den Zustandsautomaten in *Ascet-SD* jedoch die Merkmale der Orthogonalität und des *Broadcastings*. Der Name ‚hierarchische Zustandsautomaten' [ETA02] ist also durchaus richtig gewählt. Da die fehlenden Merkmale die Zustandsautomaten zu einer Teilmenge von *Statecharts* gradieren, liegt die Vermutung nahe, dass es einen Morphismus gibt, der *Statecharts*, evtl. mit Ausschluss bestimmter Komponenten, auf Zustandsautomaten (oder umgekehrt) abbildet. Möglich scheint das durch Anwendung des in Abschnitt 3.3 entwickelten Formalismus. Grundlage für dessen Anwendung ist die einheitliche Darstellbarkeit beider Modelle nach Definition 31. Dass dies möglich ist, wurde bereits in den Abschnitten 2.2.3 und 2.3.3 bewiesen. Es bleibt also noch zu klären, welche Komponenten für die Überführung in Frage kommen. Dabei kann zwischen der Abbildung auf äquivalente Komponenten und auf Kombinationen von

Komponenten, die in ihrer Kombination das gleiche Verhalten nachahmen, unterschieden werden. Genau das soll für die Überführung der *Statecharts* in die Zustandsautomaten für die Mengen visueller Komponenten, die in den jeweiligen Definitionen beschrieben wurden, im folgenden Abschnitt geprüft werden.

4.1.2 Strukturelemente von *Statecharts* für die Modelltransformation

Folgend sollen nun die Strukturelemente von *Statecharts* aus *Artisan RtS* auf ihre Übersetzbarkeit in Zustandsautomaten aus *Ascet-SD* geprüft werden. Dabei wird großer Wert sowohl auf Umsetzung möglichst aller Elemente, als auch auf die semantisch richtige Umsetzung dieser Komponenten gelegt. Existieren keine Elemente mit gleichem Verhalten, werden Vorschläge zum Nachahmen des Verhaltens von *Statecharts* mit Zustandsautomaten unterbreitet.

Zustände

Basiszustand: Basiszustände existieren sowohl im *Statechart* als auch im Zustandsautomaten und bedürfen keiner expliziten Übersetzung.

serieller Hierarchiezustand: Diese Teilmenge der Zustände ist zwar in beiden Modellen vorhanden, jedoch müssen jeweils verschiedene Verhaltensweisen bei der Auswertung von Transitionen ausgehend von verschiedenen Hierarchiestufen festgestellt werden. Bei *Statecharts* hat die Transition die höhere Priorität, die von der untersten Hierarchiestufe ausgeht. Bei Zustandsautomaten ist dies diejenige, die von der höchsten ausgeht. Dieses Verhalten kann durch *Flattening* mit neuer Prioritätenbewertung der Transitionen in Zustandsautomaten nachgeahmt werden.

geschlossene Hierarchie: Zustände mit dieser Eigenschaft müssen vor der Übersetzung in herkömmliche Hierarchiezustände überführt werden. Dazu wird einfach der *submachine*-Zustand mit dem Unter-*Statechart*, für das er steht, ersetzt. Diese Prozedur wird dadurch vereinfacht, dass *Statecharts* in *Artisan RtS* keine Transitionen vom (Haupt-) *Statechart* in das Unter-*Statechart* besitzen können.

orthogonaler Hierarchiezustand: Solche Zustände sind in Zustandsautomaten nicht bekannt. Das Zusammenspiel der Unterzustände dieser Zustände kann aber durch Clusterung der jeweiligen Zustände, die gleichzeitig aktiv sein können, nachgebildet werden (Sequentialisierung).

Aktivität: Da beim *Flattening* durchaus die Ein- oder Austrittsaktivität in die Aktionen benachbarter Transitionen verschoben werden müssen, soll eine Beschränkung auf Aktivitäten, die keine Zeit benötigen (z. B. Zuweisungen, einfache Operationen), erfolgen. Dies ist deshalb notwendig, da in Zustandsautomaten die Ausführung einer

Transition ohne Zeitverlust erfolgt, eine Aktion, die Zeit fordert, jedoch abgebrochen und nicht ausgeführt wird. Für statische Aktivitäten sind keine Einschränkungen zu bemerken. Bei der Überführung von *Statechart* wird davon ausgegangen, dass die Ausdrücke in *Statecharts* und Zustandsautomaten der gleichen Syntax, nämlich der der Programmiersprache *C*, unterliegen.

interne Transition: Solche Zustandsmarkierungen sind in Zustandsautomaten nicht bekannt. Eine Möglichkeit der Umsetzung besteht in der Ausführung der zugeordneten Aktion in der statischen Aktivität des Zustandes. Die Transition kann dafür auch bedingt sein. Allerdings kann eine *Trigger*-Markierung nicht umgesetzt werden, da es sowohl beim *Flattening* als auch bei der Sequentialisierung zur Verknüpfung mehrerer *Trigger* kommen würde. Solche Operationen sind in Zustandsautomaten nicht möglich.

Endzustand: Ein Endzustand kann in Zustandsautomaten nicht definiert werden, er kann aber leicht durch einen Basiszustand ersetzt werden.

Synchronisationszustand: Das spezielle Verhalten von orthogonalen Zuständen, welches durch einen Synchronisationszustand hervorgerufen wird, kann bei der Sequentialisierung berücksichtigt werden.

Zustandsname: In Zustandsautomaten werden Zustände als eindeutige Identifikatoren benutzt; mehrere Zustände mit gleichem Namen sind deshalb dort nicht erlaubt. In *Statecharts* können mehrere Zustände gleiche Bezeichner haben. Aus diesem Grunde soll eine Beschränkung der Zustände auf solche mit unterschiedlichen Bezeichnern erfolgen.

Pseudozustände

Startverknüpfung: Mit diesem Pseudozustand legt man in beiden Modellen den Zustand fest, der als erster aktiv sein soll.

Deephistory-Verknüpfung: Eine *Deephistory*-Verknüpfung steht dem Modellierer in Zustandsautomaten nicht zur Verfügung. Das Verhalten, das dieser Pseudozustand im *Statechart* hervorruft, kann allerdings leicht durch Hinzufügen einer Geschichte zu den Unterhierarchiezuständen nachgeahmt werden.

Shallowhistory-Verknüpfung: Dieser Pseudozustand existiert in beiden Modellen und bedarf nicht der expliziten Übersetzung.

Join-/Fork-Verknüpfung: Beide Pseudozustände sind feste Bestandteile von orthogonalen Konstruktionen und können gut bei der Sequentialisierung durch Ersetzung der Segmente mit einfachen Transitionen transformiert werden. Dabei müssen allerdings die Aktionen und Bedingungen logisch verknüpft werden.

Choice-Verknüpfung: Da die *Choice*-Verknüpfung lediglich Transitionen verzweigt, ist es möglich, sie als einfache Transitionen zu übersetzen, wobei die optionalen Bedingungen an den Transitionen logisch kombiniert werden. Da die Bedingungen und Aktionen der ausgehenden Transitionssegmente nicht auf die Aktionen der eingehenden reagieren, sollen nur die ausgehenden Transitionssegemente mit Aktionen beschriftet werden. Alternativ zur Übersetzung mit Transitionen könnte auch die *Junction*-Verknüpfung der Zustandsautomaten benutzt werden.

Transitionen

Trigger: In Zustandsautomaten sind nur *Timeout-Events* bekannt. Sie veranlassen eine Transition bei erfüllter Bedingung immer dann zum Zustandsübergang, wenn eine bestimmte Zeit *dT* eines internen *Timers* verstrichen ist. Sobald *dT* erreicht ist, startet der *Timer* erneut. Die Zeitdifferenz *dT* wird in der *Runtime*-Umgebung eingestellt, die graphischen Implementierungen eines Zustandsautomaten haben auf diesen Parameter also keinen Einfluss.

Die Trigger von *Statecharts* können ob ihrer unterschiedlichen Generierung und Funktionsweise in Zustandsautomaten nicht simuliert werden. Da möglichst viele Komponenten von *Statechart* überführt werden sollen, soll hier ein Konzept erarbeitet werden, wie *Trigger*, die in Zustandsautomaten nicht existieren, dort trotzdem benutzt werden können. Dabei sind gewisse Einschränkungen anzumerken, die im Folgenden für die *Trigger* in *Statecharts* herausgearbeitet werden:

Signal-Event: Ein *Signal-Event* im *Statechart* ist ein von außerhalb des *Statecharts* erzeugtes Signal. Dieses Verhalten kann in Zustandsautomaten nur vom Benutzer selbst simuliert werden. Entweder konstruiert er sich dazu eine signalgebende Komponente außerhalb des Zustandsautomat oder er benutzt die Möglichkeit, in der *Runtime*-Umgebung eine Variable zu stimulieren. Auf diese Weise lässt sich z. B. ein Boolescher Wert in der Bedingung auf *true* setzen, was Voraussetzung für das Schalten einer Transition ist.

Call-Event: Im *Statechart* kann infolge eines *Signal-Events* als *Trigger* einer Transition ein Methodenaufruf in der Aktion erfolgen. Dieser Methodenaufruf löst ein dazugehöriges *Call-Event* aus, das als *Trigger* eine andere Transition auslösen kann. Will man nun ein *Call-Event* in Zustandsautomaten benutzen, legt man für den eben genannten Fall für *Signal-* und *Call-Event* (Methodenaufruf) eine Boolesche Variable (z. B. var_{signal} und var_{call}) an. Die Variable var_{signal} wird dann mit der Bedingung der zu übersetzenden Transition *UND*-verknüpft und die Variable var_{call} in der Aktion auf *true* gesetzt. Nun sorgt die Variable var_{call} in der Bedingung einer anderen Transition dafür, dass die Bedingung für den Schaltvorgang erfüllt ist.

Change-Event: Das *Change-Event* wird auf den Namen eines *Timeout-Events* abgebildet, indem eine Konkatenation ‚*when_parameter*‘ gebildet wird, wobei

‚*when*' der Index für die ursprüngliche Art des *Triggers* und ‚*parameter*' der Parameter (die Bedingung für den *Trigger*) des *Change-Events* im *Statechart* ist. Dabei muss bemerkt werden, dass die Eigenschaft eines *Change-Events* im *Statechart*, bei noch laufender Aktivität keine Zustandsüberführung, sondern nur den Abbruch der Aktivität hervorzurufen, in Zustandsautomaten nicht zum Tragen kommt. Die Beschränkung aller Aktivitäten (also auch statischer) auf solche, die keine Zeit zur Ausführung benötigen, schafft Abhilfe.

Time-Event: Das *Time-Event* wird auf den Namen eines *Timeout-Events* abgebildet, indem eine Konkatenation ‚*after_parameter*' gebildet wird, wobei ‚*after*' der Index für die ursprüngliche Art des *Triggers* und ‚*parameter*' die Zeitangabe für den *Trigger* des *Time-Events* im *Statechart* ist.

Completion-Event: Das *Completion-Event* wird als Zeichenkette ‚*completion*' auf den Namen eines *Timeout-Events* abgebildet.

Wichtig für die Simulation der so neu erzeugten *Timeout-Trigger* ist die Einstellung in der *Runtime*-Umgebung auf denselben *Timeout*-Wert dT, damit die *Trigger* kooperieren.

Bedingung und Übergangsaktion: Transitionen in *Statecharts* als auch in Zustandsautomaten können mit Bedingungen und Übergangsaktionen beschriftet sein. Eine Übersetzung wird deshalb dafür nicht notwendig. Es wird davon ausgegangen, dass Bedingungen bzw. Aktionen beider Modelle die gleiche Syntax besitzen, nämlich die der Programmiersprache *C*.

Priorität: Bei der Überführung werden Transitionen Prioritäten zugeordnet, da diese für Transitionen in Zustandsautomaten zwingend sind. Bei der Vergabe von Prioritäten fließen zum einen die durch die *UML* festgelegten Informationen über Transitionen verschiedener Hierarchien ein und zum anderen die impliziten, durch die Analyse festgestellten Prioritäten, wie z. B. von Transition in verschiedenen Regionen.

Segmente von zusammengesetzten Transitionen:

- Das Segment ausgehend von einer Startverknüpfung kann in *Statecharts* mit einer Aktion beschriftet sein. Da die Startverknüpfung in Zustandsautomaten ohne Transition erfolgt, muss die Aktion bei der Übersetzung einen anderen Platz finden und die Transition gelöscht werden.
- Segmente, die von der *Deep-/Shallowhistory*-Verknüpfung in *Statecharts* abgehen bzw. in sie hineinzeigen, sind in Zustandsautomaten nicht erlaubt. Solchen Transitionen müssen andere Quellen/Ziele zugeordnet werden. Besonderes Augenmerk ist auf Verknüpfungen zu legen, welche die Startverknüpfung ersetzen. Transitionen, die diese Eigenschaft gewährleisten, können dann umgeleitet werden und die Startverknüpfung muss explizit eingeführt werden.

- Segmente von oder zu *Fork*-, *Join*- und *Choice*-Verknüpfungen können gut durch einfache Transitionen ersetzt werden, wobei jede Transition genau eine *Trigger*-Beschriftung erhält sowie Bedingungen logisch kombiniert und Aktionen in Reihenfolge konkateniert werden.

Variablen

Da Zustandsautomaten nur arithmetische und Boolesche Variablen anbieten, wird die Menge der benutzbaren Variablen in *Statecharts* wie in Taf. 4.1 beschränkt. Dabei müssen die Werte des Typs *octal* eine Konvertierung erfahren, damit sie einem Typ mit dezimaler Zahlenbasis zugewiesen werden können. Da Informationen über die Implementierungstypen für Zustandsautomaten (siehe Abschnitt 2.3.1) nicht im Zustandsautomat enthalten sind, ist der Modellierer gefordert, entsprechende Typen für die Modelltypen vor der Code-Erzeugung zu wählen. Ansonsten werden die Standardimplementierungen der Taf. 2.4 vom System voreingestellt.

Zusammenfassend werden die Modelle im Anhang A übersichtlich in einer Tabelle gegenübergestellt. Sie gibt Auskunft über enthaltene Komponenten und darüber, welche von ihnen in einen Zustandsautomaten übersetzt werden können.

Tafel 4.1: Abbildung der Datentypen von Statecharts auf Datentypen von Zustandsautomaten

Datentyp im *Statechart*	Datentyp im Zustandsautomat
any	–
boolean	log
char	–
double	cont
float	cont
long	sdisc
octal	sdisc[a]
short	sdisc
unsigned long	udisc
unsigned short	udisc

[a] Zahlenbasis-Konvertierung

4.1.3 Eingeschränktes *Statechart*

Nachdem nun im vorigen Abschnitt die Strukturelemente der *Statecharts* auf ihre Übersetzbarkeit überprüft wurden und Möglichkeiten der Übersetzung aufgezeigt wurden, kann nun darauf bezugnehmend das eingeschränkte *Statechart* formal beschrieben werden. Ein eingeschränktes *Statechart* ist dabei nichts anderes als ein *Statechart* in *Artisan RtS*, wie es in Abschnitt 2.2.3 definiert wurde, vermindert um die in Abschnitt 4.1.2 diskutierten Strukturelemente. Werden also im Folgenden Eigenschaften und Komponenten nicht aufgeführt, dann gilt Gleiches wie in Abschnitt 2.2.3.

Definition 40 (Eingeschränktes *Statechart*) Es sei $SC_{Artisan\,RtS}$ ein *Statechart* von *Artisan RtS* nach Definition 1. Ein Tupel $SC'_{Artisan\,RtS} = (\Sigma', \Pi', T', s', l')$ über drei Mengen S', L'_Σ, L'_T heißt dann *eingeschränktes Statechart* für die Übersetzung in einen Zustandsautomaten in *Ascet-SD*. Es gelten dabei folgende Einschränkungen für die Mengen eingeschränkter *Statecharts* gegenüber denen in *Artisan RtS*:

1. $S' = S$,
2. $L'_\Sigma = L_\Sigma$ mit der Einschränkung, dass die Sprache der Bedingungen L_p und die Sprache der Aktivitäten L_a nicht über die Menge der Methoden gebildet werden,
3. $L'_T = L_T$ mit der Einschränkung, dass
   a) die Sprache der Bedingungen L_p und die Sprache der Aktivitäten L_a nicht über die Menge der Methoden gebildet werden,
   b) $action(t \in T_{pre}) \rightarrow \varepsilon$ mit $T_{pre} \subset T$ und $target(t \in T_{pre}) = c_{choice}$,
   c) $priority(t \in T) \rightarrow \mathbb{N} \cup \varepsilon$,
4. $\Sigma' = \Sigma$ mit der Einschränkung, dass die Hierarchiezustände der *submachine*-Zustände in einfache Hierarchiezustände umgewandelt wurden,
5. $\Pi' = \Pi$,
6. $T' = T$,
7. $l' = l$ mit der Einschränkung, dass
   a) $name(s) \neq name(s')$ für $s \neq s'$, mit $s, s' \in \Sigma$,
   b) $trigger(internalTransition(s \in \Sigma)) \rightarrow \varepsilon$ und
8. $s' = s$ □

Die Syntax der eingeschränkten *Statecharts* SC' ist also über einer Teilmenge der Syntaxelemente von *Statecharts* in *Artisan RtS* gebildet; die Semantik wird damit gegenüber einem *Statechart* in *Artisan RtS* nicht verändert.

4.2 Transformationssystem

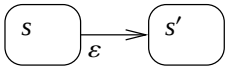
Mit den Vorbereitungen der Definition von *Statecharts* in *Artisan RtS* in Abschnitt 2.2.1 und von Zustandsautomaten in *Ascet-SD* in Abschnitt 2.3.1 sowie einem Konstrukt zur Übersetzung in Abschnitt 3.3 und der Beschränkung auf übersetzbare Komponenten von *Statecharts* in *Artisan RtS* im vorigen Abschnitt, sind nun die Voraussetzungen gegeben, ein speziell auf die Abbildung von *Statecharts* auf Zustandsautomaten zugeschnittenes Transformationssystem zu konstruieren.

Definition 41 (Transformationssystem) Es sei $SC_{Artisan\,RtS} \in \mathcal{SC}(S, L_\Sigma, L_T)$ ein eingeschränktes *Statechart* in *Artisan RtS* nach Definition 40 und $ZA_{Ascet\text{-}SD} \in \mathcal{SC}(S', L'_\Sigma, L'_T)$ ein Zustandsautomat in *Ascet-SD*, der wie ein *Statechart* definiert ist. Eine *Statechart*-Grammatik $G^T = (S^T, L^T_\Sigma, L^T_T, SC_0, P, EM_g)$ heißt *Transformationssystem* für die Abbildung von $SC_{Artisan\,RtS}$ auf $ZA_{Ascet\text{-}SD}$, wenn gilt:

(a) $S^T = S \cup S'$ ist die Symbolmenge, die die disjunkten Mengen der nichtterminalen Symbole $N_S = S \setminus S'$ und die der terminalen Symbole $T_S = S'$ vereinigt.

(b) $L^T_\Sigma = L_\Sigma \cup L'_\Sigma$ ist die Zustandsmarkierungsmenge, die die disjunkten Mengen der nichtterminalen Zustandsmarkierungen $N_\Sigma = L_\Sigma \setminus L'_\Sigma$ und die der terminalen Zustandsmarkierungen $T_\Sigma = L'_\Sigma$ vereinigt.

(c) $L^T_T = L_T \cup L'_T$ ist die Transitionsmarkierungsmenge, die die disjunkten Mengen der nichtterminalen Transitionsmarkierungen $N_T = L_T \setminus L'_T$ und die der terminalen Zustandsmarkierungen $T_T = L'_T$ vereinigt.

(d) SC_0 ist das Start-*Statechart*; bei ihm beginnt die Ableitung. $SC_0 =$ 

(e) $P = \{T_1, \ldots, T_{49}, F_1, \ldots, F_{18}\, S_1, S_2\}$ ist die Menge von Produktionen oder auch *Transformationsregeln*.

(f) EM_g, die globale Einbettungsüberführungsregel, wird hier nicht gebraucht, deshalb ist $EM_g = \emptyset$. □

Um sowohl Merkmale von *Statecharts* als auch von Zustandsautomaten darstellen zu können, erfolgt die Notation der Regeln nach den Vorgaben in Definition 41. Die notierten Regeln können in Kategorien bzgl. der Syntax und Semantik eingeteilt werden:

1. Syntax und Semantik von Komponenten von *Statecharts* und Zustandsautomaten sind identisch und bedürfen keiner Übersetzung (z. B. Basiszustände).

2. Syntax von Komponenten von *Statecharts* und Zustandsautomaten sind identisch, jedoch muss die Semantik wegen ihrer unterschiedlichen Verhaltenweisen angepasst werden (z. B. Auswertung der Transitionen unterschiedlicher Hierarchiestufen).

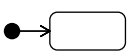
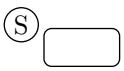
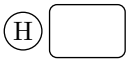
3. Komponenten von *Statecharts* und Zustandsautomaten haben unterschiedliche syntaktische Ausprägungen aber gleiche Semantik (z. B. Startzustand).

4. Es existieren in *Statecharts* Komponenten, für die es in Zustandsautomaten keine Entsprechung gibt (z. B. orthogonale Zustände). Diese Komponenten werden mit den Transformationsregeln P des Transformationssystems G^T so umgeformt, dass sie mit den sprachlichen Mitteln von Zustandsautomaten nachgeahmt werden.

Auf die sprachlichen Mittel, die in die Kategorie unter Punkt 3 fallen, wird in Taf. 4.2 eingegangen; es werden semantisch äquivalente Komponenten mit unterschiedlicher syntaktischer Ausprägung der *Statecharts* von *Artisan RtS* und der Zustandsautomaten von *Ascet-SD* gegenübergestellt. Da sie lediglich Repräsentanten ein und derselben Verhaltenweisen darstellen, sind sie für Übersetzungen von Modellen von rein formalem Interesse. Sie kommen genau dann nicht zum Einsatz, wenn reale Modelle übersetzt werden sollen, da die Syntax für die Strukturelemente von den Modellimplementierungen vorgegeben werden und nicht frei wählbar sind.

Den Transformationsregeln, die der Kategorie 4 entsprechen, ist der nächste Abschnitt gewidmet. Dabei wird der Übersichtlichkeit halber ‚T n' statt ‚T_n' für Transformationsregeln, ‚F n' statt ‚F_n' für *Flattening*-Regeln und ‚S n' statt ‚S_n' für syntaktische Transformationsregeln für die Produktionen aus P notiert. *Flattening*-Regeln sind spezielle Transformationsregeln, die der Einebnung der seriellen Hierarchiezustände dienen. Die syntaktischen Regeln dienen lediglich der syntaktischen Transformation von Strukturelementen mit gleicher Semantik (s. Kategorie 3).

Tafel 4.2: Strukturelemente gleicher Semantik und unterschiedlicher syntaktischer Ausprägung

Strukturelement	*Statecharts*	Zustandsautomaten
Startzustand		
Zustand mit Geschichte		
submachine-Zustand		
Beginn der Eintrittsaktivität	‚Entry/'	‚Entry:'
Beginn der Austrittsaktivität	‚Exit/'	‚Exit:'
Beginn der statischen Aktivität	‚do:'	‚Static:'

4.3 Transformationsregeln

Die Regeln, die in diesem Abschnitt aufgeführt werden, komplettieren das Transformationssystem des letzten Abschnitts. Eine Regel besteht aus einem graphischen und einem Textteil. Der linke Teil der Graphik ist L und der rechte Teil ist R in einer Produktion $p = (L, R, EM_l) \in P$. Dem graphischen Teil folgt die genaue textliche Beschreibung von L und R, wobei L und R aus ein oder mehreren Termen bestehen und der Notation von [LP99] folgen. Beide sind durch ein ‚$\Longrightarrow$' getrennt. Daran schließt sich die lokale Einbettungsüberführungsregel EM_l einer Produktion an. Werden Komponenten der Teil-*Statecharts* nicht erwähnt, sind sie für die jeweilige Produktion unwesentlich und bleiben dem abzuleitenden *Statechart* erhalten. Wird eine graphische Komponente durch die rechte Seite einer Regel entfernt, so werden mit ihr alle Komponenten entfernt die nicht ohne die entfernte existieren können (z. B. Markierungen).

Der Textteil verlangt nach einer formalen Beschreibungssprache für *Statecharts*. Die Schreibweise, wie sie in [LP99] benutzt wird, ist für die Beschreibung der *Statechart*-Grammatik sehr dienlich und soll zusammen mit den Erweiterungen, wie sie in Abschnitt 2.2.3 notiert wurden, in dieser Arbeit zum Einsatz kommen. Ein serieller Hierarchiezustand $s_{serial} \in \Sigma_{serial}$, der den direkten Unterzustand $v_{sub} \in V$ hat, wird demnach im Folgenden nur noch mit $s_{serial}(v_{sub})$ bezeichnet werden. Darüberhinaus soll Folgendes abgekürzt werden: Hat ein orthogonaler Hierarchiezustand $s_{orthogonal} \in \Sigma_{orthogonal}$, in der Region $r \in \mathcal{R}(s_{orthogonal})$, einen direkten Unterzustand $v_{sub} \in V$, dann soll nur noch $s_{orthogonal}[r(v_{sub})]$ geschrieben werden. Desweiteren soll folgende Notation für eine Transition $t = (v, l_T, v')$, mit $t \in T, v, v' \in V$, in dieser Arbeit Verwendung finden:

$$t : v \xrightarrow[p]{e[c]/a} v',$$

wobei $e = trigger(t) \in E, c = condition(t) \in L_p, a = action(t) \in L_a$ und $p = priority(t) \in \mathbb{N}$. Mengen von Komponenten werden in geschweifte Klammern eingeschlossen.

Zur Beschreibung der Transformationsregeln sollen weitere Definitionen verwendet werden:

Definition 42 (incoming, outgoing) Die Menge der einkommenden Transitionen in einem Knoten $s \in V$ sei mit $incoming(v) \stackrel{\text{def}}{=} \{t = (v', l_t, v) \mid source(t) = v' \wedge target(t) = v)\}$ definiert. Auf einer Menge von Zuständen $\Omega \subseteq V$ sei die Menge der einkommenden Transitionen definiert mit:

$$incoming^*(\Omega) \stackrel{\text{def}}{=} \bigcup_{s \in \Omega} incoming(s).$$

Äquivalent dazu kann $outgoing(v)$ als die Menge der ausgehenden Transition aus einem Knoten und $outgoing^*(\Omega)$ als die Menge der ausgehenden Transitionen über einer Menge von Knoten definiert werden. □

Definition 43 (Maximale Priorität) Es seien $\{p_i = priority(t_i)\}$ die Prioritätsmarkierungen einer Schar von Transitionen $\{t_i \in T\}$ ausgehend von einem Knoten $v \in V$. Die *maximale Priorität* $\bar{p}$ über alle Prioritäten $\{p_i\}$ von $\{t_i\}$, die alle von einem Knoten v ausgehen, sei dann wie folgt definiert:

$$\bar{p}(t = (v, l_T, v')) \stackrel{\text{def}}{=} \max_{i=1...|outgoing(v)|} \{p(v, l_{T_i}, v'_i) \mid v, v'_i \in V\}$$

Die Funktion $\bar{p}$ kann auch für eine Menge von Transitionen definiert werden, wenn man für eine Menge von Transitionen $T_1 = \bigcup_{k>1}(v_m, l_{T_k}, v_n), T_1 \subset T,\ m, n \leq k$ Folgendes definiert:

$$\bar{p}(T_1) \stackrel{\text{def}}{=} \max_{k>0}\{\bar{p}(t_k = (v, l_{T_k}, v')) \mid t_k \in T_1\}$$

□

Definition 44 (Färbung eines Zustandes) Für einen Zustand eines *Statecharts* $s \in \Omega$ mit $\Omega \subseteq \Sigma$, sei eine Abbildung $\phi(s)$ gegeben, wobei ϕ die *Färbungsfunktion* oder *Färbung* eines Zustandes ist. Die Vereinigung aller mit ϕ gefärbten Zustände sei $\Phi(\Omega) \stackrel{\text{def}}{=} \bigcup_i \phi(s_i)$. Färbungen erweitern die nichtterminalen Symbole:

$$N_\Sigma = N_\Sigma \cup \Phi(\Omega),\ N_S = N_S \cup s(\Phi(\Omega)),\ s : \Phi(\Omega) \rightarrow \boxed{}.$$

□

Definition 45 (Systeme aus gleichzeitig aktiven Unterzuständen) Ein System, zusammengesetzt aus gleichzeitig aktiven Unterzuständen $s_i \in children(s_{orthogonal})$ eines orthogonalen Zustandes $s_{orthogonal} \in \Sigma_{orthogonal}$, soll mit $\xi(\mathcal{R})$ bezeichnet werden. Das System besteht aus genau einem Zustand aus je einer der Regionen $r_i \in \mathcal{R}(s_{orthogonal})$, die $s_{orthogonal}$ aufspannen. $\xi(\mathcal{R})$ verknüpft die gleichzeitig aktiven Zustände durch die Operation $\times$. Damit kann ξ wie folgt definiert werden:

$$\xi(\mathcal{R}) \stackrel{\text{def}}{=} s_1 \times \ldots \times s_n = \bigtimes s_i,$$

für $s_i \in r_i,\ r_i \in \mathcal{R}(s_{orthogonal}),\ i \in \{1, \ldots, n\},\ n = |\mathcal{R}(s_{orthogonal})|$. Die Menge aller solcher Systeme sei mit

$$\Xi \stackrel{\text{def}}{=} \bigcup_j \xi_j(\mathcal{R})$$

benannt. Dabei sei ξ_j eine der möglichen Kombinationen der Zustände in den Regionen mit $\xi_j \neq \xi_{j+1}$. □

4.3.1 Grundlegende Regeln

Zu Beginn der Ableitung werden Regeln eingeführt, die sowohl *Flattening* als auch Sequentialisierung der *Statecharts* vorbereiten und die dazugehörige Prioritätenvergabe regeln. Dazu werden in Regel T_1 alle Transitionen, die später mit Prioritäten versehen sein sollen, mit der Priorität $priority(t) = 1$ initialisiert. Anhand von Komponenten, die prioritätsbestimmend sein können, werden zu späteren Zeitpunkten ggf. Erhöhungen und Neuordnungen der Prioritäten an Transitionen vorgenommen.

T 1 (Transition ohne Priorität)

Abbildung 4.1: Transformationsregel für eine Transition ohne Priorität

$$\Leftrightarrow \; s \xrightarrow[p=\varepsilon]{} s' \Longrightarrow s \xrightarrow[p=1]{} s',$$

$$EM_l : s \in \Sigma \cup \{c_{join}\},\ s' \in \Sigma \cup \{c_{choice}, c_{fork}, c_{history^{(*)}}\} \qquad \diamond$$

Die folgenden beiden Regeln tragen der Tatsache Rechnung, dass auch *History*-Verknüpfungen in *Statecharts* sowohl ein- als auch ausgehende Transitionen haben können.

T 2 (Transition nach *History*-Verknüpfung)

Abbildung 4.2: Transformationsregel für eine Transition nach einer History-Verknüpfung

$$\Leftrightarrow \; s \xrightarrow{e[c]/a} s_c(c_{history^{(*)}}) \Longrightarrow s \xrightarrow{e[c]/a} s_c, s_c(c_{history^{(*)}}),$$

$$EM_l : s \in \Sigma,\ s_c \in \Sigma_{cs} \qquad \diamond$$

T 3 (Transition von *History*-Verknüpfung)

Abbildung 4.3: Transformationsregel für eine Transition von einer History-Verknüpfung

$$\Leftrightarrow \; s_c(c_{history^{(*)}}) \xrightarrow{/a} s_c(s_s) \Longrightarrow s_c(c_{history^{(*)}}), s_c(c_{initial}) \xrightarrow{/a} s_c(s_s),$$

$$EM_l : s_s \in V,\ s_c \in \Sigma_{cs} \qquad \diamond$$

Ist nun eine Transition ausgehend von einer Startverknüpfung mit $a \neq \varepsilon$ beschriftet, wie z. B. in der rechten Seite der Regel T_3 zu sehen, ist dem mit Regel T_4 zu begegnen. Die Notwendigkeit der Vorschaltung eines Zustandes s_0 ist dann darin begründet, dass bei der Initialisierung eines Zustandsautomaten in *Ascet-SD* die Eintrittsaktivität des Startzustandes nicht ausgeführt wird. Würde also beispielsweise die Aktion a in die Eintrittsaktivität des Zustandes s migriert werden, würde sie beim ersten (initialen) Betreten des Zustandes nicht ausgeführt werden. Der Nachteil dieser Konstruktion liegt in der Verzögerung der Ausführung des (nun vollständig transformierten) Zustandsautomaten um einen Schritt. Abhilfe schafft die Beschränkung der *Statecharts* auf Transitionen, die von Startverknüpfungen ausgehen, auf solche mit leerer Aktionsmarkierung.

T 4 (Startverknüpfung mit Aktion/Initialisierung)

Abbildung 4.4: Transformationsregel für eine Startverknüpfung mit Aktion

$$\Leftrightarrow \; c_{initial} \xrightarrow{/a} s \Longrightarrow c_{initial} \rightarrow s_0 \xrightarrow{/a} s,$$

$$EM_l : a \neq \varepsilon,\ s \in V,\ entry(s_0) = exit(s_0) = do(s_0) = internalTransition(s_0) = \varepsilon \qquad \diamond$$

T 5 (*Trigger*)

Abbildung 4.5: Transformationsregel für Trigger

$$\Leftrightarrow \; s \xrightarrow{e[c]/a} s' \Longrightarrow s \xrightarrow{e'[c']/a'} s',$$

$$EM_l : s \in \Sigma \cup \{c_{join}\},\ s' \in \Sigma \cup \{c_{choice}, c_{fork}, c_{history^{(*)}}\},\ e' \in E_{change},$$

$$e'[c']/a' = \begin{cases} \text{,call\_}f\text{`}[c \,\&\&\, f]/a \text{ ,} f = false\text{;`} & \text{falls } e = f \in E_{call}, \text{ wobei } f \text{ eine Methode einer Klasse ist} \\ \text{,when\_}param\text{`}[c]/a & \text{falls } e = when(param) \in E_{change} \\ \text{,completion`}/a & \text{falls } e = \varepsilon \in E_{completion} \wedge c = \varepsilon \\ \text{,signal\_}n\text{`}[c \,\&\&\, n]/a \text{ ,} n = false\text{;`} & \text{falls } e \in E_{signal} \\ \text{,after\_}param\text{`}[c]/a & \text{falls } e = after(param) \in E_{time} \end{cases} \qquad \diamond$$

Die *Trigger* werden wie in Abschnitt 4.1.1 beschrieben umgesetzt. Erklärend soll hier noch aufgeführt werden, dass $f \in c'$ bei $e \in E_{call}$ eine Boolesche Variable ist und einem Methodenaufruf für die Methode $f()$ in *Statecharts* gleichkommt. Da diese Methoden in Zustandsautomaten nicht existieren, muss die Variable f immer dann auf das logische *true* gesetzt werden, wenn die Methode $f()$ in *Statecharts* aufgerufen werden würde. Sobald eine Zustandsüberführung, bedingt durch den wahren Wert f, ausgelöst wurde, wird der Wert in der Aktion wieder auf *false* gesetzt, bis der Wert der Variable f erneut wechselt. Der Wert der Variable kann durch geeignete Konstruktionen außerhalb des Zustandsautomaten oder manuell durch Werte-Stimulation in der *Runtime*-Umgebung von *Ascet-SD* beeinflusst werden.

Gleiches wie für die Variable f bei *Call*-Ereignissen gilt für die Boolesche Variable *n* bei *Signal*-Ereignissen, wobei *n* der Ereignisname in *Statecharts* ist. Sie steht jedoch nicht für einen Methodenaufruf, sondern für ein explizites Signal des *Statecharts*. Es obliegt nun dem Modellierer, diese Variable geeignet mit den bereits genannten Mitteln zu beeinflussen.

Die *Choice*-Verknüpfung kann ersetzt werden, indem eine in sie hineingehende Transition mit jeder ausgehenden Transition eine neue Transition bildet und mit dem *Trigger* der eingehenden Transition beschriftet wird. Die Bedingungen der so assoziierten Transitionen werden logisch mit *UND* verknüpft und die Prioritäten werden entsprechend den Prioritäten der eingehenden Transitionen vergeben. Das führt dazu, dass mehrere Transitionen, die von einem Zustand s_i ausgehen, mit derselben Priorität p_i ausgestattet sind, zumal alle Transitionen bisher mit $priority(t) = 1$ versehen wurden (s. Regel T_1). Die Regeln T_{15} bis T_{19} streuen und ordnen dann die Prioritäten anhand steuernder Merkmale.

T 6 (*Choice*-Verknüpfung)

Abbildung 4.6: Transformationsregel für eine Choice-Verknüpfung

$$\Leftrightarrow \{s_i \xrightarrow[p_i]{e_i[c_i]} c_{choice} \xrightarrow{[c_j]/a_j} s_j\} \Longrightarrow \{s_i \xrightarrow[p_i]{e_i[c_i \,\&\&\, c_j]/a_j} s_j\},$$

$$EM_l : s_{i,j} \in \Sigma \cup \{c_{join}, c_{fork}\},\ i = 1, \ldots, |incoming(c_{choice})|,\ j = 1, \ldots, |outgoing(c_{choice})|$$

◇

T 7 (*Deephistory*-Verknüpfung)

Abbildung 4.7: Transformationsregel für eine Deephistory-Verknüpfung

$$\Leftrightarrow \; s_c(c_{history^*}), \{s_c(s_i)\} \Longrightarrow s_c(c_{history}), \{s_c(s_i(c_{history}))\},$$

$$EM_l : s_i \in children^+(s_c) \setminus \Sigma_{ss},\ i = \{1, \ldots, |children^+(s_c) \setminus \Sigma_{ss}|\} \qquad \diamond$$

Soll das Verhalten eines Zustandes bzw. einer Region mit Geschichte nach dem *Flattening* oder der Sequentialisierung nachgeahmt werden, so muss für jeden Zustand bzw. jede Region festgehalten werden, welcher Unterzustand vor Verlassen des Zustandes bzw. der Region aktiv war. Bei der Umsetzung kommt die Tatsache sehr gelegen, dass in Zustandsautomaten von *Ascet-SD* jedem Zustand ein eindeutiger *Integer*-Wert zugeordnet wird. Diese kann man dann wie eine Konstante benutzen. Damit kann man nun für jeden Zustand bzw. jede Region mit Geschichte eine Variable vom Typ *integer* anlegen, in der beim Verlassen des Hierarchiezustandes bzw. der Region die Nummer des zuletzt aktiven Zustandes gesichert wird. Die Voraussetzungen für die Nachahmung der Geschichte-Eigenschaft werden mit den Regeln T_8 bis T_{11} geschaffen.

T 8 (*History*-Attribut für einen seriellen Zustand)

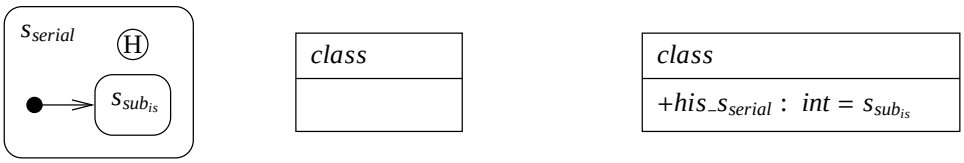


Abbildung 4.8: Transformationsregel für das History-Attribut für einen seriellen Zustand

$$\nexists(\text{Attribut } public\ his\_s_{serial} : int) \Longrightarrow (\text{Attribut } public\ his\_s_{serial} : int = s_{sub_{is}}),$$

$$EM_l : \exists s_{serial} \in \Sigma_{serial},\ \exists s_{serial}(c_{history}),\ \exists s_{serial}(s_{sub_{is}}) \in children(s_{serial}) \cap \Sigma_{is} \qquad \diamond$$

T 9 (*History*-Attribute für einen orthogonalen Zustand)

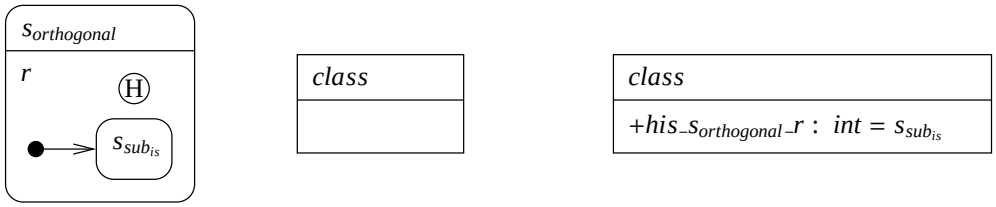


Abbildung 4.9: Transformationsregel für die History-Attribute für einen orthogonalen Zustand

$\nexists$(Attribut *public* $his\_s_{orthogonal}\_r : int$) $\Longrightarrow$ (Attribut *public* $his\_s_{orthogonal}\_r : int = s_{sub_{is}}$),
$EM_l : \exists s_{orthogonal} \in \Sigma_{orthogonal},\ \exists s_{orthogonal}[r(c_{history})],\ \exists s_{orthogonal}[r(s_{sub_{is}})] \in$
$children(s_{orthogonal}[r]) \cap \Sigma_{is}$ ◇

Mit den beiden nun folgenden Regeln schreibt jeder Zustand beim Verlassen seine Identität in die Variable des nächsthöheren Hierarchiezustandes. Diese Variable kann dann wieder benutzt werden, um abzufragen, welcher Unterzustand als letzter aktiv war.

T 10 (Setzen der Geschichte im seriellen Zustand)

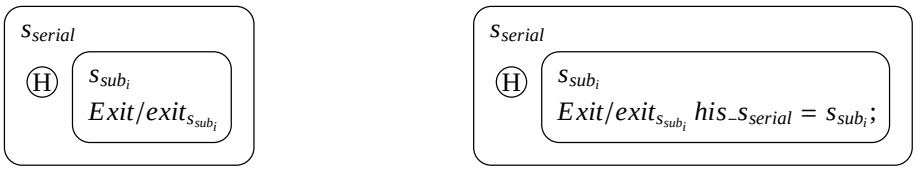


Abbildung 4.10: Transformationsregel für das Setzen der Geschichte im seriellen Zustand

$$s_{serial}(c_{history}), \{s_{serial}(s_{sub_i})\}, exit(s_{serial}(s_{sub_i})) = exit_{s_{sub_i}}$$
$$\Longrightarrow$$
$$s_{serial}(c_{history}), \{s_{serial}(s_{sub_i})\}, exit(s_{serial}(s_{sub_i})) = exit_{s_{sub_i}}\ ,his\_s_{serial} = s_{sub_i};\text{‘},$$

$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_i}) \in children(s_{serial}),\ i = 1, \ldots, |children(s_{serial})|,$
$,his\_s_{serial} = s_{sub_i};\text{‘} \notin exit_{s_{sub_i}}$ ◇

T 11 (Setzen der Geschichte im orthogonalen Zustand)

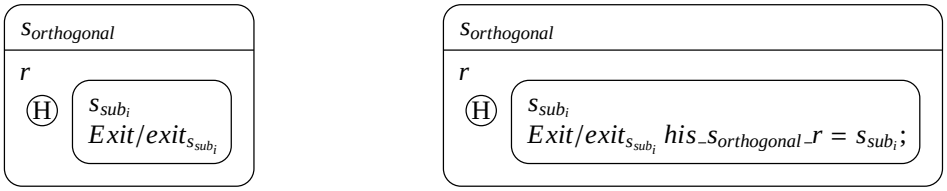


Abbildung 4.11: Transformationsregel für das Setzen der Geschichte im orthogonalen Zustand

$$s_{orthogonal}[r(c_{history})], \{s_{orthogonal}[r(s_{sub_i})]\}, exit(s_{orthogonal}[r(s_{sub_i})]) = exit_{s_{sub_i}}$$
$$\Longrightarrow$$
$$s_{orthogonal}[r(c_{history})], \{s_{orthogonal}[r(s_{sub_i})]\},$$
$$exit(s_{orthogonal}[r(s_{sub_i})]) = exit_{s_{sub_i}}, his\_s_{orthogonal}\_r = s_{sub_i}; `,$$

$EM_l : s_{orhogonal} \in \Sigma_{orthogoanl},\ s_{orthogonal}[r(s_{sub_i})] \in children(s_{orthogonal}[r]),\ i = 1, \ldots,$
$|children(s_{orthogonal}[r])|,\ , his\_s_{orthogonal}\_r = s_{sub_i}; ` \notin exit_{s_{sub_i}}$ ◇

Erfolgt eine Transition direkt in einen Unterzustand eines orthogonalen Zustandes, ist zu beachten, dass die Unterzustände aller anderen Regionen entsprechend der Geschichte der jeweiligen Region auch betreten werden. Dafür werden Transitionen, die die Unterzustände als Ziel haben, mit einer entsprechenden Bedingung ausgestattet, die das Ziel verifiziert. Entsprechend werden beim Verlassen des orthogonalen Zustandes die Unterzustände aller Regionen verlassen. Dabei werden die Informationen über aktive Zustände gesichert.

T 12 (*Fork*-Verknüpfung)

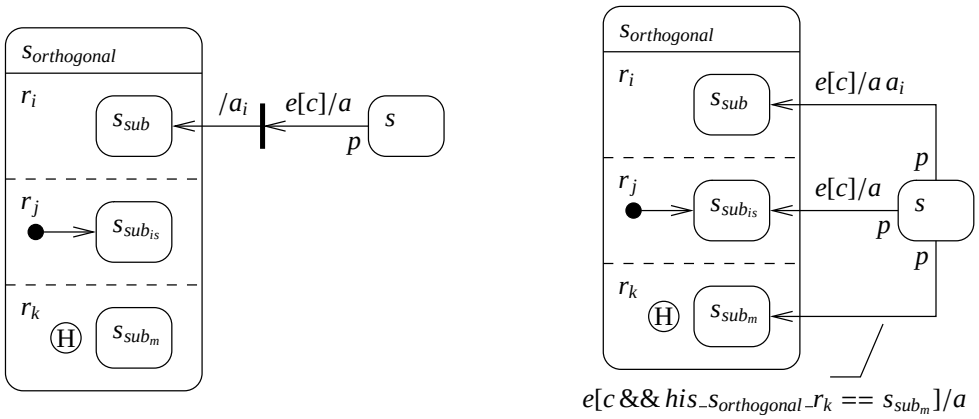


Abbildung 4.12: Transformationsregel für die Fork-Verknüpfung

$$\Leftrightarrow\ s \xrightarrow[p]{e[c]/a} c_{fork}\{\xrightarrow{/a_i} s_{orthogonal}[r_i(s_{sub})]\}, \{s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\}, \{s_{orthogonal}[r_k(s_{sub_m})]\}$$
$$\Longrightarrow$$
$$s\{\xrightarrow[p]{e[c]a\, a_i} s_{orthogonal}[r_i(s_{sub})]\}, s\{\xrightarrow[p]{e[c]a} s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$s\{\xrightarrow[p]{e[c \,\&\&\, his\_s_{orthogonal}\_r_k == s_{sub_i}]/a} s_{orthogonal}[r_k(s_{sub_m})]\}, \{s_{orthogonal}[r_k(c_{history})]\},$$

$EM_l : \nexists s_{orthogonal}[r_j(c_{history})],\ s \in \Sigma \cup \{c_{join}\},\ s_{orthogonal}[r_i(s_{sub})] \in \Sigma,\ i \neq j \neq k,$
optional: $children^*(s_{orthogonal}[r_j]), children^*(s_{orthogonal}[r_k])$ ◇

T 13 (*Join*-Verknüpfung)

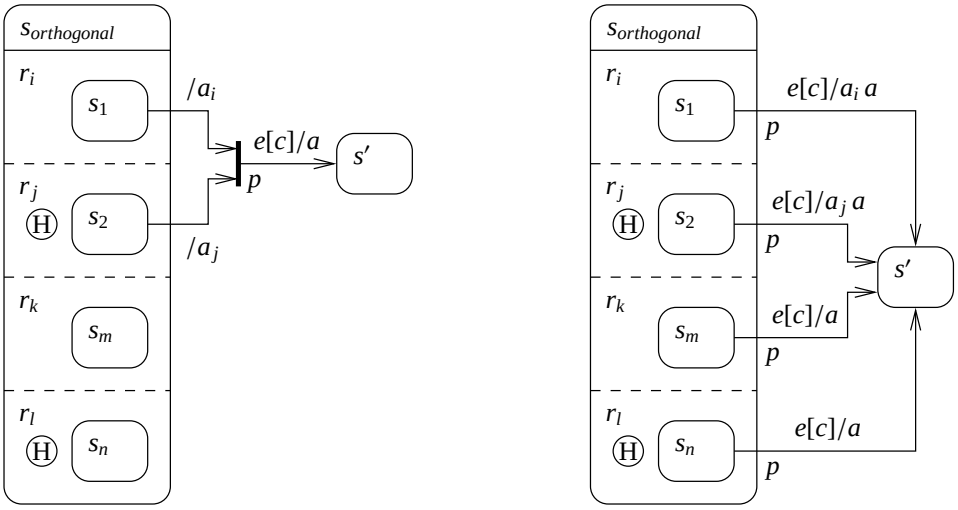


Abbildung 4.13: Transformationsregel für die Join-Verknüpfung

$$\Leftrightarrow\ \{s_{orthogonal}[r_i(s_1)] \xrightarrow{/a_i}\} c_{join},\ s_{orthogonal}[r_j(c_{history})],\ \{s_{orthogonal}[r_j(s_2)] \xrightarrow{/a_j}\} c_{join},$$
$$c_{join} \xrightarrow[p]{e[c]/a} s',\ \{s_{orthogonal}[r_k(s_m)]\},\ \{s_{orthogonal}[r_l(c_{history})]\},\ \{s_{orthogonal}[r_l(s_n)]\}$$
$$\Longrightarrow$$
$$\{s_{orthogonal}[r_i(s_1)] \xrightarrow[p]{e[c]/a_i\ a}\} s',\ s_{orthogonal}[r_j(c_{history})],$$
$$\{s_{orthogonal}[r_j(s_2)] \xrightarrow[p]{e[c]/a_j\ a}\} s',\ \{s_{orthogonal}[r_k(s_m)] \xrightarrow[p]{e[c]/a}\} s',$$
$$\{s_{orthogonal}[r_l(c_{history})]\},\ \{s_{orthogonal}[r_l(s_n)] \xrightarrow[p]{e[c]/a}\} s',$$

$EM_l : \nexists(s_{orthogonal}[r_i(c_{history})], s_{orthogonal}[r_k(c_{history})]),\ s_{\{1,2,m,n\}}, s' \in \Sigma,\ r_i \neq r_j \neq r_k \neq r_l,$
optional: $(children^*(s_{orthogonal}[r_i]) \vee children^*(s_{orthogonal}[r_j])),\ children^*(s_{orthogonal}[r_k]),$
$children^*(s_{orthogonal}[r_l])$ ◇

T 14 (*Endzustand*)

Abbildung 4.14: Transformationsregel für einen Endzustand

$$\Leftrightarrow \{s_i \xrightarrow[p_i]{e_i[c_i]/a_i}\} FinalState \Longrightarrow \{s_i \xrightarrow[p_i]{e_i[c_i]/a_i}\} s_f,$$

$EM_l : s_i, s_f \in \Sigma \setminus \Sigma_{fs},\ FinalState \in \Sigma_{fs},\ i = 1, \ldots, |\bigcup_{t=(s,l_T,FinalState)} t|,\ entry(s_f) = exit(s_f) = do(s_f) = internalTransition(s_f) = \varepsilon$ ◇

Folgend sollen nun gleiche Prioritäten von verschiedenen Transitionen, die von ein und demselben Zustand ausgehen, verändert und sortiert werden. Dazu werden Informationen über die implizite Vergabe von Prioritäten in *Artisan RtS* benutzt. Die Sortierung erfolgt durch wiederholte Vertauschungen von Prioritäten, wie sie auch bei gängigen Sortieralgorithmen, wie etwa *Bubble-Sort*, vorgenommen werden. Die Wiederholungen sind durch die Steuersprache in Abschnitt 4.3.5 genau beschrieben. Zunächst werden jedoch in Regel T_{15} allen von einem Zustand ausgehenden Transitionen eindeutige Prioritäten zugeordnet, indem die vorhandenen gestreut werden.

T 15 (Transitionen mit gleicher Priorität)

Abbildung 4.15: Transformationsregel für Transitionen mit gleicher Priorität

$$\Leftrightarrow s_1' \xleftarrow[p]{e_1[c_1]/a_1} s \xrightarrow[p]{e_2[c_2]/a_2} s_2' \Longrightarrow s_1' \xleftarrow[p']{e_1[c_1]/a_1} s \xrightarrow[p]{e_2[c_2]/a_2} s_2',\ p' = \bar{p}(t_1 = (s, l_T, s_1')) + 1,$$

$$EM_l : s, s_1', s_2' \in \Sigma \qquad \diamond$$

Für die Sortierung der Prioritäten an Transitionen wird in Regel T_{16} die Tatsache benutzt, dass in *Statecharts* Transitionen mit *Completion*-Ereignis immer vor Transitionen mit vom Modellierer definierten Ereignissen auf Ausführung geprüft werden.

T 16 (Prioritäten bei *Completion*-Ereignis)

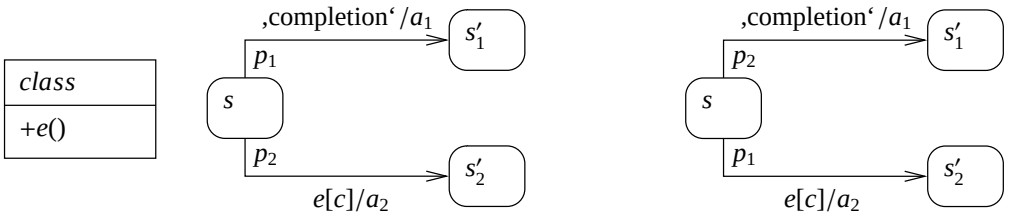


Abbildung 4.16: Transformationsregel für Prioritäten bei Completion-Ereignis

$$\Leftrightarrow\ s_1' \xleftarrow[p_1]{\text{,completion'}/a_1} s \xrightarrow[p_2]{e[c]/a_2} s_2' \Longrightarrow s_1' \xleftarrow[p_2]{\text{,completion'}/a_1} s \xrightarrow[p_1]{e[c]/a_2} s_2',$$
$$p_1 = priority(s, l_{T_1}, s_1') < p_2 = priority(s, l_{T_2}, s_2'),$$

$EM_l : s, s_1', s_2' \in \Sigma,\ e \in E$ ist in einer deklarierten Klasse *class* gegeben ◇

In Regel T_{17} werden Prioritäten nach dem Ereignis geordnet, welches vom Modellierer zuerst definiert wurde.

T 17 (Prioritäten bei definierten Ereignissen)

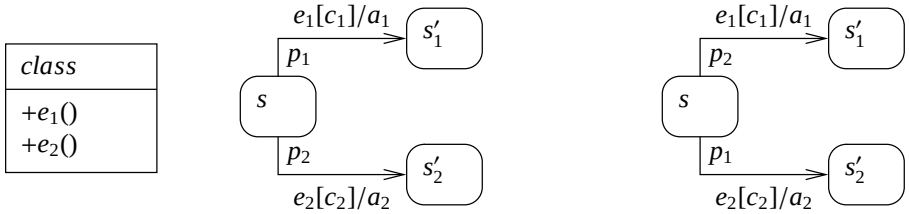


Abbildung 4.17: Transformationsregel für Prioritäten bei definierten Ereignissen

$$\Leftrightarrow\ s_1' \xleftarrow[p_1]{e_1[c_1]/a_1} s \xrightarrow[p_2]{e_2[c_2]/a_2} s_2' \Longrightarrow s_1' \xleftarrow[p_2]{e_1[c_1]/a_1} s \xrightarrow[p_1]{e_2[c_2]/a_2} s_2',$$
$$p_1 = priority(s, l_{T_1}, s_1') < p_2 = priority(s, l_{T_2}, s_2'),$$

$EM_l : s, s_1', s_2' \in \Sigma,$ Reihenfolge auf e_i ist in einer deklarierten Klasse *class* gegeben, $e_i \in E$ ◇

Werden zwei Transitionen mit dem *Completion*-Ereignis ausgelöst und benutzt nur eine Transition eine definierte Variable *attr*, hat die Transition die höhere Priorität, in deren Aktion *attr* nicht benutzt wird. (s. Regel T_{18}).

T 18 (Prioritäten bei Benutzung von Attributen)

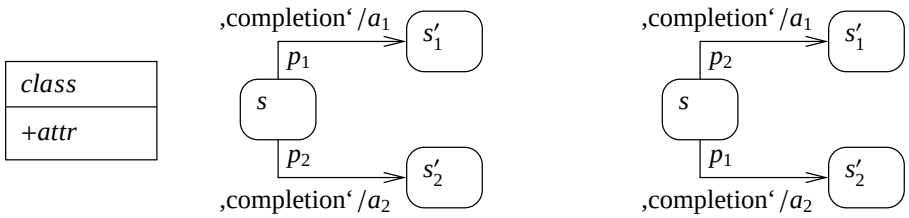


Abbildung 4.18: Transformationsregel für Prioritäten bei Benutzung von Attributen

$$\Leftrightarrow\ s_1' \xleftarrow[p_1]{\text{,completion'}/a_1} s \xrightarrow[p_2]{\text{,completion'}[c]/a_2} s_2' \Longrightarrow s_1' \xleftarrow[p_2]{\text{,completion'}/a_1} s \xrightarrow[p_1]{\text{,completion'}[c]/a_2} s_2',$$
$$p_1 = priority(s, l_{T_1}, s_1') < p_2 = priority(s, l_{T_2}, s_2'),$$

$EM_l : s, s_1', s_2' \in \Sigma,\ attr \in Var$ ist in einer deklarierten Klasse *class* gegeben,
$attr \notin a_1,\ attr \in a_2$ ◇

Demgegenüber führt die Benutzung der zuerst definierten Variable in der Aktionsmarkierung einer Transition zu höherer Priorität einer Transition (s. Regel T_{19}).

T 19 (Prioritäten bei definierten Attributen)

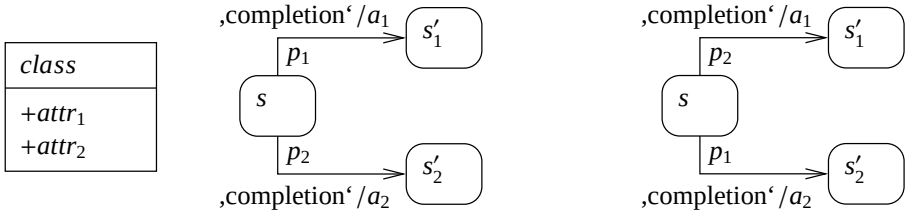


Abbildung 4.19: Transformationsregel für Prioritäten bei definierten Attributen

$$\Leftrightarrow\ s_1 \xleftarrow[p_1]{\text{,completion'}/a_1} s \xrightarrow[p_2]{\text{,completion'}/a_2} s_2 \Longrightarrow s_1 \xleftarrow[p_2]{\text{,completion'}/a_1} s \xrightarrow[p_1]{\text{,completion'}/a_2} s_2,$$
$$p_1 = p(s, l_{T_1}, s_1') < p_2 = p(s, l_{T_2}, s_2'),$$

$EM_l : s, s_1', s_2' \in \Sigma,$ Reihenfolge auf $attr_i$ ist in einer deklarierten Klasse *class* gegeben,
$attr_1 \in a_1,\ attr_2 \in a_2,\ attr_1, attr_2 \in Var$ ◇

Interne Transitionen werden umgesetzt, in dem Bedingungsmarkierung *c* und Aktionsmarkierung *a* der internen Transition in den ‚if . . . then'-Konstrukt der *ESDL* eingebettet werden.

T 20 (Interne Transitionen)

Abbildung 4.20: Transformationsregel für interne Transitionen

$$\Leftrightarrow\ l_{T_i}(internalTransition(s)) = [c_i]/a_i,\ do(s) = do_s \Longrightarrow do(s) = do_s \text{ ,if } (c_i)\ \{a_i\};\text{‘}$$

$$EM_l : s \in \Sigma,\ i = 1, \ldots, \text{Anzahl interner Transitionen von } s \qquad \diamond$$

Nun sollen noch Werte des Datentyps *octal* in Werte des Datentyps *integer* transformiert werden. Dazu wird eine Konvertierung der Basisdarstellung der Variablenwerte notwendig.

T 21 (Variable des Typs *octal*)

Abbildung 4.21: Transformationsregel für eine Variable des Typs octal

$$\Leftrightarrow\ attr_8 \Longrightarrow attr_{10}$$

$$\text{ein Attribut } attr_8 \text{ ist in einer deklarierten Klasse } class \text{ gegeben} \qquad \diamond$$

In Regel T_{21} bedeutet die Indexzahl einer öffentlichen Variable *attr* das Zahlenbasissystem, indem die Variable notiert ist. Folgend soll beschrieben werden, wie die Konvertierung zwischen Zahlensystemen mit verschiedener Basis erfolgt.

In einem Zahlensystem mit der Basis $q \in \mathbb{N}^+$ wird eine reelle Zahl z durch eine Summe von Potenzen dargestellt:

$$z = \pm(a_k q^k + a_{k-1} q^{k-1} + \ldots + a_0 q^0 + a_{-1} q^{-1} + a_{-2} q^{-2} + \ldots + a_{j+1} q^{j+1} + a_j q^j).$$

Die Ziffern a_i sind dabei je eine der nichtnegativen ganzen Zahlen $0, 1, \ldots, q-1$. Speziell im Oktalsystem ist die Basis $q = 8$.

Die Umrechnung von einem Zahlensystem in ein anderes wird als *Konvertierung* bezeichnet. Werden mehrere Zahlensysteme gleichzeitig benutzt, so ist es zur Vermeidung von Irrtümern üblich, die Basis als (Dezimal-) Index an die darzustellende Zahl anzuhängen. Bei der Konvertierung führt man eine Zahl

$$z_q = a_k q^k + a_{k-1} q^{k-1} + \ldots + a_0 q^0$$

aus der q-Darstellung in die p-Darstellung

$$z_p = b_l p^l + b_{l-1} p^{l-1} + \ldots + b_0 p^0 \tag{4.1}$$

über, indem man z_q durch p_q dividiert. Dabei erhält man einen ganzzahligen Teil und einen Rest. Dieser Rest nimmt dabei die Werte $0, 1, \ldots, p_q - 1$ an und ist die Ziffer b_0 in Gleichung (4.1). Durch wiederholtes Dividieren des entstandenen ganzzahligen Teils erhält man die Reste, die den Koeffizienten $b_1, b_2, \ldots, b_l$ entsprechen.

Beispiel 1 Wie lautet die Dezimaldarstellung der Oktalzahl $z_q = 204_8$? Die Summendarstellung für z_q ist $204_8 = 2 \times 8^2 + 0 \times 8^1 + 4 \times 8^0$ und wird durch Division durch $p_q = 12_8 (= 10_{10})$ konvertiert:

$$\begin{aligned} 204 : 12 = 15 + 2/12 \quad &\Rightarrow \quad b_0 = 2_8 = 2_{10}, \\ 15 : 12 = 1 + 3/12 \quad &\Rightarrow \quad b_1 = 3_8 = 3_{10}, \\ 1 : 12 = 0 + 1/12 \quad &\Rightarrow \quad b_2 = 1_8 = 1_{10}. \end{aligned}$$

Man erhält $z_p = b_2 \times 10^2 + b_1 \times 10^1 + b_0 \times 10^0 = 132$. □

Entsprechend verfährt man für die Berechnung des gebrochenen Anteils der darzustellenden Zahl z. Die Grundlage für die Berechnung ist hier die Darstellung des Nachkommaanteils

$$z_q = a_{-1}q^{-1} + a_{-2}q^{-2} + \ldots + a_{j+1}q^{j+1} + a_j q^j.$$

4.3.2 *Flattening* von *Statecharts*

Transformationsregeln, die im Folgenden als *Flattening*-Regeln eingeführt werden, führen serielle Hierarchiezustände, deren Unterzustände sowie deren Abhängigkeiten untereinander in Basiszustände über unter Erhalt der Ausführung von Aktivitäten und Transitionen, d. h. implizite Verhaltensweisen werden in explizite überführt. Beispiele für *Flattening* finden sich in der Literatur in [AY98, BLA+99, Bjö01, GPP98, HW98, PMP01, Sim00], jedoch geht keine der Arbeiten so dezidiert und systematisch auf alle Effekte bei *Statecharts* ein. *Flattening* wird für *Statecharts* in dieser Arbeit zum einen für die Nachahmung der Auswertung der Transitionen von der untersten zur obersten Hierarchiestufe in Zustandsautomaten und zum anderen als Vorbereitung der Sequentialisierung notwendig.

In einer *Flattening*-Regel wird ein Teil-*Statechart* behandelt, das auch mehrfach in einem *Statechart* vorhanden sein kann. Dies geschieht durch die Mehrfachanwendung der Regel, die durch die Steuerung in Abschnitt 4.3.5 beschrieben wird. Es werden immer gerade nur die Hierarchiezustände, die das niedrigste Hierarchieniveau besitzen, betrachtet. Nachdem nun dieses Hierarchieniveau durch Anwendung aller *Flattening*-Regeln eingeebnet wurde, wird durch die wiederholte Anwendung des gesamten *Flattening*s das nächsthöhere Niveau transformiert, usw. Dies geschieht solange, bis kein serieller Hierarchiezustand mehr vorhanden ist.

Der Entwurf der *Flattening*-Regeln ermöglicht in hohem Maße die Wiederverwendbarkeit bei der Anwendung der Regeln auf andere *Statecharts*. Das wird zum einen durch die inhaltliche Abgrenzung zu den anderen Transformationsregeln, aber auch durch die Modularisierung der *Flattening*-Regeln erreicht. So sind z. B. die Regeln, die Prioritätsveränderungen betreffen, streng getrennt vom Rest der *Flattening*-Regeln. Dem entsprechend betrifft *Flattening*-Regel F_1 die Anpassung der Prioritäten; die *Flattening*-Regeln

F_2 bis F_{12} betrachten systematisch alle in seriellen Hierarchiezuständen möglichen Transitionen und verändern diese gegebenenfalls; mit den *Flattening*-Regeln F_{13} bis F_{16} wird das *Flattening* mit der Löschung des Hierarchiezustandes und aller seiner Markierungen terminiert. Ist ein Hierarchiezustand mit Geschichte ausgestattet, wird diese berücksichtigt.

Vorbereitend werden nun also in Regel F_1 die Prioritäten an das Modell der Zustandsautomaten angepasst, indem die Prioritäten der Transitionen, die vom innersten Unterzustand abgehen, über die Prioritäten der Transitionen des Hierarchiezustandes gesetzt werden.

F 1 (Prioritäten am seriellen Hierarchiezustand)

Abbildung 4.22: Flattening-Regel für Prioritäten am seriellen Hierarchiezustand

$$\Leftrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p_i]{e_i[c_i]/a_i} s_n\} \Longrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p'_i]{e_i[c_i]/a_i} s_n\},$$

$$\text{wobei } p'_i = \bar{p}(outgoing(s_{serial})) + p_i,$$

$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_m}) \in \Sigma_{ss},\ s_n \in children(s_{serial}) \cup \complement children(s_{serial}) \cap \Sigma,\ i = 1, \ldots, \left| \bigcup_{\substack{t=(s_{serial}(s),l_T,s'),\\ s_{serial}(s)\in\Sigma_{ss},s'\in\Sigma}} t \right|,\ m, n \leq i$ ◇

Beginnend mit der *Flattening*-Regel F_2 werden nun systematisch alle Transitionen des Hierarchiezustandes betrachtet, die sich an und in ihm gruppieren, von diesem abgelöst und an seine Unterzustände angebracht. Dabei werden Fallunterscheidungen zwischen Hierarchiezuständen mit und ohne Geschichte separat aufgeführt.

F 2 (Transition des Typs (s_{sub}, l_T, s_{sub}))

Abbildung 4.23: Flattening-Regel für eine Transition des Typs (s_{sub}, l_T, s_{sub})

$$\Leftrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p_i]{e_i[c_i]/a_i} s_{serial}(s_{sub_n})\} \Longrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p_i]{e_i[c_i],/do_{s_{serial}}\ a_i} s_{serial}(s_{sub_n})\},$$

$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_m}),\ s_{serial}(s_{sub_n}) \in \Sigma_{ss},\ i = 1, \ldots, \left| \bigcup_{\substack{t=(s_{serial}(s), l_T, s_{serial}(s')), \\ s_{serial}(s), s_{serial}(s') \in \Sigma_{ss}}} t \right|,$
$m, n \leq i$ ◇

F 3 (Transition des Typs $(s_{sub}, l_T, s_{serial})$, $s_{serial} \notin \Sigma_{his}$)

Abbildung 4.24: Flattening-Regel für eine Transition des Typs $(s_{sub}, l_T, s_{serial})$, $s_{serial} \notin \Sigma_{his}$

$$\Leftrightarrow\ s_{serial}(s_{sub_{is}}), s_{serial}(s_{sub}) \xrightarrow[p]{e[c]/a} s_{serial} \Longrightarrow s_{serial}(s_{sub}) \xrightarrow[p]{e[c]/do_{s_{serial}}\, a} s_{serial}(s_{sub_{is}}),$$

$$EM_l : s_{serial} \in \Sigma_{serial} \setminus \Sigma_{his},\ s_{serial}(s_{sub_{is}}) \in \Sigma_{is},\ s_{serial}(s_{sub}), s_{serial}(s_{sub_{is}}) \in \Sigma_{ss} \quad ◇$$

F 4 (Transition des Typs $(s_{sub}, l_T, s_{serial})$, $s_{serial} \in \Sigma_{his}$)

Abbildung 4.25: Flattening-Regel für eine Transition des Typs $(s_{sub}, l_T, s_{serial})$, $s_{serial} \in \Sigma_{his}$

$$\Leftrightarrow\ s_{serial}(c_{history}), s_{serial}(s_{sub}) \xrightarrow[p]{e[c]/a} s_{serial}$$

$$\Longrightarrow$$

$$s_{serial}(c_{history}), s_{serial}(s_{sub}) \xrightarrow[p]{e[c]/do_{s_{serial}}\, a} s_{serial}(s_{sub}),$$

$$EM_l : s_{serial} \in \Sigma_{serial} \cap \Sigma_{shis},\ s_{serial}(s_{sub}) \in \Sigma_{ss} \quad ◇$$

F 5 (Transition des Typs (s_{sub}, l_T, $s_{outside}$))

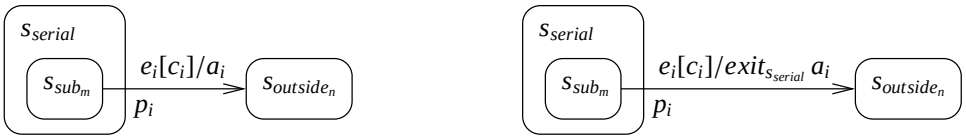


Abbildung 4.26: Flattening-Regel für eine Transition des Typs (s_{sub}, l_T, $s_{outside}$)

$$\Leftrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p_i]{e_i[c_i]/a_i} s_{outside_n}\} \Longrightarrow \{s_{serial}(s_{sub_m}) \xrightarrow[p_i]{e_i[c_i]/exit_{s_{serial}}\ a_i} s_{outside_n}\},$$

$$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_m}) \in \Sigma_{ss},\ i = 1, \ldots, \left| \bigcup_{\substack{t=(s_{serial}(s),l_T,s_{outside}),\\ s_{serial}(s)\in\Sigma_{ss},\\ s_{outside}\in\Sigma\setminus children^*(s_{serial})}} t \right|,\ m,n \leq i \qquad \diamond$$

F 6 (Transition des Typs (s_{serial}, l_T, s_{sub}))

Abbildung 4.27: Flattening-Regel für eine Transition des Typs (s_{serial}, l_T, s_{sub})

$$\Leftrightarrow\ s_{serial} \xrightarrow[p]{e[c]/a} s_{serial}(s_{sub}),\ \{s_{serial}(s_{sub_i})\} \Longrightarrow \{s_{serial}(s_{sub_i}) \xrightarrow[p]{e[c]/exit_{s_{serial}}\ a\ entry_{s_{serial}}} \} s_{serial}(s_{sub}),$$

$$EM_l : s_{serial} \in \Sigma_{serial},\ s_{sub}, s_{sub_i} \in \Sigma_{ss},\ i = 1, \ldots, |children(s_{serial})| \qquad \diamond$$

F 7 (Transition des Typs (s_{serial}, l_T, s_{serial}), $s_{serial} \notin \Sigma_{his}$)

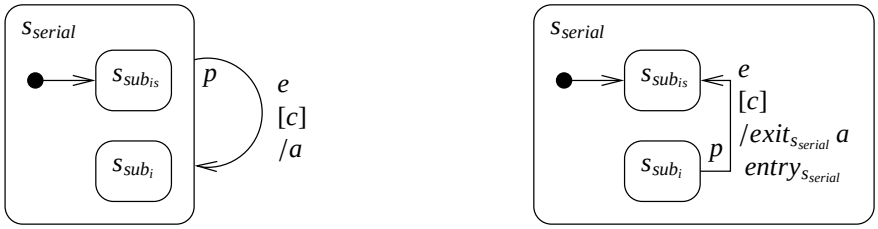


Abbildung 4.28: Flattening-Regel für eine Transition des Typs (s_{serial}, l_T, s_{serial}), $s_{serial} \notin \Sigma_{his}$

$$\Leftrightarrow\ s_{serial} \xrightarrow[p]{e[c]/a} s_{serial}, s_{serial}(s_{sub_{is}}), \{s_{serial}(s_{sub_i})\}$$

$$\Longrightarrow$$

$$\{s_{serial}(s_{sub_i}) \xrightarrow[p]{e[c]/exit_{s_{serial}}\ a\ entry_{s_{serial}}} \} s_{serial}(s_{sub_{is}}),$$

$EM_l : s_{serial} \in \Sigma_{serial} \setminus \Sigma_{his},\ s_{serial}(s_{sub_i}), s_{serial}(s_{sub_{is}}) \in \Sigma_{ss},\ s_{serial}(s_{sub_{is}}) \in \Sigma_{is},\ i = 1, \ldots, |children(s_{serial})|$ ◇

F 8 (Transition des Typs $(s_{serial}, l_T, s_{serial})$, $s_{serial} \in \Sigma_{his}$)

Abbildung 4.29: Flattening-Regel für eine Transition des Typs $(s_{serial}, l_T, s_{serial})$, $s_{serial} \in \Sigma_{his}$

$$\Leftrightarrow\ s_{serial} \xrightarrow[p]{e[c]/a} s_{serial}, \{s_{serial}(s_{sub_i})\} \Longrightarrow \{s_{serial}(s_{sub_i}) \xrightarrow[p]{e[c]/exit_{s_{serial}}\ a\ entry_{s_{serial}}} s_{serial}(s_{sub_i})\},$$

$$EM_l : s_{serial} \in \Sigma_{serial} \cap \Sigma_{his},\ s_{serial}(s_{sub_i}) \in \Sigma_{ss},\ i = 1, \ldots, |children(s_{serial})|$$ ◇

F 9 (Transition des Typs $(s_{serial}, l_T, s_{outside})$)

Abbildung 4.30: Flattening-Regel für eine Transition des Typs $(s_{serial}, l_T, s_{outside})$

$$\Leftrightarrow\ s_{serial} \xrightarrow[p]{e[c]/a} s_{outside}, \{s_{serial}(s_{sub_i})\} \Longrightarrow \{s_{serial}(s_{sub_i}) \xrightarrow[p]{e[c]/exit_{s_{serial}}\ a} \} s_{outside},$$

$$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_i}) \in \Sigma_{ss},\ i = 1, \ldots, |children(s_{serial})|$$ ◇

F 10 (Transition des Typs ($s_{outside}$, l_T, s_{sub}))

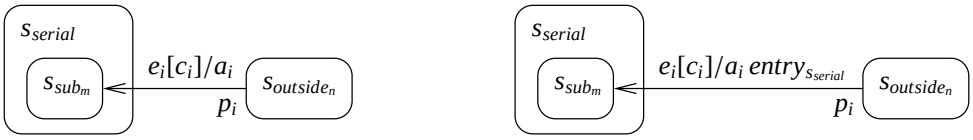


Abbildung 4.31: Flattening-Regel für eine Transition des Typs ($s_{outside}$, l_T, s_{sub})

$$\Leftrightarrow \{s_{outside_n} \xrightarrow[p_i]{e_i[c_i]/a_i} s_{serial}(s_{sub_m})\} \Longrightarrow \{s_{outside_n} \xrightarrow[p_i]{e_i[c_i]/a_i\ entry_{s_{serial}}} s_{serial}(s_{sub_m})\},$$

$$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub}) \in \Sigma_{ss},\ i = 1, \ldots, \left| \bigcup_{\substack{t=(s,l_T,s_{serial}(s')),\\ s\in\Sigma\setminus children^*(s_{serial}),\\ s_{serial}(s')\in\Sigma_{ss}}} t \right|,\ m, n \leq i \qquad \diamond$$

F 11 (Transition des Typs ($s_{outside}$, l_T, s_{serial}), $s_{serial} \notin \Sigma_{his}$)

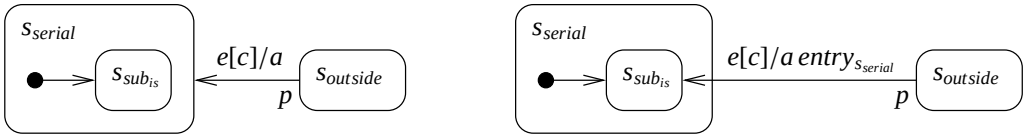


Abbildung 4.32: Flattening-Regel für eine Transition des Typs ($s_{outside}$, l_T, s_{serial}), $s_{serial} \notin \Sigma_{his}$

$$\Leftrightarrow s_{outside} \xrightarrow[p]{e[c]/a} s_{serial}, s_{serial}(s_{sub_{is}}) \Longrightarrow s_{outside} \xrightarrow[p]{e[c]/a\ entry_{s_{serial}}} s_{serial}(s_{sub_{is}}),$$

$$EM_l : s_{serial} \in \Sigma_{serial} \setminus \Sigma_{his},\ s_{serial}(s_{sub_{is}}) \in \Sigma_{ss} \cap \Sigma_{is} \qquad \diamond$$

F 12 (Transition des Typs ($s_{outside}$, l_T, s_{serial}), $s_{serial} \in \Sigma_{his}$)

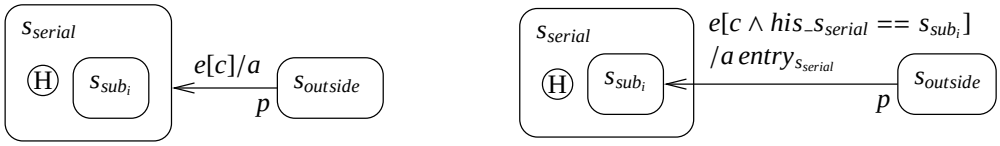


Abbildung 4.33: Flattening-Regel für eine Transition des Typs ($s_{outside}$, l_T, s_{serial}), $s_{serial} \in \Sigma_{his}$

$$\Leftrightarrow s_{outside} \xrightarrow[p]{e[c]/a} s_{serial}, \{s_{serial}(s_{sub_i})\} \Longrightarrow s_{outside}\{\xrightarrow[p]{e[c\wedge his\_s_{serial}==s_{sub_i}]/a\ entry_{s_{serial}}} s_{serial}(s_{sub_i})\},$$

$$EM_l : s_{serial} \in \Sigma_{serial} \cap \Sigma_{his},\ s_{serial}(s_{sub_i}) \in \Sigma_{ss},\ i = 1, \ldots, |children(s_{serial})| \qquad \diamond$$

Damit wurde nun ein transitionsloser Hierarchiezustand erzeugt. Mit den *Flattening*-Regeln F_{13} und F_{14} soll nun herausgefunden werden, ob ein Unterzustand des Hierarchiezustandes nach dem *Flattening* ein Startzustand sein kann. Das kann er offensichtlich nur dann, wenn der Hierarchiezustand ein Startzustand ist (so zu sehen in *Flattening*-Regel F_{14}).

F 13 (Startverknüpfung, $s_{serial} \notin \Sigma_{is}$)

Abbildung 4.34: Flattening-Regel für eine Startverknüpfung, $s_{serial} \notin \Sigma_{is}$

$$\Leftrightarrow \; s_{serial}(c_{initial}) \rightarrow s_{serial}(s_{sub}) \Longrightarrow s_{serial}(s_{sub}),$$

$$EM_l : s_{serial} \in \Sigma_{serial} \setminus \Sigma_{is},\ s_{sub} \in \Sigma_{ss},\ incoming(s_{serial}) = \emptyset,\ outgoing(s_{serial}) = \emptyset \qquad \diamond$$

F 14 (Startverknüpfung, $s_{serial} \in \Sigma_{is}$)

Abbildung 4.35: Flattening-Regel für eine Startverknüpfung, $s_{serial} \in \Sigma_{is}$

$$\Leftrightarrow \; c_{initial} \rightarrow s_{serial}(s_{sub}) \Longrightarrow s_{serial}(s_{sub}),$$

$$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub}) \in \Sigma_{ss} \cap \Sigma_{is},\ incoming(s_{serial}) = \emptyset,\ outgoing(s_{serial}) = \emptyset$$
$$\diamond$$

F 15 (*Shallowhistory*-Verknüpfung)

Abbildung 4.36: Flattening-Regel für eine Shallowhistory–Verknüpfung

$$\Leftrightarrow\ s_{serial}(c_{history}) \Longrightarrow s_{serial},$$

$EM_l : s_{serial} \in \Sigma_{serial},\ \nexists children(children(s_{serial})),\ incoming(s_{serial}) = \emptyset,$
$outgoing(s_{serial}) = \emptyset$ ◇

Mit *Flattening*-Regel F_{15} wurden verbleibende *History*-Verknüpfungen aus den nun transitionslosen Hierarchiezuständen gelöscht. Nun kann der serielle Hierarchiezustand zusammen mit seinen Markierungen aus seinem Kontext gelöscht werden, nicht aber ohne seine statische Aktivität in den statischen Aktivitäten aller seiner ehemaligen Unterzustände ausführen zu lassen. Seine Unterzustände sind nun Unterzustände der nächsthöheren Hierarchie.

F 16 (Einebnen des seriellen Hierarchiezustandes)

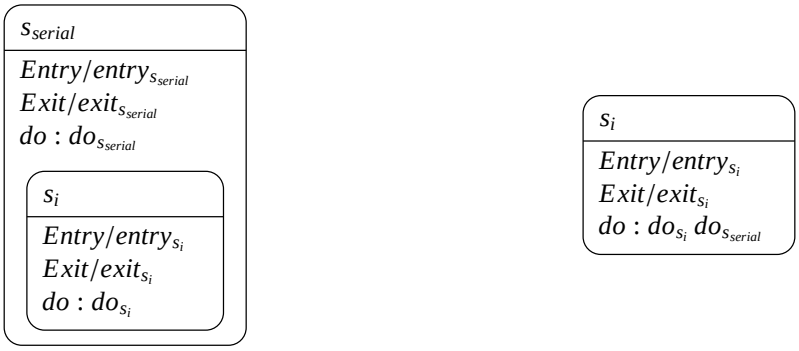


Abbildung 4.37: Flattening-Regel für das Einebnen des seriellen Hierarchiezustandes

$$\Leftrightarrow\ \{s_{serial}(s_i)\} \Longrightarrow \{s_i\},$$

$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_i), s_i \in \Sigma_{ss},\ do(s_i) = do(s_{serial}(s_i))\, do(s_{serial}),\ i = 1, \ldots,$
$|children(s_{serial})|,\ incoming(s_{serial}) = \emptyset,\ outgoing(s_{serial}) = \emptyset$ ◇

4.3.3 Sequentialisierung orthogonaler Hierarchiezustände

Nachdem nun alle seriellen Hierarchiezustände aus dem *Statechart* entfernt wurden, kann dazu übergegangen werden, die orthogonalen Zustände einzuebnen. Dabei wird in den Regeln T_{22} bis T_{34} zunächst genau wie beim *Flattening* vorgegangen, um die Transitionen von den orthogonalen Hierarchiezuständen zu entfernen. Daran anschließend wird mit den Regeln T_{35} bis T_{43} die Produktbildung orthogonaler Zustände mit dem letztendlichen Tilgen des orthogonalen Hierarchiezustandes in den Regeln T_{44} bis T_{48} ausgeführt. Ansätze der Produktbildung stammen aus [vdBMS, KHC+99, HW98].

In den folgenden Regeln zur Sequentialisierung wird, wie bei einer *Flattening*-Regel, ein Teil-*Statechart* behandelt, das mehrfach in einem *Statechart* vorhanden sein kann. Dies geschieht durch die Mehrfachanwendung der Regel, die durch die Steuerung in Abschnitt 4.3.5 beschrieben wird. Es werden immer gerade nur die orthogonalen Hierarchiezustände, die das niedrigste Hierarchieniveau besitzen, betrachtet. Nachdem nun dieses Hierarchieniveau durch Anwendung aller Regeln zur Sequentialisierung eingeebnet wurde, wird durch die wiederholte Anwendung der gesamten Sequentialisierung das nächsthöhere Niveau transformiert, usw. Dies geschieht solange, bis kein orthogonaler Hierarchiezustand mehr vorhanden ist.

Dem Leser wird die große Ähnlichkeit zwischen dem *Flattening* und der Einebnung der orthogonalen Zustände auffallen und es liegt die Vermutung nahe, dass ein *Flattening* nach der Produktbildung denselben Effekt auf *Statecharts* hätte wie die Sequentialisierung. Da die orthogonalen Zustände jedoch implizite Zustandüberführungen in Regionen und heraus enthalten, ist das *Flattening* hier nicht anwendbar und die Transformierung muss für orthogonale Zustände neu entworfen werden.

Zuerst soll nun in Regel T_{22} der Tatsache Rechnung getragen werden, dass Zustandsübergänge in höheren Regionen zuerst geprüft und ausgeführt werden.

T 22 (Konkurrierende Transitionen an Regionen)

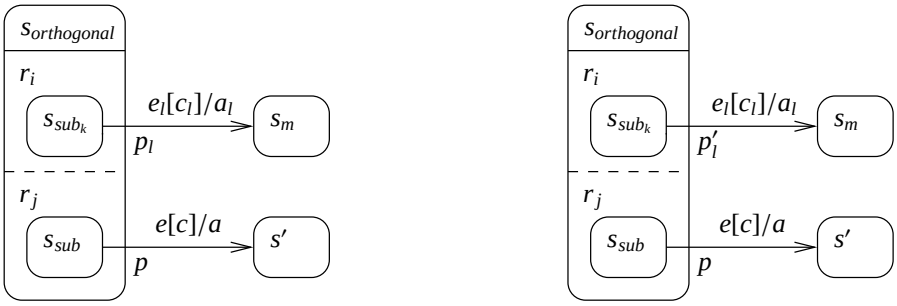


Abbildung 4.38: Transformationsregel für konkurrierende Transitionen an Regionen

$$\Leftrightarrow \{s_{orthogonal}[r_i(s_{sub_k})] \xrightarrow[p_l]{e_l[c_l]/a_l} s_m\}, s_{orthogonal}[r_j(s_{sub})] \xrightarrow[p]{e[c]/a} s'$$

$$\Longrightarrow$$

$$\{s_{orthogonal}[r_i(s_{sub_k})] \xrightarrow[p'_l]{e_l[c_l]/a_l} s_m\}, s_{orthogonal}[r_j(s_{sub})] \xrightarrow[p]{e[c]/a} s',$$

$$\text{wobei } p'_l = \overline{p}(s_{orthogonal}[r_j(s_{sub})], l_T, s') + p_l$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_i(s_{sub_k})], s_{orthogonal}[r_j(s_{sub})] \in \Sigma_{ss},\ s_m, s' \in children(s_{orthogonal}) \cup \complement children(s_{orthogonal}) \cup \Pi,\ \forall r_i \in \mathcal{R}(s_{orthogonal}) \setminus r_{|\mathcal{R}(s_{orthogonal})|},$

$$p(t = (s_{orthogonal}[r_j(s_{sub})], l_T, s')) = \max\{\bar{p}(t = (s_{orthogonal}[r_j(s_1)], l_T, s_2)) \mid s_2 \in \Sigma\},$$
$$l = 1, \ldots, \left|\bigcup_{\substack{t=(s_{orthogonal}[r_i(s)], l_T, s'),\\ s_{orthogonal}[r_i(s)] \in \Sigma_{ss},\\ s' \in \Sigma}} t\right|,\ i < j,\ k, m \leq l,\ p_l \leq p, \qquad \diamond$$

Analog zu *Flattening*-Regel F_1 sollen nun Transitionen, die von Zuständen in der untersten Hierarchiestufe ausgehen, die höchste Priorität im Vergleich mit Transitionen höherer Hierarchiestufen erhalten.

T 23 (Prioritäten am orthogonalen Hierarchiezustand)

Abbildung 4.39: Transformationsregel für Prioritäten am orthogonalen Hierarchiezustand

$$\Leftrightarrow \{s_{orthogonal}[r(s_{sub_m})] \xrightarrow[p_i]{e_i[c_i]/a_i} s_n\} \Longrightarrow \{s_{orthogonal}[r(s_{sub_m})] \xrightarrow[p'_i]{e_i[c_i]/a_i} s_n\},$$
$$\text{wobei } p'_i = \bar{p}(outgoing(s_{orthogonal})) + p_i,$$
$$EM_l : \forall r = \mathcal{R}(s_{orthogonal}),\ s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub_m})] \in \Sigma_{ss},\ m, n \leq i,$$
$$s_n \in children(s_{orthogonal}) \cup \complement children(s_{orthogonal}),\ i = 1, \ldots, \left|\bigcup_{\substack{t=(s_{orthogonal}[r(s)], l_T, s'),\\ s_{orthogonal}[r(s)] \in \Sigma_{ss},\\ s' \in \Sigma}} t\right| \qquad \diamond$$

Ebenso wie beim *Flattening* werden nun systematisch alle in und an einem orthogonalen Hierarchiezustand existierenden Transitionen betrachtet. Dabei werden sowohl implizite Zustandsaktivierungen und -deaktivierungen als auch die Möglichkeit der Geschichte einer jeden Region berücksichtigt. Es ist zu bemerken, dass eine Regel immer die Gesamtheit aller möglichen Regionen und deren Unterzustände betrachtet. Oftmals sind orthogonale Zustände jedoch einfacher komponiert, deshalb werden Regionen und Zustände, die nur der Vollständigkeit dienen, unter dem Punkt *optional* aufgeführt (wie z. B. in Regel T_{27}). Wird demnach einen optionaler Aspekt vernachlässigt, müssen auch die dazugehörigen Transitionen vernachlässigt werden.

T 24 (Intraregionale Transition des Typs (s_{sub}, l_T, s_{sub}))

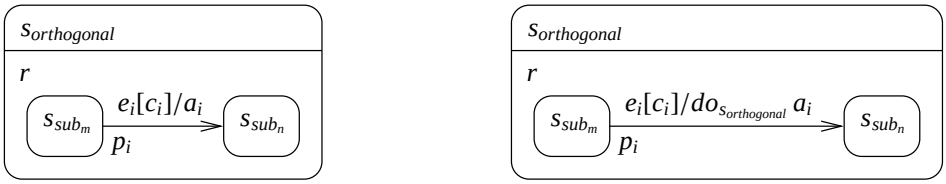


Abbildung 4.40: Transformationsregel für intraregionale Transitionen des Typs (s_{sub}, l_T, s_{sub})

$$\Leftrightarrow \{s_{orthogonal}[r(s_{sub_m})] \xrightarrow[p_i]{e_i[c_i]/a_i} s_{orthogonal}[r(s_{sub_n})]\}$$

$$\Longrightarrow$$

$$\{s_{orthogonal}[r(s_{sub_m})] \xrightarrow[p_i]{e_i[c_i]/do_{s_{orthogonal}}\, a_i} s_{orthogonal}[r(s_{sub_n})]\},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub_m})],\ s_{orthogonal}[r(s_{sub_n})] \in \Sigma_{ss},$
$\forall r \in \mathcal{R}(s_{orthogonal})$ mit $s_{orthogonal}[r(s_{sub_m})] \wedge s_{orthogonal}[r(s_{sub_n})],\ m, n \leq i,\ i = 1, \ldots,$
$\left| \bigcup_{\substack{t=(s_{orthogonal}[r(s)], l_T, s_{orthogonal}[r(s')]),\\ s_{orthogonal}[r(s)], s_{orthogonal}[r(s')] \in \Sigma_{ss}}} t \right|$ ◇

T 25 (Interregionale Transition des Typs (s_{sub}, l_T, s_{sub}), ohne Geschichte)

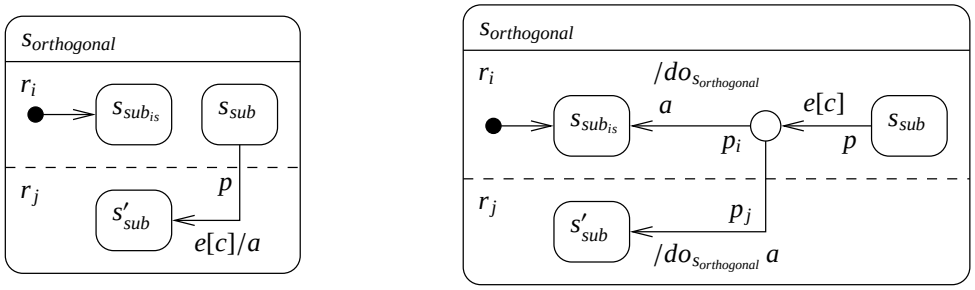


Abbildung 4.41: Transformationsregel für interregionale Transitionen des Typs $(s_{sub}, l_T s_{sub})$, ohne Geschichte

$$\Leftrightarrow s_{orthogonal}[r_i(s_{sub_{is}})], s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]/a} s_{orthogonal}[r_j(s'_{sub})]$$

$$\Longrightarrow$$

$$s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]} s_{orthogonal}[r_i(c_{junction})],$$

$$s_{orthogonal}[r_i(s_{sub_{is}})] \xleftarrow[p_1]{/do_{s_{orthogonal}}\, a} s_{orthogonal}[r_i(c_{junction})] \xrightarrow[p_2]{/do_{s_{orthogonal}}\, a} s_{orthogonal}[r_j(s_{sub})],$$

wobei $p_i = i$ und $p_j = j$,

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ \nexists s_{orthogonal}[r_i(c_{history})],\ s_{orthogonal}[r_i(s_{sub_{is}})] \in \Sigma_{ss} \cap \Sigma_{is},$
$s_{orthogonal}[r_i(s_{sub})], s_{orthogonal}[r_j(s_{sub})] \in \Sigma_{ss},\ \forall r_i \in \mathcal{R}(s_{orthogonal})$ mit $t = (s_{orthogonal}[r_i(s_{sub})],$
$l_T, s_{orthogonal}[r_j(s_{sub})]),\ i \neq j$ ◇

T 26 (Interregionale Transition des Typs (s_{sub}, l_T, s_{sub}), mit Geschichte)

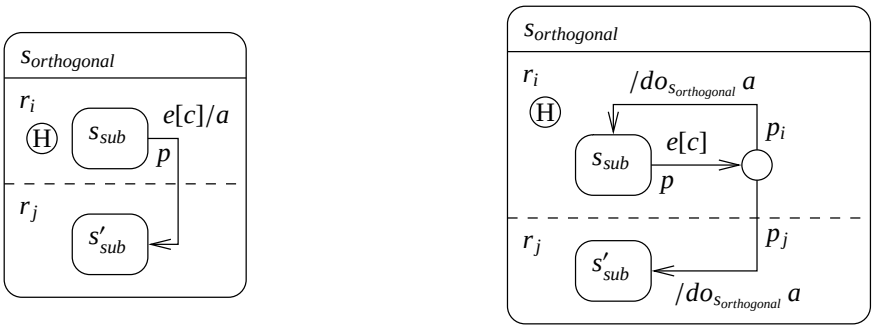


Abbildung 4.42: Transformationsregel für interregionale Transitionen des Typs (s_{sub}, l_T, s_{sub})*, mit Geschichte*

$$\Leftrightarrow\ s_{orthogonal}[r_i(c_{history})], s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]/a} s_{orthogonal}[r_j(s'_{sub})]$$

$$\Longrightarrow$$

$$s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]} s_{orthogonal}[r_i(c_{junction})],$$

$$s_{orthogonal}[r_i(s_{sub})] \xleftarrow[p_1]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_i(c_{junction})] \xrightarrow[p_2]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_j(s'_{sub})],$$

$$\text{wobei } p_i = i \text{ und } p_j = j,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_i(s_{sub})],\ s_{orthogonal}[r_j(s'_{sub})] \in \Sigma_{ss},$
$\forall r_i \in \mathcal{R}(s_{orthogonal})$ mit $t = (s_{orthogonal}[r_i(s_{sub})], l_T, s_{orthogonal}[r_j(s'_{sub})]),\ i \neq j$ ◇

T 27 (Transition des Typs (s_{sub}, l_T, $s_{orthogonal}$), ohne Geschichte)

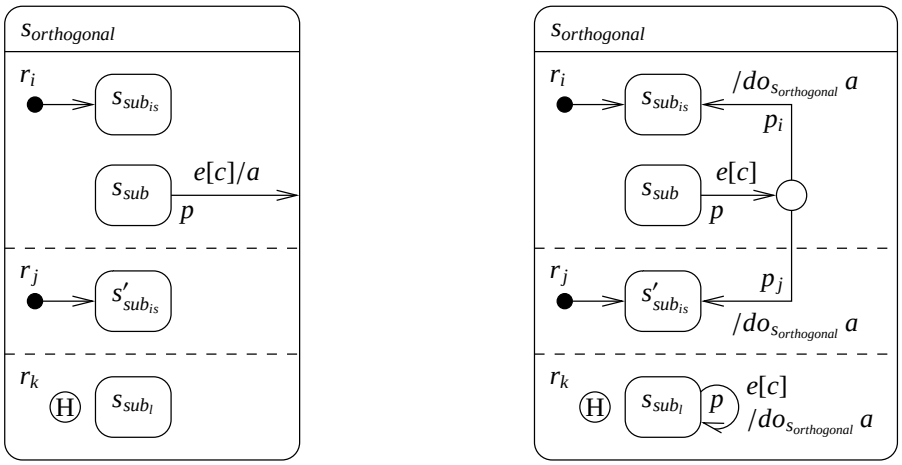


Abbildung 4.43: Transformationsregel für Transitionen des Typs $(s_{sub}, l_T, s_{orthogonal})$*, ohne Geschichte*

$$\Leftrightarrow\ s_{orthogonal}[r_i(s_{sub_{is}})],\ s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]/a} s_{orthogonal},\ \{s_{orthogonal}[r_j(s'_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\},\ \{s_{orthogonal}[r_k(s_{sub_l})]\}$$
$$\Longrightarrow$$
$$s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]} s_{orthogonal}[r_i(c_{junction})],$$
$$s_{orthogonal}[r_i(s_{sub_{is}})] \xleftarrow[p_i]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_i(c_{junction})]\{\xrightarrow[p_j]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_j(s'_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\}, s_{orthogonal}[r_k(s_{sub_l})] \xrightarrow[p]{e[c]/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_k(s_{sub_l})],$$
$$\text{wobei } p_i = i \text{ und } p_j = j,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_i(s_{sub_{is}})], s_{orthogonal}[r_j(s'_{sub_{is}})] \in \Sigma_{ss} \cap \Sigma_{is},$
$\nexists s_{orthogonal}[r_{\{i,j\}}(c_{history})],\ s_{orthogonal}[r_i(s_{sub})],\ s_{orthogonal}[r_k(s_{sub_l})] \in \Sigma_{ss},\ i \neq j \neq k,$
optional: $children^*(s_{orthogonal})[r_j], children^*(s_{orthogonal})[r_k]$ ◇

T 28 (Transition des Typs (s_{sub}, l_T, $s_{orthogonal}$), mit Geschichte)

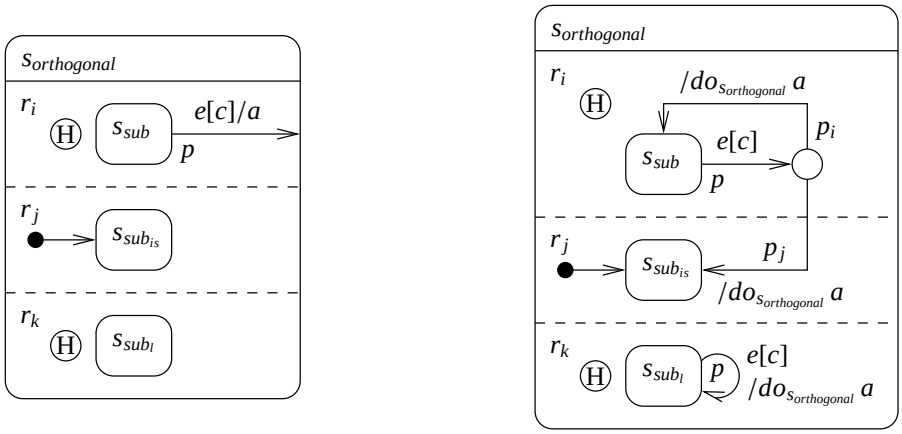


Abbildung 4.44: Transformationsregel für Transitionen des Typs (s_{sub}, l_T, $s_{orthogonal}$), mit Geschichte

$$\Leftrightarrow\ s_{orthogonal}[r_i(c_{history})], s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]/a} s_{orthogonal},$$
$$\{s_{orthogonal}[r_j(s_{sub_{is}})]\}, \{s_{orthogonal}[r_k(c_{history})]\}, \{s_{orthogonal}[r_k(s_{sub_l})]\}$$
$$\Longrightarrow$$
$$s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]} s_{orthogonal}[r_i(c_{junction})],$$
$$s_{orthogonal}[r_i(s_{sub})] \xleftarrow[p_i]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_i(c_{junction})]\{\xleftarrow[p_j]{/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\}, s_{orthogonal}[r_k(s_{sub_l})] \xrightarrow[p]{e[c]/do_{s_{orthogonal}}\ a} s_{orthogonal}[r_k(s_{sub_l})],$$
$$s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_j(s_{sub_{is}})] \in \Sigma_{ss} \cap \Sigma_{is},$$
$$\text{wobei } p_i = i \text{ und } p_j = j,$$

$EM_l : \nexists s_{orthogonal}[r_j(c_{history})],\ s_{orthogonal}[r_i(s_{sub})], s_{orthogonal}[r_k(s_{sub_l})] \in \Sigma_{ss},\ i \neq j \neq k,$
optional: $children^*(s_{orthogonal})[r_j],\ children^*(s_{orthogonal})[r_k]$ ◇

T 29 (Transition des Typs (s_{sub}, l_T, $s_{outside}$))

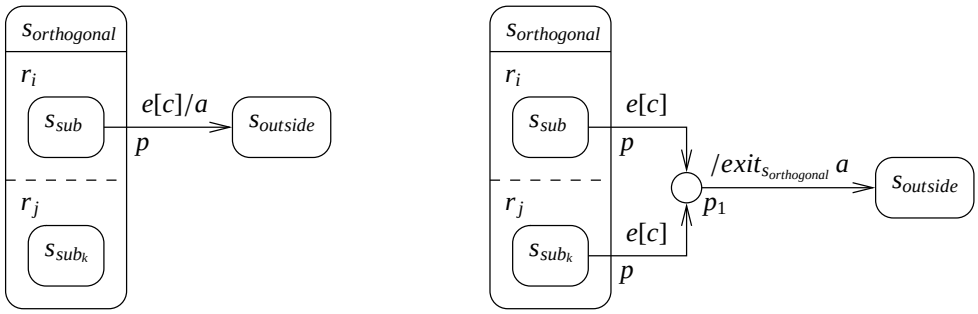


Abbildung 4.45: Transformationsregel für Transitionen des Typs ($s_{sub}, l_T, s_{outside}$)

$$\Leftrightarrow\ s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]/a} s_{outside}, \{s_{orthogonal}[r_j(s_{sub_k})]\}$$
$$\Longrightarrow$$
$$s_{orthogonal}[r_i(s_{sub})] \xrightarrow[p]{e[c]} c_{junction}\{\xleftarrow[p]{e[c]} s_{orthogonal}[r_j(s_{sub_k})]\}, c_{junction} \xrightarrow[p_1]{/exit_{s_{orthogonal}}\ a} s_{outside}\},$$
$$\text{wobei } p_1 = 1,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_i(s_{sub})],\ s_{orthogonal}[r_j(s_{sub_k})] \in \Sigma_{ss},\ i \neq j$ ◇

T 30 (Transition des Typs ($s_{orthogonal}$, l_T, s_{sub}))

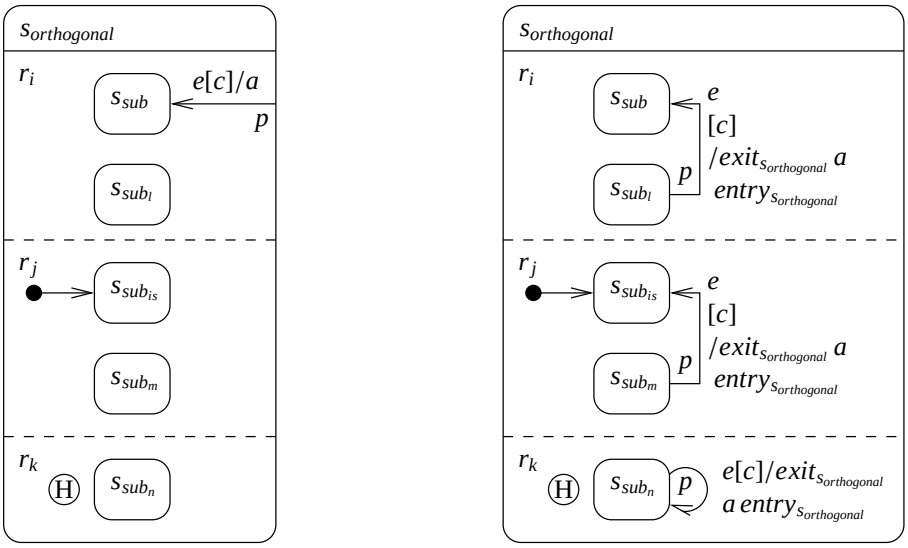


Abbildung 4.46: Transformationsregel für Transitionen des Typs $(s_{orthogonal}, l_T, s_{sub})$

$$\Leftrightarrow\ s_{orthogonal} \xrightarrow[p]{e[c]/a} s_{orthogonal}[r_i(s_{sub})], \{s_{orthogonal}[r_i(s_{sub_l})]\},$$
$$s_{orthogonal}[r_j(s_{sub_{is}})], \{s_{orthogonal}[r_j(s_{sub_m})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\}, \{s_{orthogonal}[r_k(s_{sub_n})]\}$$
$$\Longrightarrow$$
$$\{s_{orthogonal}[r_i(s_{sub_l})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\ a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_i(s_{sub})]\},$$
$$\{s_{orthogonal}[r_j(s_{sub_m})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\ a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(s_{sub_n})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\ a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_k(s_{sub_n})]\},\ \{s_{orthogonal}[r_k(c_{history})]\},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_{\{i,j,k\}}(s_{sub_{\{l,is,m,n\}}})] \in \Sigma_{ss},\ s_{orthogonal}[r_j(s_{sub_{is}})] \in \Sigma_{is},$
$\nexists s_{orthogonal}[r_j(c_{history})],\ i \neq j \neq k$ ◇

T 31 (Transition des Typs ($s_{orthogonal}$, l_T, $s_{orthogonal}$))

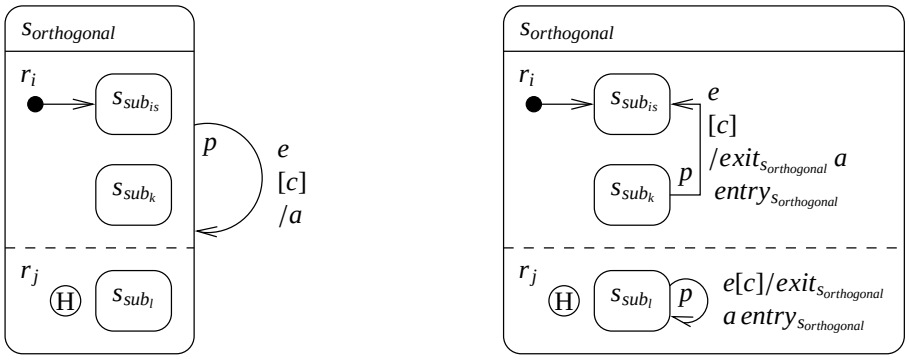


Abbildung 4.47: Transformationsregel für Transitionen des Typs ($s_{orthogonal}$, l_T, $s_{orthogonal}$)

$$\Leftrightarrow \; s_{orthogonal} \xrightarrow[p]{e[c]/a} s_{orthogonal},$$

$$s_{orthogonal}[r_i(s_{sub_{is}})], \{s_{orthogonal}[r_i(s_{sub_k})]\}, \{s_{orthogonal}[r_j(c_{history})]\}, \{s_{orthogonal}[r_j(s_{sub_l})]\}$$

$$\Longrightarrow$$

$$\{s_{orthogonal}[r_i(s_{sub_k})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\, a\, entry_{s_{orthogonal}}} \} s_{orthogonal}[r_i(s_{sub_{is}})]\},$$

$$\{s_{orthogonal}[r_j(c_{history})]\}, \{s_{orthogonal}[r_j(s_{sub_l})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\, a\, entry_{s_{orthogonal}}} \} \{s_{orthogonal}[r_j(s_{sub_l})]\},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_{\{i,j\}}(s_{sub_{\{is,k,l\}}})] \in \Sigma_{ss},\ s_{orthogonal}[r_i(s_{sub_{is}})] \in \Sigma_{is},$
$\nexists s_{orthogonal}[r_i(c_{history})],\ i \neq j$ ◇

T 32 (Transition des Typs ($s_{orthogonal}$, l_T, $s_{outside}$))

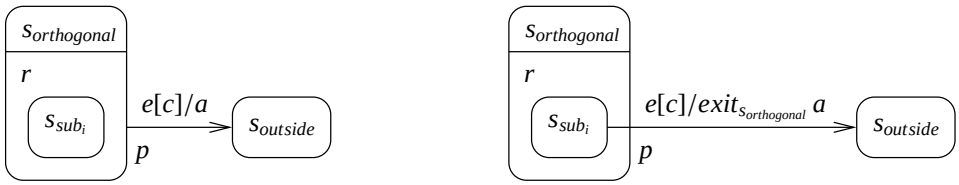


Abbildung 4.48: Transformationsregel für Transitionen des Typs ($s_{orthogonal}$, l_T, $s_{outside}$)

$$\Leftrightarrow \; s_{orthogonal} \xrightarrow[p]{e[c]/a} s_{outside}, \{s_{orthogonal}[r(s_{sub_i})]\}$$

$$\Longrightarrow$$

$$\{s_{orthogonal}[r(s_{sub_i})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}}\, a} \} s_{outside}\},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub_i})] \in \Sigma_{ss},\ i = 1, \ldots, |children(s_{orthogonal})|,$
$r = \mathcal{R}(s_{orthogonal})$ ◇

T 33 (Transition des Typs ($s_{outside}$, l_T, s_{sub}))

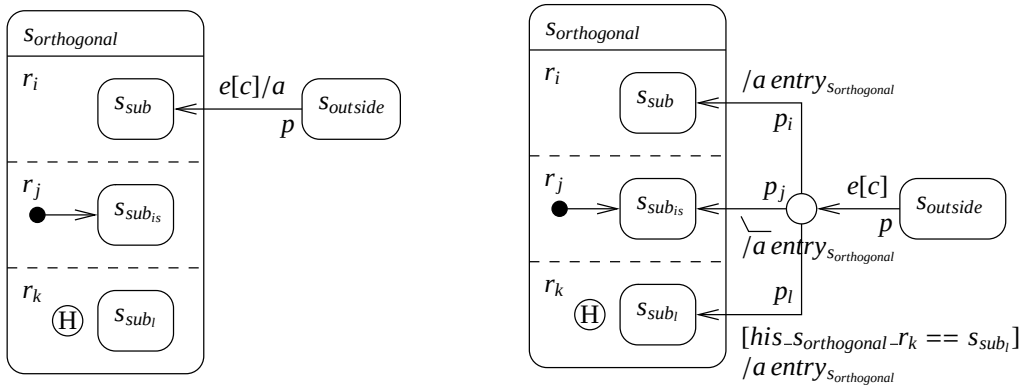


Abbildung 4.49: Transformationsregel für Transitionen des Typs $(s_{outside}, l_T, s_{sub})$

$$\Leftrightarrow\ s_{outside} \xrightarrow[p]{e[c]/a} s_{orthogonal}[r_i(s_{sub})], \{s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_k(c_{history})]\}, \{s_{orthogonal}[r_k(s_{sub_l})]\}$$
$$\Longrightarrow$$
$$s_{outside} \xrightarrow[p]{e[c]} c_{junction}, \{s_{orthogonal}[r_k(c_{history})]\},$$
$$c_{junction} \xrightarrow[p_i]{/a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_i(s_{sub})], c_{junction}\{\xrightarrow[p_j]{/a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_j(s_{sub_{is}})]\},$$
$$c_{junction}\{\xrightarrow[p_l]{[his\_s_{orthogonal}\_r_k == s_{sub_l}]/a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_k(s_{sub_l})]\},$$
$$\text{wobei } p_i = i,\ p_j = j \text{ und } p_l = |\mathcal{R}(s_{orthogonal})| + l,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ \nexists s_{orthogonal}[r_j(c_{history})],\ s_{orthogonal}[r_{\{i,j,k\}}(s_{sub_{\{is,l\}}})] \in \Sigma_{ss},$
$s_{orthogonal}[r_j(s_{sub_{is}})] \in \Sigma_{is},\ i \neq j \neq k,\ l = 1, \ldots, |children^*(\mathcal{R}_{his}(s_{orthogonal})|$ ◇

T 34 (Transition des Typs ($s_{outside}$, l_T, $s_{orthogonal}$))

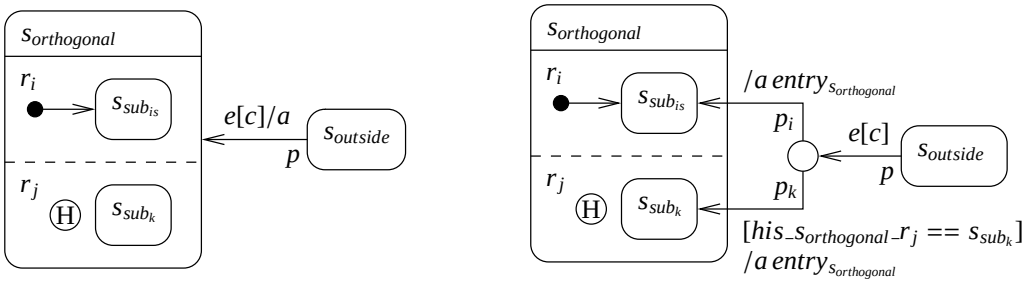


Abbildung 4.50: Transformationsregel für Transitionen des Typs ($s_{outside}, l_T, s_{orthogonal}$)

$$\Leftrightarrow\ s_{outside} \xrightarrow[p]{e[c]/a} s_{orthogonal}, \{s_{orthogonal}[r_i(s_{sub_{is}})]\},$$
$$\{s_{orthogonal}[r_j(c_{history})]\}, \{s_{orthogonal}[r_j(s_{sub_k})]\}$$
$$\Longrightarrow$$
$$\{s_{orthogonal}[r_j(c_{history})]\},\ s_{outside} \xrightarrow[p]{e[c]} c_{junction} \xrightarrow[p_i]{/a\ entry_{s_{orthogonal}}} \{s_{orthogonal}[r_i(s_{sub_{is}})],$$
$$c_{junction}\{\xrightarrow[p_k]{[his\_s_{orthogonal}\_r_j == s_{sub_k}]/a\ entry_{s_{orthogonal}}} s_{orthogonal}[r_j(s_{sub_k})]\},$$
$$\text{wobei } p_i = i \text{ und } p_k = |\mathcal{R}(s_{orthogonal})| + k,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_{\{i,j\}}(s_{sub_{\{is,k\}}})] \in \Sigma_{ss},\ s_{orthogonal}[r_i(s_{sub_{is}})] \in \Sigma_{is},$
$k = 1, \ldots, |children^*(\mathcal{R}_{his}(s_{orthogonal}))|,\ i \neq j$ ◇

Nachdem nun alle Transitionen vom orthogonalen Zustand abgelöst und neu angeordnet sind, kann die Produktbildung von Zuständen erfolgen. Vorbereitend darauf seien die Regeln T_{35} bis T_{38} aufgeführt. Diese sorgen dafür, dass die Aktivitäten des orthogonalen Zustandes und seiner Unterzustände auch nach der Sequentialisierung zum richtigen Zeitpunkt ausgeführt werden. Ein- und Austrittsaktivitäten der Zustände werden dabei, wie gezeigt, in die Aktionen von Transitionen verschoben.

T 35 (Aktivitäten im orthogonalen Hierarchiezustand I)

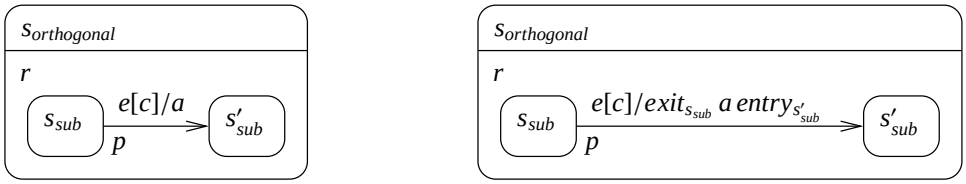


Abbildung 4.51: Transformationsregel für Aktivitäten im orthogonalen Hierarchiezustand I

$$\Leftrightarrow\ s_{orthogonal}[r(s_{sub})] \xrightarrow[p]{e[c]/a} s_{orthogonal}[r(s'_{sub})]$$

$$\Longrightarrow$$

$$s_{orthogonal}[r(s_{sub})] \xrightarrow[p]{e[c],/exit_{s_{orthogonal}[r(s_{sub})]}\ a\ entry_{s_{orthogonal}[r(s'_{sub})]}} s_{orthogonal}[r(s'_{sub})],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub})], s_{orthogonal}[r(s'_{sub})] \in \Sigma_{ss},\ r = \mathcal{R}(s_{orthogonal}),$
$incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

T 36 (Aktivitäten im orthogonalen Hierarchiezustand II)

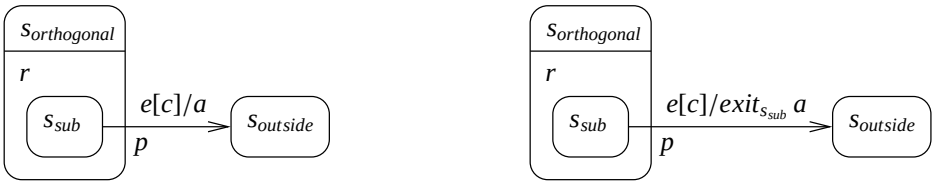


Abbildung 4.52: Transformationsregel für Aktivitäten im orthogonalen Hierarchiezustand II

$$\Leftrightarrow\ s_{orthogonal}[r(s_{sub})] \xrightarrow[p]{e[c]/a} s_{outside} \Longrightarrow s_{orthogonal}[r(s_{sub})] \xrightarrow[p]{e[c]/exit_{s_{orthogonal}[r(s_{sub})]}\ a} s_{outside},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss},\ r = \mathcal{R}(s_{orthogonal}),$
$incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

T 37 (Aktivitäten im orthogonalen Hierarchiezustand III)

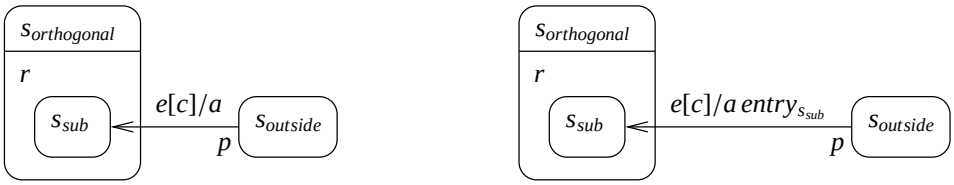


Abbildung 4.53: Transformationsregel für Aktivitäten im orthogonalen Hierarchiezustand III

$$\Leftrightarrow\ s_{outside} \xrightarrow[p]{e[c]/a} s_{orthogonal}[r(s_{sub})] \Longrightarrow s_{outside} \xrightarrow[p]{e[c]/a\ entry_{s_{orthogonal}[r(s_{sub})]}} s_{orthogonal}[r(s_{sub})],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss},\ r = \mathcal{R}(s_{orthogonal}),$
$incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

T 38 (Aktivitäten im orthogonalen Hierarchiezustand IV)

Abbildung 4.54: Transformationsregel für Aktivitäten im orthogonalen Hierarchiezustand IV

$$\Leftrightarrow \; entry(s_{orthogonal}[r(s_{sub})]) = entry_{s_{sub}}, exit(s_{orthogonal}[r(s_{sub})]) = exit_{s_{sub}}$$
$$\Longrightarrow$$
$$entry(s_{orthogonal}[r(s_{sub})]) = \varepsilon, exit(s_{orthogonal}[r(s_{sub})]) = \varepsilon,$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss},\ incoming(s_{orthogonal}) = \emptyset,$
$outgoing(s_{orthogonal}) = \emptyset$ ◇

Produktbildung

Aufbauend auf den eben beschriebenen kann nun die Produktbildung begonnen werden. Mit ihr wird ein Unterzustand einer Region mit jeweils einem Unterzustand aus allen anderen Regionen verknüpft. Geschieht das für alle Unterzustände, erhält man alle Bereiche aktiver Unterzustände eines orthogonalen Zustandes, d. h. zu einem bestimmten Zeitpunkt beim Ablauf eines *Statecharts* ist die Menge aller Zustände eines Bereiches aktiv. Zu diesem Zeitpunkt befindet sich ein *Statechart* also in einem bestimmten Metazustand, der für einen solchen Bereich steht. Ziel der Produktbildung ist es nun, genau diese Metazustände zu finden und auf neue Zustände des *Statecharts* abzubilden.

Es sei darauf hingewiesen, dass die Produktbildung streng nach den Regeln für die Abarbeitung von Transitionen unter *Artisan RtS* entworfen wurde. Wie bereits in Abschnitt 2.2.2 dokumentiert wurde, können dort niemals zwei Transitionen in verschiedenen Regionen gleichzeitig schalten, auch wenn sie zur gleichen Zeit ausgelöst wurden, sondern werden immer in der Reihenfolge der Regionen von oben nach unten abgearbeitet; dabei wird ein Zustandsübergang immer mit dem Betreten des neuen Zustandes beendet.

Als Hilfe werden Anleihen bei der Graphfärbung genommen. Gefärbte Zustände dienen in den folgenden Regeln der Kennzeichnung noch zu besuchender Zustände. Für die formale Umsetzung gefärbter Zustände sei der Leser auf Definition 44 verwiesen. Eröffnend werden in Regel T_{39} alle Zustände des orthogonales Zustandes als noch nicht besucht gefärbt.

T 39 (Färben)

Abbildung 4.55: Transformationsregel zum Färben von Zuständen

$$\Leftrightarrow \; s_{orthogonal}[r(s_{sub})] \Longrightarrow s_{orthogonal}[r(\phi(s_{sub}))],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss},\ r = \mathcal{R}(s_{orthogonal}),$
$|\mathcal{R}(s_{orthogonal})| > 1,\ incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

Beginnend bei der untersten Region r_n, wobei $n = |\mathcal{R}(s_{orthogonal})|$, wird anschließend ein gefärbter Zustand $\phi(s_{sub})$ der untersten Region mit allen gefärbten Zuständen $\phi(s_{sub_i})$ der darüberliegenden Region r_{n-1} zu einem neuen Zustand $s_{sub_i} \times s_{sub}$ in der Region r_{n-1} verknüpft. Der neue Zustand erhält beide statischen Aktionen, wobei die des gefärbten Zustandes $\phi(s_{sub_i})$ zuerst ausgeführt wird. Gleichzeitig wird der gefärbte Zustand $\phi(s_{sub_i})$ der Region r_{n-1} in der gleichen Region dupliziert und der gefärbte Zustand $\phi(s_{sub})$ der Region r_n gelöscht. Das Duplikat wird dann benutzt, um alle weiteren Zustände aus der Region r_n wie oben zu behandeln.

T 40 (Produktbildung)

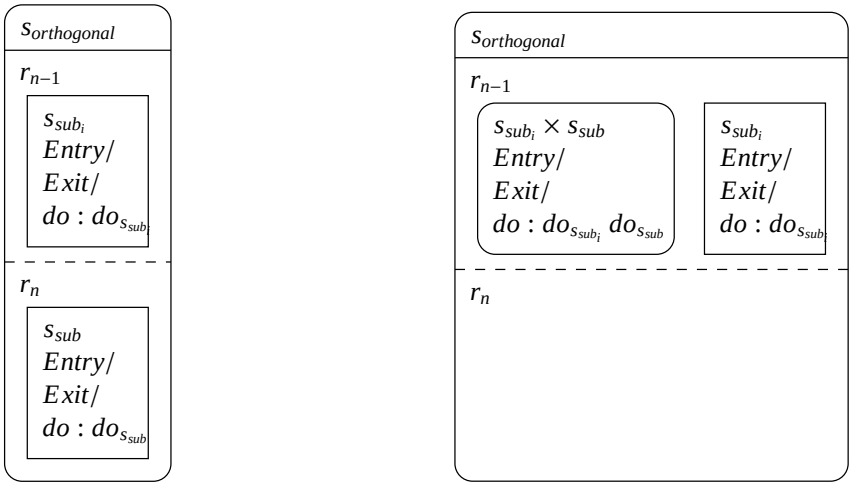


Abbildung 4.56: Transformationsregel zur Produktbildung

$$\Leftrightarrow \{X[r_{n-1}(\phi(s_{sub_i}))]\}, X[r_n(\phi(s_{sub}))]$$
$$\Longrightarrow$$
$$\{X[r_{n-1}((s_{sub_i} \times s_{sub}))]\}, X[r_n(\phi(s_{sub}))],$$

EM_l : Zugunsten der Übersichtlichkeit fand eine Textersetzung statt: $X = s_{orthogonal}$:
$X \in \Sigma_{orthogonal},\ n = |\mathcal{R}(X)|,\ |n| > 1,\ children(X[r_{n-1}]) \neq \emptyset,$
$incoming(X[r_{n-1}(s_{sub_i} \times s_{sub})]) = incoming(X[r_{n-1}(\phi(s_{sub_i}))]) \cup incoming(X[r_n(\phi(s_{sub}))]),$
$outgoing(X[r_{n-1}(s_{sub_i} \times s_{sub})]) = outgoing(X[r_{n-1}(\phi(s_{sub_i}))]) \cup outgoing(X[r_n(\phi(s_{sub}))]),$
$$X[r_{n-1}(s_{sub_i} \times s_{sub})] :\in \begin{cases} \Sigma_{is} & \text{falls } \exists t_i = (X[r_{n-1}(c_{initial})], l_{T_i}, X[r_{n-1}(\phi(s_{sub_i}))]) \wedge \\ & \exists t = (X[r_n(c_{initial})], l_T, X[r_n(\phi(s_{sub}))]), \\ \complement\Sigma_{is} & sonst \end{cases}$$
$do(X[r_{n-1}(s_{sub_i} \times s_{sub})]) = do(X[r_{n-1}(s_{sub_i})])\, do(X[r_n(s_{sub})]),\ entry(X[r_{n-1}(s_{sub_i} \times s_{sub})]) =$
$exit(X[r_{n-1}(s_{sub_i} \times s_{sub})]) = \varepsilon,\ incoming(X) = outgoing(X) = \emptyset$ ◇

Sind alle gefärbten Zustände aus der Region $r_n, n = |\mathcal{R}(s_{orthogonal})|$, behandelt und gelöscht, und sind dort evtl. noch Pseudozustände der Art $c_{junction}$ vorhanden, so werden diese in die Region r_{n-1} verschoben und zugleich gefärbte Zustände der Region r_{n-1} gelöscht. Alle nun vorhandenen Zustände in Region r_{n-1} werden mit ϕ gefärbt und die Region r_n mit den darin evtl. noch vorhandenen Pseudozuständen gelöscht. Mit der erneuten Färbung ist die Voraussetzung für die erneute Anwendung der Regel T_{40} gegeben.

T 41 (Tilgen einer Region)

Abbildung 4.57: Transformationsregel zum Tilgen einer Region

$$\Leftrightarrow \{s_{orthogonal}[r_{n-1}(s_{sub_i})]\}, \{s_{orthogonal}[r_{n-1}(\phi(s_{sub_j}))]\}, \{s_{orthogonal}[r_n(c_{junction})\}$$
$$\Longrightarrow$$
$$\{s_{orthogonal}[r_{n-1}(\phi(s_{sub_i}))]\}, \{s_{orthogonal}[r_{n-1}(c_{junction})\},$$

EM_l : $s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_{n-1}(s_{sub_i})] \in \Sigma_{ss},\ \nexists s_{orthogonal}[r_n(s)] \in \Sigma \cup \Phi,$
$n = |\mathcal{R}(s_{orthogonal})|,\ |n| > 1,\ incoming(s_{orthogonal}) = outgoing(s_{orthogonal}) = \emptyset$
optional: $c_{junction}$ ◇

Ist nun nur noch eine Region vorhanden, d. h. $|\mathcal{R}(s_{orthogonal}| = 1$, ist die Voraussetzung für die Anwendung der Regeln T_{40} und T_{41} nicht mehr gegeben. Die Produktbildung wird dann mit dem Löschen noch vorhandender gefärbter Zustände $\phi(s_{sub})$ in der einzig verbliebenen Region r_1 beendet.

T 42 (Entfärben)

Abbildung 4.58: Transformationsregel zum Entfärben von Zuständen

$$\Leftrightarrow \{s_{orthogonal}[r_1(\phi(s_{sub}))]\} \Longrightarrow s_{orthogonal}[r_1(s_{sub})],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal},\ s_{orthogonal}[r_1(s_{sub})] \in \Sigma_{ss},\ |\mathcal{R}(s_{orthogonal})| = 1,$
$incoming(s_{orthogonal}) = outgoing(s_{orthogonal}) = \emptyset$ ◇

Regel T_{43} stellt eine Ausnahmeregelung für die Produktbildung nach den Regeln T_{39} bis T_{42} dar. Ein Synchronisationszustand teilt die Menge der Zustände in verschiedene Bereiche aktiver Zustände. Dazu soll zunächst die Mengenschreibweise von Zuständen in den Regionen definiert werden: Die Deklaration $s_i'\bullet$ bedeutet die Menge der Zustände der Region r_k, die der zum Synchronisationszustand gehörigen *Fork*-Verknüpfung nachgestellt sind, und $\bullet s_j$ die Menge der Zustände der Region r_l, die der zum Synchronisationszustand gehörigen *Join*-Verknüpfung vorangestellt sind. Die Vor- oder Nachstellung von Zuständen betrifft die Zeit bei der Abarbeitung von *Statecharts*.

T 43 (Produktbildung bei einem Synchronisationszustand)

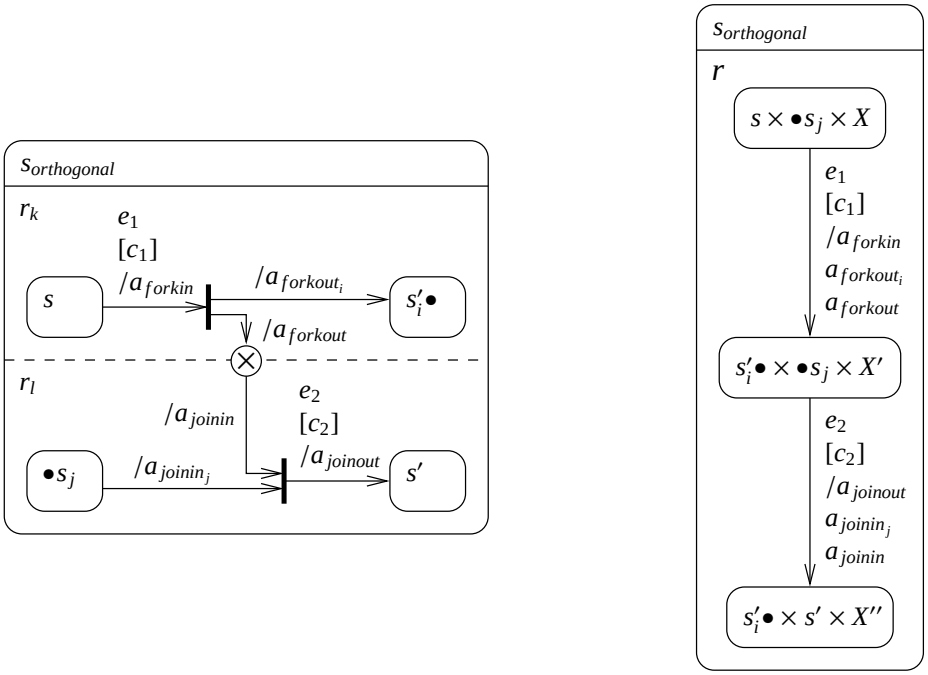


Abbildung 4.59: Transformationsregel zur Produktbildung bei einem Synchronisationszustand

$$\Leftrightarrow\ s_{orthogonal}[r_k(s)] \xrightarrow{e_1[c_1]/a_{forkin}} s_{orthogonal}[r_k(c_{fork})]\{\xrightarrow{/a_{forkout_i}} s_{orthogonal}[r_k(s'_i\bullet)]\},$$

$$s_{orthogonal}[r_k(c_{fork})] \xrightarrow{/a_{forkout}} c_{synchstate} \xrightarrow{/a_{joinin}} s_{orthogonal}[r_l(c_{join})],$$

$$\{s_{orthogonal}[r_l(\bullet s_j)] \xrightarrow{/a_{joinin_j}}\} s_{orthogonal}[r_l(c_{join})] \xrightarrow{e_2[c_2]/a_{joinout}} s_{orthogonal}[r_l(s')],$$

$$\Longrightarrow$$

$$s_{orthogonal}[r((s \times \bullet s_j \times X))] \xrightarrow{e_1[c_1]/a_{forkin}\, a_{forkout_i}\, a_{forkout}} s_{orthogonal}[r((s'_i\bullet \times \bullet s_j \times X'))]$$

$$s_{orthogonal}[r((s'_i\bullet \times \bullet s_j \times X'))] \xrightarrow{e_2[c_2]/a_{joinout}\, a_{joinin_i}\, a_{joinin}} s_{orthogonal}[r((s'_i\bullet \times s' \times X''))],$$

$EM_l : X, X', X'' = \Xi(\mathcal{R}(s_{orthogonal}) \setminus \{r_k, r_l\}),\ k < l,$
$incoming(s_{orthogonal}) = outgoing(s_{orthogonal}) = \emptyset$ ◇

Am Synchronisationszustand können nun in den Regionen r_k und r_l drei Arten von Bereichen aktiver Zustände festgestellt werden: Bereiche aktiver Zustände,

1. die sich in r_k und r_l vor,
2. die sich in r_k nach und in r_l vor und

3. die sich in r_k und r_l nach

der Synchronisationskonstruktion befinden. Die Abfolge der Bereiche aktiver Zustände ist nur in dieser Reihenfolge möglich. Für jeden dieser Bereiche ist in der rechten Seite der Produktion ein neuer Zustand aufgeführt, der für Mengen von Zuständen steht. X, X' bzw. X'' stehen dabei für Kombinationen von Zuständen über Regionen außer r_k und r_l, die zum gleichen Zeitpunkt wie die in der Regel erwähnten aktiv sind. Die Übergangsaktionen werden in der Reihenfolge der Ausführung um den Synchronisationzustand in den Aktionsmarkierungen der neuen Transitionen konkateniert.

Nachdem nun die Produktbildung abgeschlossen ist, wird nun mit den Regeln T_{44} und T_{45} wie beim *Flattening* festgestellt, ob und welcher Unterzustand ein Startzustand sein kann.

T 44 (Startverknüpfung nach Sequentialisierung, $s_{orthogonal} \notin \Sigma_{is}$)

Abbildung 4.60: Transformationsregel für die Startverknüpfung nach Sequentialisierung, $s_{orthogonal} \notin \Sigma_{his}$

$$\Leftrightarrow\ s_{orthogonal}[r(c_{initial})] \rightarrow s_{orthogonal}[r(s_{sub})] \Longrightarrow s_{orthogonal}[r(s_{sub})],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal} \setminus \Sigma_{is},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss},\ r = \mathcal{R}(s_{orthogonal}),$
$incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

T 45 (Startverknüpfung, $s_{orthogonal} \in \Sigma_{is}$)

Abbildung 4.61: Transformationsregel für die Startverknüpfung, $s_{orthogonal} \in \Sigma_{his}$

$$\Leftrightarrow\ c_{initial} \rightarrow s_{orthogonal}[r(s_{sub})] \Longrightarrow s_{orthogonal}[r(s_{sub})],$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal} \cap \Sigma_{is},\ s_{orthogonal}[r(s_{sub})] \in \Sigma_{ss} \cap \Sigma_{is},\ r = \mathcal{R}(s_{orthogonal}),$
$incoming(s_{orthogonal}) = \emptyset,\ outgoing(s_{orthogonal}) = \emptyset$ ◇

T 46 (*Shallowhistory*-Verknüpfung)

Abbildung 4.62: Transformationsregel für die Shallowhistory-Verknüpfung

$$\Leftrightarrow \; s_{orthogonal}(c_{history}) \Longrightarrow s_{orthogonal},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal}, \nexists children(children(s_{orthogonal}[r])), incoming(s_{orthogonal}) = outgoing(s_{orthogonal}) = \emptyset$ ◇

Mit Regel T_{46} werden alle noch vorhandenen *History*-Verknüpfungen gelöscht und in Regel T_{47} wird der transitionslose orthogonale Zustand gelöscht. Seine Unterzustände sind nun Unterzustände der nächsthöheren Hierarchie. In ihren statischen Aktivitäten wird zusätzlich als Letztes die statische Aktivität des ehemals orthogonalen Zustandes ausgeführt.

T 47 (Einebnen des orthogonalen Hierarchiezustandes)

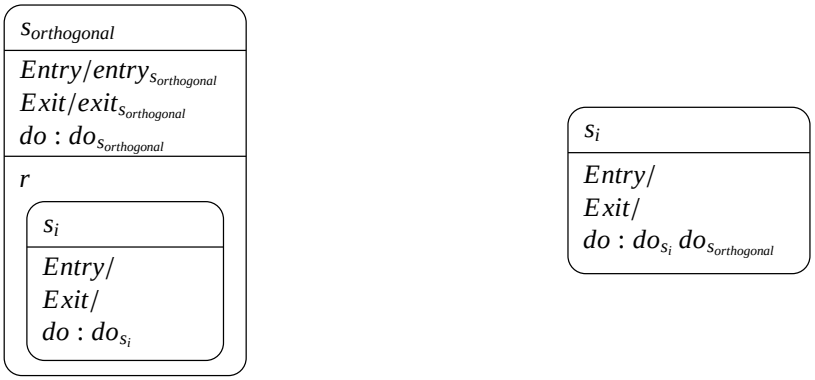


Abbildung 4.63: Transformationsregel für das Einebnen des orthogonalen Hierarchiezustandes

$$\Leftrightarrow \; \{s_{orthogonal}[r(s_i)]\} \Longrightarrow \{s_i\},$$

$EM_l : s_{orthogonal} \in \Sigma_{orthogonal}, \; s_{orthogonal}[r(s_i)], s_i \in \Sigma_{ss}, \; i = 1, \ldots, |children(s_{orthogonal})|,$
$do(s_i) = do(s_{orthogonal}[r(s_i)])\, do(s_{orthogonal}),$ ◇

Ein Nebeneffekt der Explizierung von Transitionen in den Regeln T_{22} bis T_{34} und der darauffolgenden Produktbildung ist die Schaffung redundanter Transitionen, die mit Regel T_{48} behandelt werden sollen.

T 48 (Redundante Transitionen)

Abbildung 4.64: Transformationsregel für redundante Transitionen

$$\Leftrightarrow\ s' \xleftarrow[p]{e[c]/a} s \xrightarrow[p]{e[c]/a} s' \Longrightarrow s \xrightarrow[p]{e[c]/a} s',$$

$$EM_l : s \in \Sigma_{ss} \cup c_{junction},\ s' \in \Sigma_{ss} \qquad \diamond$$

Falls eine *Junction*-Verknüpfung übrig bleibt, die nur eine eingehende und eine ausgehende Transition besitzt, können beide Transitionen unter Vernachlässigung der *Junction*-Verknüpfung ausgedrückt werden:

T 49 (Erübrigende *Junction*-Verknüpfung)

Abbildung 4.65: Transformationsregel für eine sich erübrigende Junction-Verknüpfung

$$\Leftrightarrow\ s \xrightarrow[p]{e[c]} c_{junction} \xrightarrow[p]{e'[c']/a} s' \Longrightarrow s \xrightarrow[p]{e[c\ \&\&\ c']/a} s'$$

$EM_l : s, s' \in \Sigma_{ss},\ t_1 = (s, l_{T_1}, c_{junction}),\ t_2 = (c_{junction}, l_{T_2}, s'),\ \nexists t_1' \in incoming(c_{junction}),\ t_1' \neq t_1,\ \nexists t_2' \in outgoing(c_{junction}),\ t_2' \neq t_2$ $\diamond$

4.3.4 Syntaktische Transformationsregeln

Der Vollständigkeit halber soll noch auf Transformationen von Komponenten mit gleicher Semantik, aber verschiedener Syntax eingegangen werden. Sie sind jedoch nur in einem formalen System von Interesse, da in einem Modell die Notationen für Komponenten nicht frei wählbar sind. Wegen der Einfachheit der Regeln soll dabei auf die textliche Beschreibung verzichtet werden.

S 1 (Startzustand)

Abbildung 4.66: Syntaktische Transformationsregel für einen Startzustand

S 2 (Markierungen von Aktivitäten)

Abbildung 4.67: Syntaktische Transformationsregel für Markierungen von Aktivitäten

4.3.5 Steuersprache der Transformation

Nach Definition 39 soll nun ein Steuermechanismus für die Sprache des Transformationssystems $L(G^T)$, die Reihenfolge und die Anzahl der Anwendungen von Produktionen aus G^T betreffend, entwickelt werden. Die Marke für eine Produktion P in G^T sei ihre Benennung: $Lab(P) = \{T_1, \ldots, T_{49}, F_1, \ldots, F_{18}, S_1, S_2\}$. Die Steuersprache C^T der Ableitung $SC_0 \Longrightarrow^* SC \in \mathcal{SC}(T_S, T_\Sigma, T_T)$ wird dann durch den folgenden regulären Ausdruck R bestimmt:

$$
\begin{gathered}
R = T_1^* T_2^* T_3^* T_4^* T_5^* T_6^* T_7^* T_8^* T_9^* T_{10}^* T_{11}^* T_{12}^* T_{13}^* T_{14}^* (T_{15} T_{16}^* T_{17}^* T_{18}^* T_{19}^*)^* T_{20}^* T_{21}^* \\
(F_1^* F_2^* F_3^* F_4^* F_5^* F_6^* F_7^* F_8^* F_9^* F_{10}^* F_{11}^* F_{12}^* F_{13}^* F_{14}^* F_{15}^* F_{16}^* \\
T_{22}^* T_{23}^* T_{24}^* T_{25}^* T_{26}^* T_{27}^* T_{28}^* T_{29}^* T_{30}^* T_{31}^* T_{32}^* T_{33}^* T_{34}^* T_{35}^* T_{36}^* T_{37}^* T_{38}^* T_{39}^* (T_{40}^* T_{41}^*)^* \\
T_{42}^* T_{44}^* T_{45}^* T_{46}^* T_{47}^* T_{48}^* T_{49}^*)^* S_1^* S_2^*,
\end{gathered}
$$

d. h. $C = < R >$. Durch Mehrfachanwendungen der Folge T_{15} bis T_{19} wird durch jeweiliges Vertauschen der Prioritäten eine Sortierung von Prioritäten erreicht. Die Mehrfachanwendungen der Folge F_1 bis F_{16} erreichen eingeebnete serielle Hierarchiezustände des gesamten *Statecharts*. Gleiches gilt für die Folge T_{22} bis T_{48}, die die orthogonalen Zustände transformiert. Nach Anwendung aller Transformationsregeln auf ein *Statechart* aus *Artisan RtS* ist mit der Steuersprache C^T somit eine visuelle *Statechart*-Sprache $L(G^T, C^T)$ entworfen worden, die nur noch Elemente von Zustandsautomaten aus *Ascet-SD* enthält. Das Ergebnis ist somit ein flacher Zustandsautomat mit dem gleichen Verhalten wie das *Statechart* SC_0 aus *Artisan RtS*, welches zu transformieren war.

4.3.6 *Flattening* von *Ascet-SD*

Da bei der Entwicklung der Transformationsregeln großer Wert auf Wiederverwendbarkeit gelegt wurde, ist es nun sehr einfach möglich, die Regeln auf andere Transformationsaufgaben auszurichten. So wäre im Rahmen des Einsatzes eines *Model-Checkers* für die Konsistenzprüfung der implementierten Zustandsautomaten in *Ascet-SD* eine flache Hierarchie als Ausgangsbasis für die Prüfung unabdingbar. Selbstverständlich ist dabei, dass ein flacher Zustandsautomat das gleiche Verhalten aufweist, wie der entsprechend hierarchische.

Das *Flattening* von *Statechart* in *Artisan RtS* wurde bereits im Abschnitt 4.3.2 besprochen. Soll nun ein Zustandsautomat in *Ascet-SD* eingeebnet werden, so ist dabei die genaue Kenntnis der oft implizit vorhandenen Regeln beim Ablauf des Modells von entscheidender Bedeutung. analysiert man nun Zustandsautomaten, lässt sich erkennen, dass die Reihenfolge der Auswertung der Transitionen an den verschiedenen Hierarchiestufen im Gegensatz zu *Statecharts* mit der höchsten Hierarchiestufe beginnt. Dem kann beim *Flattening* der Zustandsautomaten durch entsprechende Veränderung der Transformationsregel F_1 Rechnung getragen werden. Abb. 4.68 zeigt die veränderte Regel.

F 17 (Prioritäten am seriellen Zustand in *Ascet-SD*)

Abbildung 4.68: Flattening-Regel für Prioritäten am seriellen Zustand

$$\Leftrightarrow \{s_{serial}(s_{sub_m})\},\ s_{serial}\{\xrightarrow[p_i]{e_i[c_i]/a_i} s_n\} \Longrightarrow \{s_{serial}(s_{sub_m})\},\ s_{serial}\{\xrightarrow[p_i']{e_i[c_i]/a_i} s_n\}$$

$$\text{wobei } p_i' = \overline{p}(outgoing^*(s_{sub_m})) + p_i,$$

$EM_l : s_{serial} \in \Sigma_{serial},\ s_{serial}(s_{sub_m}) \in \Sigma_{ss},\ s_n \in children(s_{serial}) \cup \complement children(s_{serial}) \cap \Sigma,\ i = 1, \ldots, \left| \bigcup_{\substack{t=(s_{serial}, l_T, s'), \\ s_{serial} \in \Sigma_{cs}, s' \in \Sigma}} t \right|,\ m, n \leq i$ ◇

Ersetzt man nun Regel F_1 durch Regel F_{17} beim *Flattening* in G^T, erhält man eine *Flattening*-Vorschrift für Zustandsautomaten in *Ascet-SD*. Die Steuersprache für das *Flattening* von Zustandsautomaten wird dann durch den regulären Ausdruck

$$T_8^* T_{10}^* (F_{17}^* F_2^* F_3^* F_4^* F_5^* F_6^* F_7^* F_8^* F_9^* F_{10}^* F_{11}^* F_{12}^* F_{13}^* F_{14}^* F_{15}^* F_{16}^*)^*$$

beschrieben. Um die große Nähe des *Flattening* für *Statechart* zu dem der Zustandsautomaten aufzuzeigen, wurden die *Flattening*-Regeln für Zustandsautomaten benutzt. Dort

benutzte Notationen von *Statecharts* finden sich teilweise nicht in Zustandsautomaten wieder. Mit dem Wissen um die semantische Äquivalenz einiger Komponenten, das dem Leser auf den Seiten 62f. vermittelt wurde, kann das *Flattening* nun durchaus auf die Komponenten von Zustandsautomaten angewandt werden.

Kapitel 5

Fallbeispiel

Anhand eines Beispiels soll nun das Transformationssystem aus Abschnitt 4.2 in der Anwendung veranschaulicht werden. Das Beispiel wurde nach den einschränkenden Kriterien des Abschnitts 4.1.3 entworfen, jedoch wurden zu Gunsten der Übersichtlichkeit längst nicht alle möglichen Konstruktionselemente benutzt.

Das Beispiel beschreibt eine Ampelsteuerung einer stark vereinfachten Straßenampelanlage, die ohne die ‚Rot-Gelb'-Lichtphase auskommt. Die drei Farben der Ampel sind in Abhängigkeit einer ihnen zugeordneten Zeit aktiv. Zusätzlich zu der Farbensteuerung wurde ein möglicher Fehlerbetrieb der Ampel modelliert, in dem die Ampel mit dem gelben Licht blinkt und unabhängig ein Funksignal versendet und so den Betreiber über den Fehlerfall informiert. Die Ampel kann durch ein Fehlersignal in den Fehlerzustand eintreten, das etwa durch Sensorik der Ampel ausgelöst wird. Nach der Wartung der Anlage kann dann ein Signal gesetzt werden, mit dem die Ampel wieder in den normalen Betriebszustand übergeht.

Die Transformation des *Statecharts* mit den sprachlichen Mitteln von *Artisan RtS* erfolgt mit den Produktionen des Transformationssystems aus Abschnitt 4.2 und wird nachfolgend auf das *Statechart* in Abb. 5.1 angewendet. Ableitungspfeile machen offensichtlich, welche *Statechart*-Produktionen angewendet wurden. Die Reihenfolge der Ableitungen wird mit der Steuersprache auf den Produktionen in Abschnitt 4.3.5 vorgeschrieben. Ein ‚$*$' am Pfeil bedeutet dabei die Mehrfachanwendung der Regel.

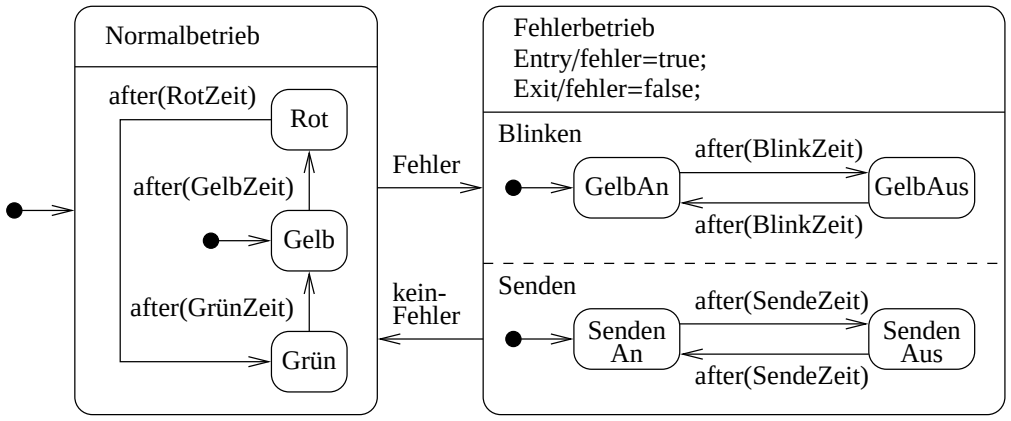
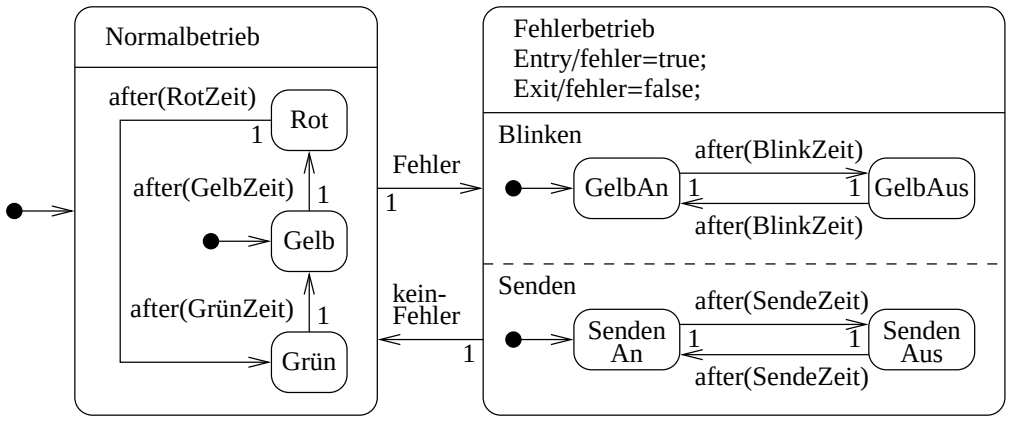
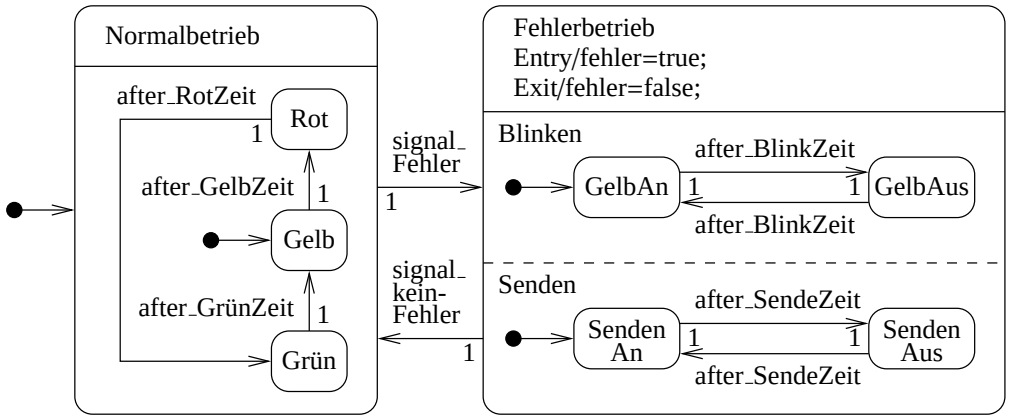


Abbildung 5.1: Fallbeispiel für das Statechart einer Ampelsteuerung

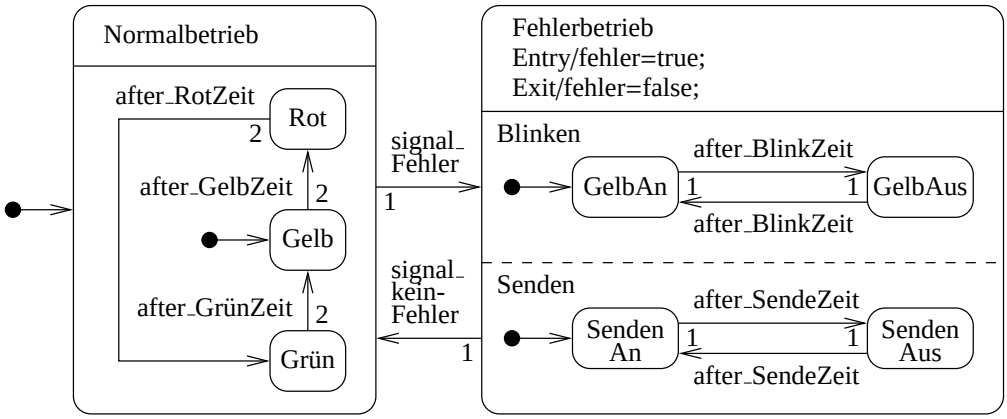
$\overset{T_1}{\Longrightarrow}^*$



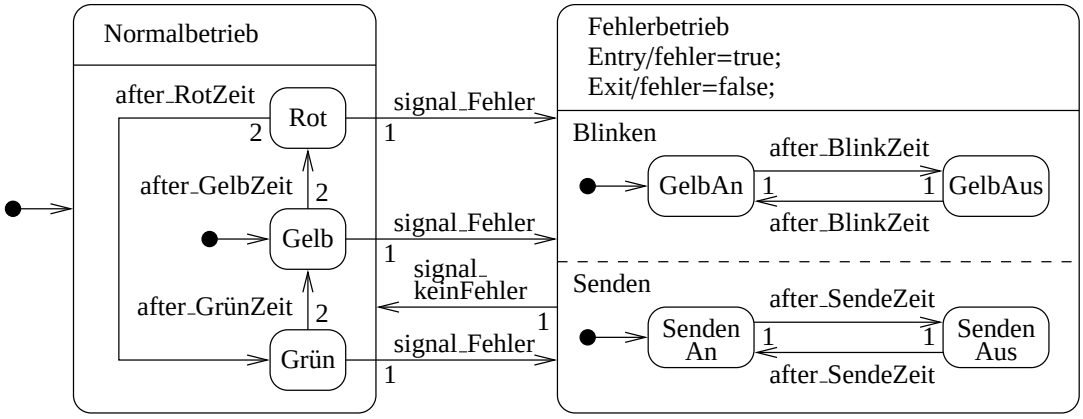
$\overset{T_5}{\Longrightarrow}^*$



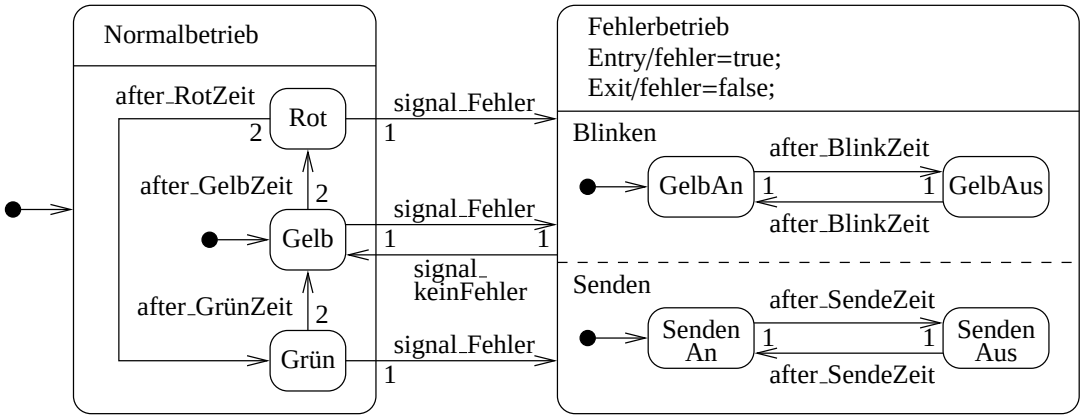
$\stackrel{F_1}{\Longrightarrow}$

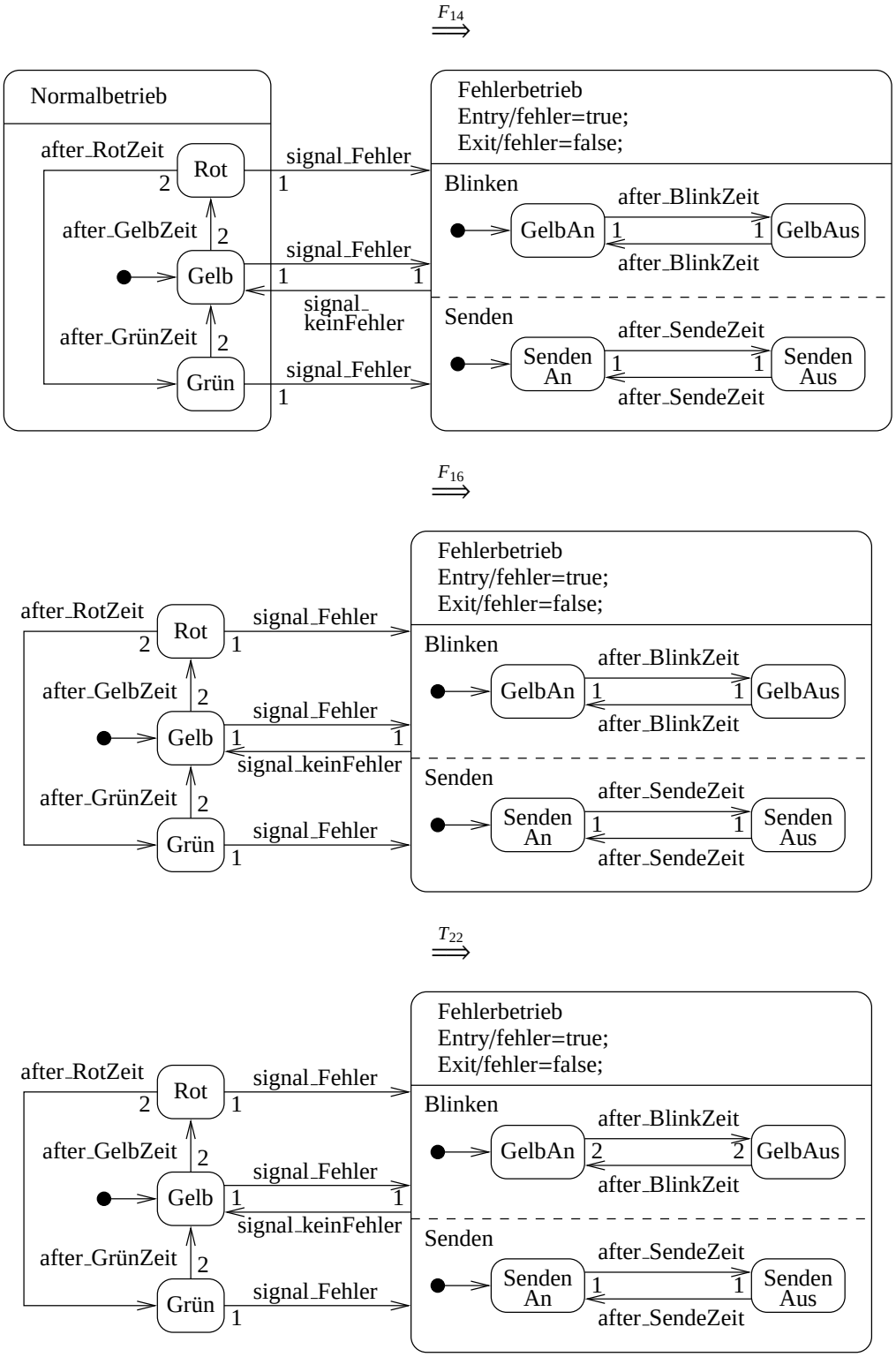


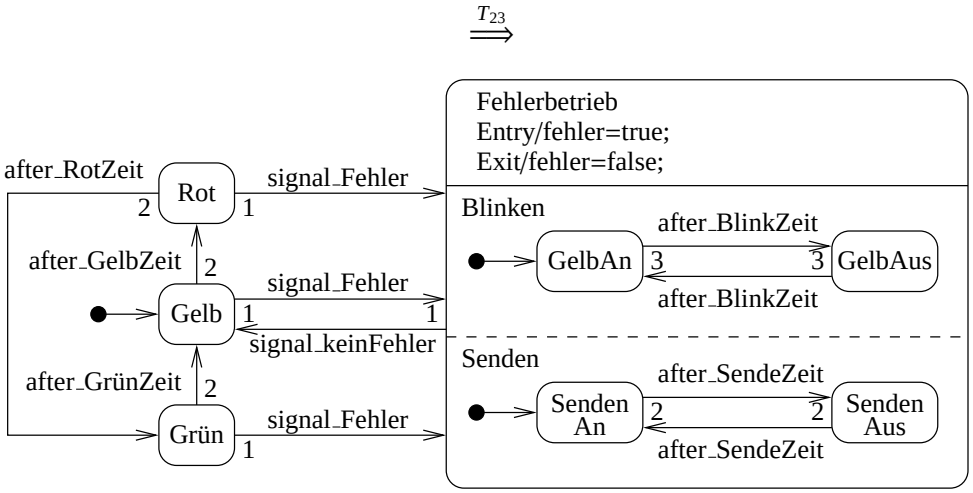
$\stackrel{F_9}{\Longrightarrow}$



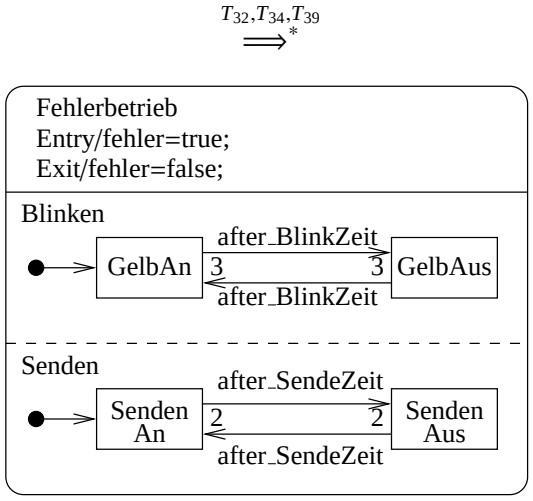
$\stackrel{F_{11}}{\Longrightarrow}$



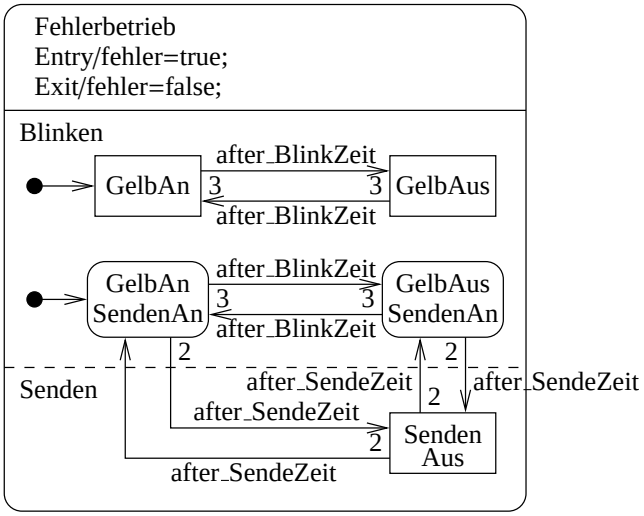

F_{14}
Normalbetrieb
after_RotZeit
Rot
2
signal_Fehler
1
after_GelbZeit
2
Gelb
signal_Fehler
1
1
signal_
keinFehler
after_GrünZeit
2
Grün
signal_Fehler
1
Fehlerbetrieb
Entry/fehler=true;
Exit/fehler=false;
Blinken
GelbAn
after_BlinkZeit
1
1
GelbAus
after_BlinkZeit
Senden
Senden
An
after_SendeZeit
1
1
Senden
Aus
after_SendeZeit
F_{16}
after_RotZeit
Rot
2
signal_Fehler
1
after_GelbZeit
2
Gelb
signal_Fehler
1
1
signal_keinFehler
after_GrünZeit
2
Grün
signal_Fehler
1
Fehlerbetrieb
Entry/fehler=true;
Exit/fehler=false;
Blinken
GelbAn
after_BlinkZeit
1
1
GelbAus
after_BlinkZeit
Senden
Senden
An
after_SendeZeit
1
1
Senden
Aus
after_SendeZeit
T_{22}
after_RotZeit
Rot
2
signal_Fehler
1
after_GelbZeit
2
Gelb
signal_Fehler
1
1
signal_keinFehler
after_GrünZeit
2
Grün
signal_Fehler
1
Fehlerbetrieb
Entry/fehler=true;
Exit/fehler=false;
Blinken
GelbAn
after_BlinkZeit
2
2
GelbAus
after_BlinkZeit
Senden
Senden
An
after_SendeZeit
1
1
Senden
Aus
after_SendeZeit



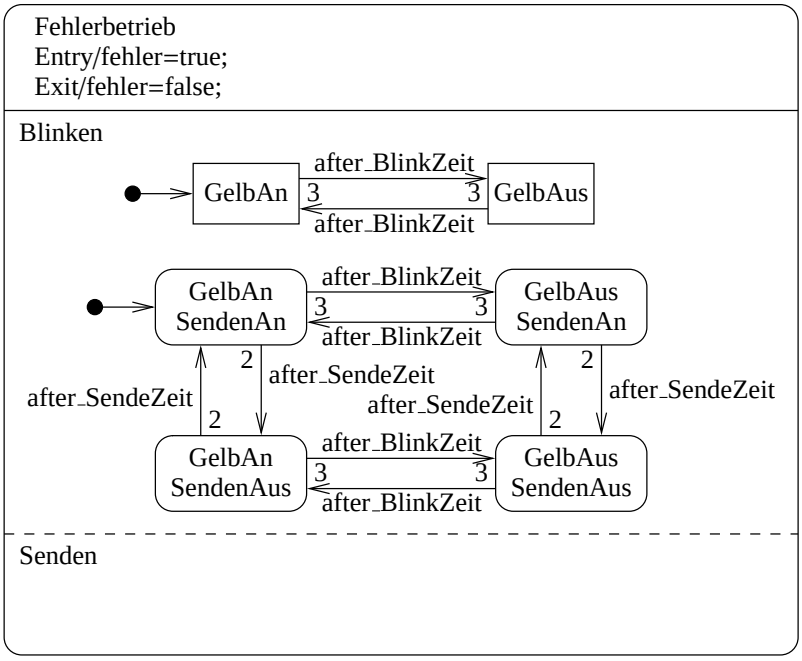
Nach Anwendung der Regeln T_{32} und T_{34} sind die Transitionen von dem orthogonalen Zustand *Fehlerbetrieb* abgelöst und haben neue Quellen bzw. Ziele gefunden, dabei wurden die Aktivitäten des orthogonalen Zustandes berücksichtigt. Da dadurch das *Statechart* sehr unübersichtlich geworden ist, soll vorerst nur das Unter-*Statechart* betrachtet werden, das jetzt zur Sequentialisierung zur Verfügung steht. Der Rest des *Statecharts* wird zu einem späteren Zeitpunkt wieder angefügt werden.



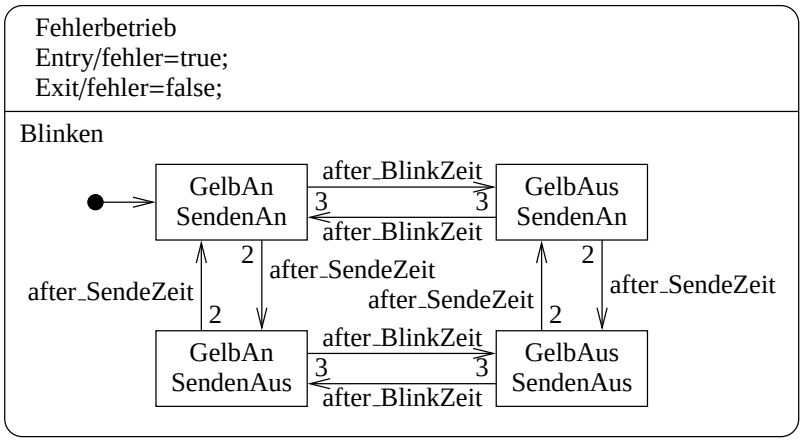
$\stackrel{T_{40}}{\Longrightarrow}$



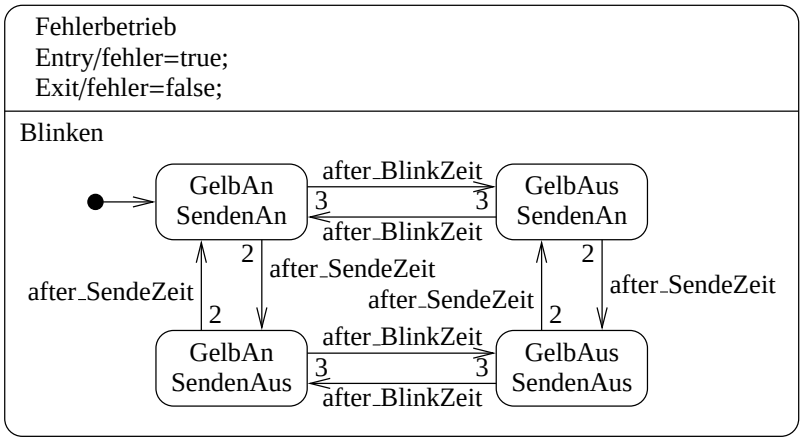
$\stackrel{T_{40}}{\Longrightarrow}$



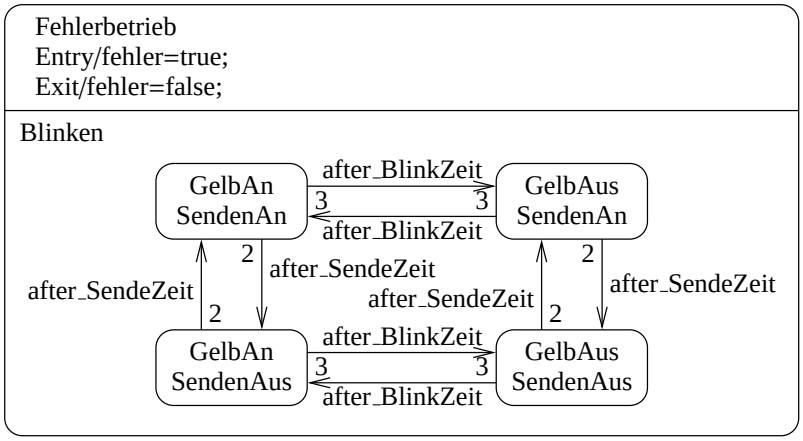
$\overset{T_{41}}{\Longrightarrow}$

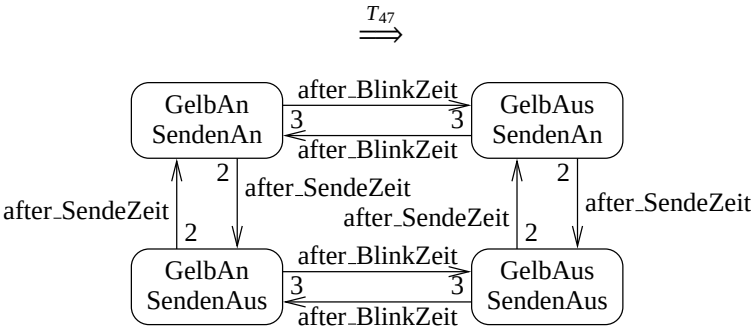


$\overset{T_{42}}{\Longrightarrow}^{*}$

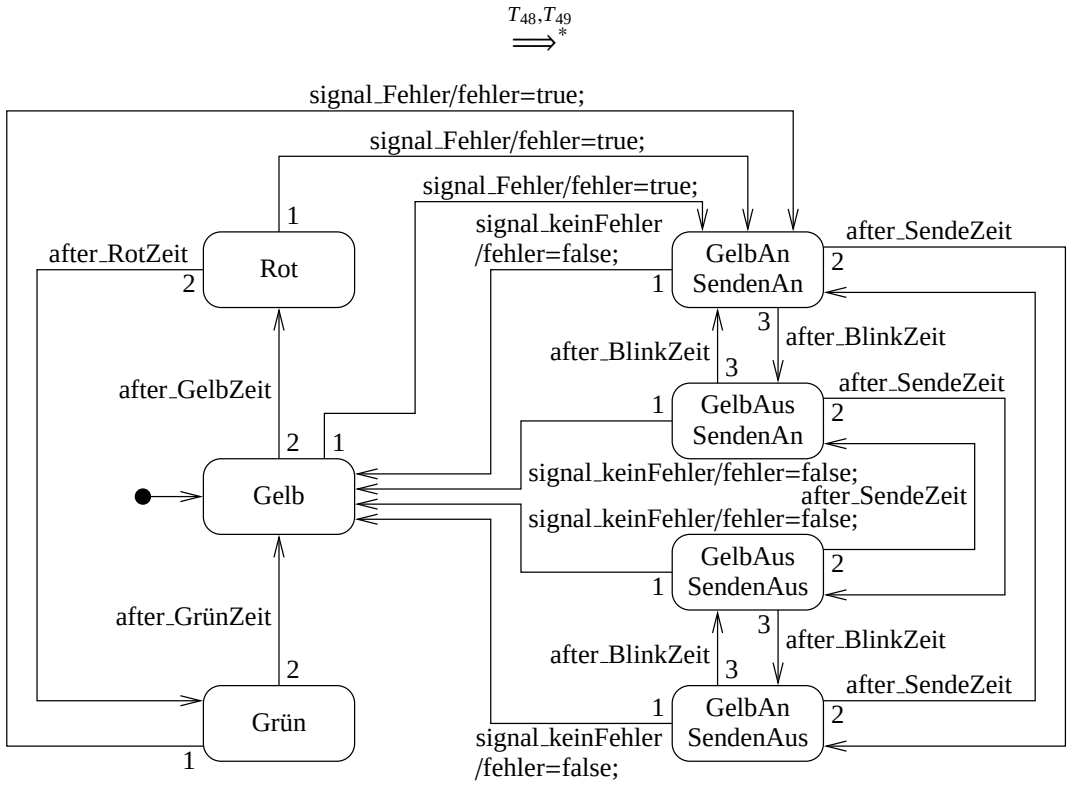


$\overset{T_{44}}{\Longrightarrow}$



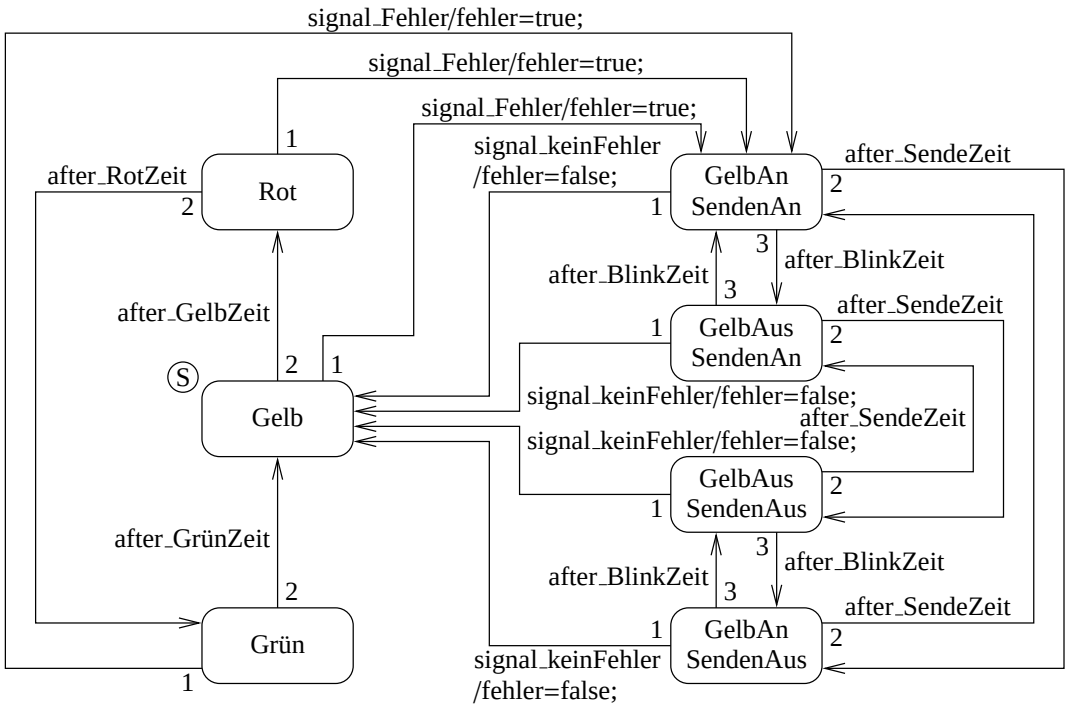


Bevor nun weitere Regeln angewendet werden, soll das bei der Sequentialisierung allein betrachtete Teil-*Statechart* wieder mit dem restlichen Teil-*Statechart* verschmolzen werden. Auf die vorher generierten Transitionen werden dann die Regeln T_{48} und T_{49} angewandt. Das daraus resultierende *Statechart* ist nachfolgend abgebildet.



Nun soll der Vollständigkeit halber noch die Syntax angepasst werden; diese Ableitung ist nur von formalem Interesse.

$\stackrel{S_1}{\Longrightarrow}$



Kapitel 6

Transformationswerkzeug

In diesem Kapitel sollen die wesentlichen Eigenschaften des *Software*-Programms beschrieben werden, das zu dieser Arbeit geschaffen wurde. Es zeigt zum einen dem Leser dieser Arbeit, dass mit den Regeln in Abschnitt 4.3 die Transformation von *Statecharts* möglich ist und zum anderen dient es dem Anwender des Programms als Werkzeug für die automatisierte Transformation von *Statecharts*.

Wie bereits im ersten Kapitel erwähnt, existiert für *Artisan RtS* derzeit kein *XMI/XML*-Export, so dass zunächst ein anderes Werkzeug mit Exportmöglichkeit für die Erstellung von *Statecharts* eingesetzt werden muss. Die Wahl fiel dabei auf *I-Logix™ Rhapsody®*, da es ähnliche Strukturelemente wie *Artisan RtS* aufweist und ebenso wie *Artisan RtS* am Institut eingesetzt wird. Die *Statecharts* von *Rhapsody* werden bei der automatischen Transformation so behandelt, als wären sie mit *Artisan RtS* erzeugt, d. h. es wird nicht auf die Eigenheiten der *Statecharts* von *Rhapsody* eingegangen. Durch den Einsatz von *Rhapsody* werden die Beschränkungen und Veränderungen bzgl. der Syntax der *Statecharts* von *Artisan RtS* wirksam, wie sie in Taf. 6.1 aufgeführt sind.

6.1 Programmbeschreibung

Das Transformationswerkzeug baut auf der Objektstruktur für *Statecharts* der *Software*-Arbeit zu einem *Regel Checker* von Pietsch [Pie03] auf, indem die dort serialisierte Objektstruktur eingelesen und verarbeitet wird. Der *Regel Checker* importiert *XMI*-Daten von *Statecharts*, prüft diese auf ihre Voraussetzungen für die Transformation dieser Ar-

Tafel 6.1: Beschränkungen und Veränderungen von Statecharts beim Einsatz von Rhapsody

Komponente	*Artisan RtS*	*Rhapsody*
Reihenfolge der Regionen eines orthogonalen Zustandes ist bestimmbar	ja	nein
Anzahl der eingehenden Transitionssegmente in eine *Choice*-Verknüpfung	mehrere	eine
Markierung des von der *Fork*-Verknüpfung ausgehenden Transitionssegmentes	Aktion	leer
Markierung des in die *Join*-Verknüpfung einkommenden Transitionssegmentes	Aktion	leer
Trigger-Beschriftung einer internen Transition kann leer sein	ja	nein
Existenz einer *Shallowhistory*-Verknüpfung	ja	nein

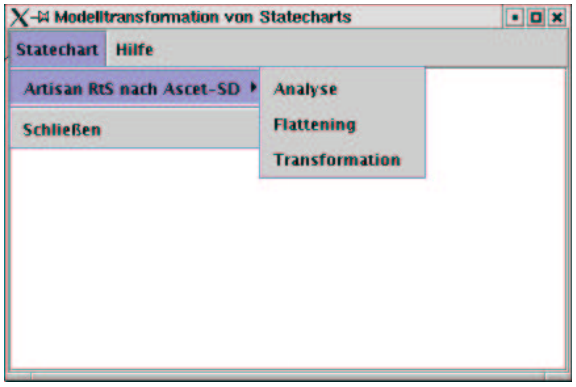


Abbildung 6.1: Graphische Oberfläche des Transformationswerkzeuges

beit und interpretiert diese durch die dafür entworfene Objektstruktur. Er liefert damit die *Statecharts*, die der Einschränkung des Abschnittes 4.1.3 entsprechen und als Eingabe für das Transformationswerkzeug dienen.

Die Implementierung des Transformationswerkzeuges erfolgte in *Java*. Es wird mit

```
java Transformation
```

gestartet und erwartet eine Datei `artisan.sc` des oben beschriebenen Formates. Bei der Initialisierung wird diese Datei geöffnet und das damit beschriebene *Statechart* eingelesen. Abb. 6.1 zeigt die graphische Oberfläche des Programms.

Der Menüpunkt „Analyse“ des Menüs „Statechart → Artisan RtS nach Ascet-SD“ aktiviert eine umfassende Analyse des eingelesenen *Statecharts* bzgl. der Transformation. Nachdem dabei die verschiedenen Komponenten klassifiziert wurden, informiert die *Software* über auszuführende Transformationsregeln und die dabei zu erledigenden Ersetzungen. Das betrifft auch solche, die in Folge anderer ausgeführt werden. Mit Menüpunkt „Transformation“ des gleichen Menüs kann die Transformation gestartet werden. Das Ergebnis der Transformation wird in die Datei `ascet.sc` geschrieben, die im Format der oben beschriebenen serialisierten Objektstruktur einen hierarchischen Zustandsautomaten von *Ascet-SD* beschreibt. Mit dem Menüpunkt „Flattening“ werden serielle Hierarchiezustände von *Artisan RtS* unabhängig von der Transformation eingeebnet und in der Datei `artisan_flach.sc` abgelegt. Im Projekt *STEP-X* ist vorgesehen, eine Wandlung der Objektstruktur in *XML*-Daten als Eingabeformat für *Ascet-SD* zu vollziehen.

Sowohl Analyse als auch Transformation des Programms notieren Informationen zu den Ersetzungen auf der graphischen Oberfläche des Programms sowie in einer *ASCII*-Datei `transformation.log`. Folgend ist ein Auszug dieser Ausgabe zu sehen. Das Programm wurde dafür auf das *Statechart* des Beispiels des Kapitels 5 angewendet. Die Ausgabe enthält Informationen über die Vorgehensweise des Programms bei der Reorganisation von Transitionen am seriellen Hierarchiezustand.

```
128 Transitionen des Typs (s_serial,l_T,s_outside)
129 --> Transition '(Normalbetrieb,l_T,Fehlerbetrieb)', l_T='Fehler/', Priorität: 1
130 --> neue Transition '(Rot,l_T,Fehlerbetrieb)', l_T='Fehler/', Priorität: 1
131 --> neue Transition '(Gelb,l_T,Fehlerbetrieb)', l_T='Fehler/', Priorität: 1
132 --> neue Transition '(Gruen,l_T,Fehlerbetrieb)', l_T='Fehler/', Priorität: 1
133 --> löschen der Transition '(Normalbetrieb,l_T,Fehlerbetrieb)'
134 Transitionen des Typs (s_outside,l_T,s_serial), s_serial ist kein History-Zustand
135 --> Transition '(Fehlerbetrieb,l_T,Normalbetrieb)', l_T='keinFehler/', Priorität: 1
136 --> neue Transition '(Fehlerbetrieb,l_T,Gelb)', l_T='keinFehler/', Priorität: 1
137 --> löschen der Transition '(Fehlerbetrieb,l_T,Normalbetrieb)'
```

6.2 Implementierung

Die Klasse `ArtisanToAscet` soll als Kern des Programms hier alleinig beschrieben werden. Zu ihr gehören alle im Folgenden beschriebenen Methoden. Die Klasse erhält bei Initialisierung das zu transformierende *Statechart* sowie eine *log*-Datenstruktur zum Schreiben der Benutzerinformationen. Die Methode `Transform()` vollzieht als wichtigste Methode der Klasse die gesamte Transformation eines *Statecharts*. Sie benutzt die Methode `Order()` zur Steuerung der Regeln des Abschnittes 4.3 bzgl. Reihenfolge und Häufigkeit der Anwendung, wie sie mit der Steuersprache des Abschnittes 4.3.5 festgelegt wurde. Der bereits implementierte Teil der Regeln soll im Folgenden kurz beschrieben werden. Für jede Regel erfolgt ein Verweis auf ihre formale Definition in Kapitel 4.3.

`T_TransitionOhnePrioritaet()`: Die Prioritäten aller Transitionen werden mit dieser Methode auf den Wert eins gesetzt. (T_1)

`T_Trigger()`: Alle *Trigger* werden entsprechend ihrer ursprünglichen Definition benannt und für die Verwendung in *Ascet-SD* als *Call*-Ereignisse klassifiziert. (T_5)

`T_InterneTransitionen()`: Mit dieser Methode werden Bedingungen und Aktionen von internen Transition in statische Aktivitäten gewandelt. (T_{20})

`F_PrioritaetenAmSeriellenZustand()`: Die Methode erhöht Prioritäten der Transitionen, die von Unterzuständen eines seriellen Hierarchiezustandes ausgehen, um die maximale Priorität der vom Hierarchiezustand ausgehenden Transitionen. (F_1)

`F_TSSerialSOutside()`: Hier werden die vom seriellen Hierarchiezustand ausgehenden Transitionen abgelöst und jedem Unterzustand eine zusätzliche Transition mit der Beschriftung und dem Ziel der abgelösten Transition zugeteilt. (F_9)

`F_TSOutsideSSerial_SSerialNotHis()`: Die in einen seriellen Zustand einkommenden Transitionen werden an dem Unterzustand, der auch Startzustand ist, angebracht. (F_{11})

`F_Startverknuepfung_SSerialIs()`: Hier wird die Startverknüpfung eines Unterzustandes nebst zugehöriger Transition gelöscht. (F_{14})

`F_EinebnenDesSeriellenHierarchiezustandes()`: Ein serieller Hierarchiezustand wird gelöscht und seine Unterzustände rücken eine Hierarchiestufe höher; sie bekommen zusätzlich die statische Aktivität des Hierarchiezustandes. (F_{16})

`T_KonkurrierendeTransitionenAnRegionen()`: Prioritäten von Transitionen, welche von Unterzuständen einer Region ausgehen, werden bzgl. jenen, die von Unterzuständen der Region darunter ausgehen, erhöht. (T_{22})

`T_PrioritätenAmOrthogonalenZustand()`: Die Prioritäten von Unterzuständen des orthogonalen Hierarchiezustandes ausgehender Transitionen werden um die maximale Priorität der vom Hierarchiezustand ausgehenden Transitionen erhöht. (T_{23})

`T_TSOrthogonalSOutside()`: Es werden die vom orthogonalen Hierarchiezustand ausgehenden Transitionen abgelöst und jedem Unterzustand eine zusätzliche Transition mit der Beschriftung und dem Ziel der abgelösten Transition zugeteilt. (T_{32})

`T_TSOutsideSOrthogonal()`: In den orthogonalen Zustand einkommende Transitionen werden an den Unterzuständen, der auch Startzustände sind, angebracht. (T_{34})

`T_Faerben()`: Alle Unterzustände von orthogonalen Zuständen werden in Vorbereitung auf die Produktbildung als gefärbt markiert. (T_{39})

`T_Produktbildung()`: Mit dieser Methode werden neue Zustände aus jeweils zwei Zuständen der untersten beiden Regionen eines orthogonalen Zustandes gebildet. Die unterste Region wird dabei von Unterzuständen befreit. (T_{40})

`T_TilgenEinerRegion()`: Die unterste Region eines orthogonalen Zustandes, die keine Unterzustände hat, wird gelöscht. (T_{41})

`T_Entfaerben()`: Die Färbung von Zuständen wird rückgängig gemacht. (T_{42})

`T_Startverknüpfung_SOrthogonalNotIs()`: Der Startzustand, der Unterzustand eines orthogonalen Hierarchiezustandes ist, wird von seiner Startverknüpfung und der dazugehörigen Transition befreit. (T_{44})

`T_EinebnenDesOrthogonalenZustandes()`: Es erfolgt die Tilgung eines orthogonalen Hierarchiezustandes, wobei seine Unterzustände eine Hierarchiestufe aufsteigen und die statische Aktivität des Hierarchiezustandes erhalten. (T_{47})

Bevor eine Regel angewendet wird, werden gemäß der jeweiligen Einbettungsüberführungsregel die Voraussetzungen überprüft. Sind diese erfüllt, wird die Regel beim *Statechart* zum Einsatz gebracht, ansonsten wird zur nächsten Regel entsprechend der Reihenfolge in der Steuermethode `Order()` übergegangen.

Kapitel 7

Zusammenfassung und Ausblick

Die wachsende Komplexität elektronischer Systeme fordert immer stärker den Einsatz abstrakter modellbasierter Entwurfstechniken. Dabei kommen Systeme zur Modellierung, Simulation und Verifikation zur Anwendung; speziell für die Modellierung dynamischer Systeme sind dort *Statecharts* vorgesehen. Sobald man sich jedoch für den Einsatz eines Produktes entschieden hat, ist man häufig auf dieses beschränkt, da es immer eine eigene spezielle – mehr oder weniger genau beschriebene – Syntax und Semantik einführt (s. Abschnitt 2.1). Damit ist man, auch wenn es Im- und Exportmodule zum Produkt geben sollte, auf ebendiese Konstruktionsmerkmale beschränkt, da die Semantik nicht überführt wird. Der Grund dafür, dass trotz der Standardisierungsversuche der *UML* Divergenzen bei der Entwicklung von *Statechart*-Implementierungen auftreten, ist offenbar, dass Beschreibungen vieler Konstruktionsmerkmale durch die *UML* offen gelassen werden, was zum einen den nötigen Raum für eigene Interpretationen schafft, zum anderen aber unterschiedliche Implementationen provoziert.

Diese Arbeit schafft durch Transformation Abhilfe bzgl. des Problems verschiedener Semantiken von *Statecharts* in Modellimplementierungen. Dabei konnten die folgenden Erkenntnisse und Ergebnisse erzielt werden:

- Es wurde ein formaler Ansatz für die Überführung verschiedener Semantiken und Syntaxen von *Statecharts* jeglicher Modellimplementierung entwickelt. Grundlage dafür war die Definition des *verallgemeinerten Statecharts* in Definition 31, mit dem alle möglichen Versionen von *Statecharts* syntaktisch beschrieben werden können. Der Mechanismus der Überführung nimmt Anleihen beim mengentheoretischen Ansatz für Graphgrammatiken und realisiert so einen *mengentheoretischen Ansatz für Statechart-Grammatiken* (Definition 36). Er arbeitet auf der vereinigten Menge der Notationen der Strukturelemente für ein zu transformierendes *Statechart* und ermöglicht Transformationen durch Manipulation von Teil-*Statecharts* durch *Statechart*-Produktionen.

- Zur Steuerung der Reihenfolge und Anzahl der Anwendungen von Produktionen wurde eine *Steuersprache* entworfen (Definition 39), mit der die Sprache einer *Statechart*-Grammatik (Definition 38) bei der Ableitung eines *Statecharts* konstruiert wird. Mit der so entwickelten Sprache ist zugleich die neue Semantik eines (abgeleiteten) *Statecharts* definiert.

- Eine umfassende *Analyse* ergab die in den Abschnitten 2.2.1 bzw. 2.2.2 und 2.3.1 bzw. 2.3.2 erscheinende, hinreichend exakte *Beschreibung der Notation und des Verhaltens* von *Statecharts* in *Artisan RtS* und von Zustandsautomaten in *Ascet-SD*. Diese betrifft zum einen Anlehnungen der Strukturelemente an Standardisierungen der *UML* als auch modellspezifische Bestandteile, die keiner Standardisierung unterliegen. Die genaue Verhaltensbeschreibung resultierte in einem *Syntax- und Semantikvergleich* beider Modelle in Abschnitt 4.1.1.

- Basierend auf der syntaktischen Analyse und als Grundlage für eine Transformation erfolgte eine Adaption von *Statecharts* und Zustandsautomaten an *verallgemeinerte Statecharts*. Damit wurde in den Abschnitten 2.2.3 und 2.3.3 die mathematische *Formalisierung der Syntax* der Modelle aufgestellt.

- Aufbauend auf den Formalisierungen der *Statecharts*, dem Entwurf einer angepassten *Statechart*-Grammatik für die Überführung von *Statecharts* in *Artisan RtS* nach Zustandsautomaten in *Ascet-SD* in Abschnitt 4.2 und der Beschränkung der Strukturelemente auf übersetzbare erfolgte nun exemplarisch in Abschnitt 4.3 die Formulierung von entsprechenden *Statechart*-Produktionen zur Ableitung. Dabei wurde großer Wert auf *Überführung möglichst aller Kompositionsmerkmale*, Wiederverwendbarkeit und Bibliotheksbildung der Produktionen gelegt. Eine konkrete Steuersprache für die Ableitung gibt Anweisungen für Häufigkeit und Reihenfolge der Produktionen (Abschnitt 4.3.5).

- Durch die Möglichkeit der Wiederverwendbarkeit der Produktionen konnte in Abschnitt 4.3.2 das *Einebnen hierarchischer Konstrukte in Zustandsautomaten von Ascet-SD* zusätzlich und unabhängig von der Transformation mittels veränderter o. g. Produktionen realisiert werden.

- Mit dem *Fallbeispiel* in Abschnitt 5 konnte letztendlich der praktische Beweis für das Funktionieren der *Statechart*-Grammatik als visueller Ableitmechanismus angetreten werden.

- Der Beginn der Entwicklung einer *Software zur automatischen Übersetzung* der o. g. Modelle realisiert einen großen Teil der visuellen Transformationsregeln und liefert bei Fertigstellung als Ergebnis unter *Ascet-SD* simulierfähige Zustandsautomaten.

Die vorliegende Arbeit leistet einen wesentlichen Beitrag zur Integration des Produktes *Ascet-SD* im Projekt *STEP-X*. Durch sie wird die Benutzung von mit *Artisan RtS* erstellten *Statecharts* durch Interpretation mit Zustandsautomaten möglich und somit auch die Erzeugung äußerst effektiven Programmcodes aus den entworfenen Modellen mit *Ascet-SD*. Die Optimierung von *Ascet-SD* fasst alle Veränderungen an Modellen zusammen,

die einen effektiveren Programmcode sowohl unter Zeit- und als auch unter Speicherplatzaspekten zur Folge haben und erlaubt auf diese Weise eine Verbesserung des Entwurfablaufs; z. B. versprechen hochoptimierte Codegeneratoren zwischenzeitlich für die Effizienzsteigerung einen Faktor von 1,3 selbst bei komplexen Aufgabenstellungen im Motormanagement [ETA02].

Im Rahmen des Projektes sind weitere Arbeiten zur Analyse und Diagnose von *Statecharts* vorgesehen. Dabei ist ein interessanter Ansatz, die Verhaltensweisen der Quell-*Statecharts* mit den transformierten Zustandsautomaten auf Äquivalenz zu untersuchen. Weiteren Anlass zu Arbeiten im Umfeld der vorliegenden Arbeit bietet die Software zur automatischen Übersetzung von *Statecharts*. Die Regeln, die unter Abschnitt 4.3 noch keinen Eingang in das Programm gefunden haben, können umgesetzt und implementiert werden. Des Weiteren wäre die Aufnahme des *Bound*-Attributs für einen Synchronisationszustand der *Statecharts* in *Artisan RtS* denkbar. Der Ansatz, Übersetzungen mit der in dieser Arbeit entwickelten Software voranzutreiben, ist nur eine und ungleich proprietäre Art der Transformation. Konzepte zur Graphtransformation bieten Graphtransformationssysteme wie *PAGG* [Göt83] oder *PROGRES* [Zün95, SWZ97]. Interessant wäre sicherlich die Anpassung der Regeln dieser Arbeit an solch ein System. Allerdings müssten beim Einsatz dieser Werkzeuge zusätzlich Schnittstellen zu *Artisan RtS* bzw. *Ascet-SD* geschaffen und entsprechende Regeln für Transformationen in diesen Systemen implementiert werden. Der wesentlich größere Aufwand dafür sollte jedoch erst nach einer gründlichen Nutzenanalyse in Betracht gezogen werden; es liegt jedoch die Vermutung nahe, dass die derzeitige Lösung des Transformationsproblems die optimalere darstellt.

Anhang A

Komponenten für die Modelltransformation

Tafel A.1: Komponenten und Eigenschaften von Statecharts und Zustandsautomaten

Eigenschaften	*Artisan RtS*	*Ascet-SD*	eingeschränktes *Statechart*
Anweisungen			
Zuweisung	•	•	•
Unterbrechungsanweisung	–	•	–
Methodenaufruf	•	•	–
CompositeState			
orthogonale Hierarchie	•	–	•
serielle Hierarchie	•	•	•
Event			
CallEvent	•	–	•[a]
ChangeEvent	•	○[b]	•[a]
CompletionEvent	•	–	•[a]
SignalEvent	•	–	•[a]
TimeEvent	•	–	•[a]
Besonderheiten			
Operationen auf Event	–	–	–
FinalState	•	–	•
Guard	•	•	•
Besonderheiten			
Guards haben *C*-Syntax	•	•	•

Fortsetzung auf der nächsten Seite

[a] Übersetzung im Namen verankert
[b] nur Timeout

Fortsetzung der letzten Seite

Eigenschaften	*Artisan RtS*	*Ascet-SD*	eingeschränktes *Statechart*
Pseudostate			
initial	●[a]	○[b]	●
deepHistory	●	–	●
shallowHistory	●[c]	○[d]	●
join	●	–	●
fork	●	–	●
junction	–	●	–
choice	●	–	●
SimpleState	●	●	●
State			
entry	●	●	●
exit	●	●	●
doActivity	●	●	●
internalTransition	●	–	●
Stubstate	–	●	–
SubmachineState	●	●	●
Synchstate	●	–	●
Transition			
trigger	●	●	●
guard	●	●	●
effect	●	●	●
Besonderheiten			
negative Trigger	–	–	–
e/a	●	●	●
$e[c]/a$	●	●	●
$[c]/a$	●	–	●
Transition zwischen Basiszuständen	●	●	●
Transition von und nach Hierarchiezuständen	●	●	●
Transition zwischen Zuständen versch. Hierarchien	●	●	●
Transition als Schleife	●	●	●
Priorität für Transition	○[e]	●[f]	●

Fortsetzung auf der nächsten Seite

[a] mit Aktion
[b] ohne Label
[c] Transition von/nach Pseudostate
[d] keine Transition hinein/heraus
[e] implizit
[f] explizit

Fortsetzung der letzten Seite

Eigenschaften	*Artisan RtS*	*Ascet-SD*	eingeschränktes *Statechart*
zwischen Basiszuständen	●	●	●
zwischen Hierarchiezuständen	●$^{\uparrow}$	●$^{\downarrow}$	●$^{\downarrow}$
Prioritäten erforderlich	–	●	–
deterministische Transitionen möglich	○^{a}	●	●
Ereignisse in der Aktion	●	–	●
Variablen			
any	●	–	–
boolean	●	●	●
char	●	–	–
double	●	●	●
float	●	●	●
long	●	●	●
octal	●	–	●
short	●	●	●
unsigned long	●	●	●
unsigned short	●	●	●
Besonderheiten			
global	–	–	–
lokal	●	●	●
Verhalten			
Aktion/Aktivitäten			
Aktivität vorzeitig abbrechbar	○^{b}	●	●
Aktion/Aktivitäten haben *C*-Syntax	●	●	●
run-to-completion step	●	●	●
Zugriff asynchron/synchron	●/●	●/●	●/●
Hierarchie			
Initialisierung des *Statecharts*	●$^{\downarrow}$	○$^{\downarrow c}$	●$^{\downarrow}$
Eintrittsaktivität ausführen	●$^{\downarrow}$	●$^{\downarrow}$	●$^{\downarrow}$
Austrittsaktivität ausführen	●$^{\uparrow}$	●$^{\uparrow}$	●$^{\uparrow}$
statische Aktivität ausführen	●$^{\uparrow}$	●$^{\uparrow}$	●$^{\uparrow}$

Fortsetzung auf der nächsten Seite

$^{\uparrow}$ Auswertung von innen nach außen
$^{\downarrow}$ Auswertung von außen nach innen
^{a} durch Reihenfolge in der Klassendefinition
^{b} schlecht zu simulieren
^{c} keine Ausführung von Eintrittsaktivitäten

Fortsetzung der letzten Seite

Eigenschaften	*Artisan RtS*	*Ascet-SD*	eingeschränktes *Statechart*
interne Transition ausführen	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$
Übergänge prüfen	$\bullet^{\downarrow}$	$\bullet^{\downarrow}$	$\bullet^{\downarrow}$

Literaturverzeichnis

[Alt] *Altia Design*. www.altia.com.

[AY98] Alur, Rajeev und Mihalis Yannakakis: *Model Checking of Hierarchical State Machines*. In: *Foundations of Software Engineering*, Seiten 175–188, 1998.

[BA77] Bobrow, L. S. und M. A. Arbib: *Discrete Mathematics*. Saunders, Philadelphia, 1977.

[Bjö01] Björklund, Dag: *The SMDL Statechart Description Language: Design, Semantics and Implementation*. Diplomarbeit, Åbo Akademi University, TUCS Turku Centre for Computer Science, 15th November 2001.

[BLA+99] Behrmann, Gerd, Kim Guldstrand Larsen, Henrik Reif Andersen, Henrik Hulgaard und Jorn Lind-Nielsen: *Verification of Hierarchical State/Event Systems Using Reusability and Compositionality*. In: *Tools and Algorithms for Construction and Analysis of Systems*, Seiten 163–177, 1999.

[BW95] Barr, Michael und Charles Wells: *Category Theory for Computing Science*. Prentice Hall Int., zweite Auflage, 1995.

[Doo] *Telelogic DOORS*. www.telelogic.com.

[EEKR97] Ehrig, H., G. Engels, H.J. Kreowski und G Rozenberg (Herausgeber): *Handbook of Graph Grammars and Computing by Graph Transformation, Volume II: Applications, Languages and Tools*. World Scientific, 1997.

[Ehr79] Ehrig, Hartmut: *Introduction to the Algebraic Theory of Graph Grammars*. In: V. Claus, H. Ehrig und G. Rozenberg (Herausgeber): *1st Graph Grammar Workshop*, Lecture Notes in Computer Science, Seiten 1–69, Berlin, 1979.

[EKR91] Ehrig, H., H.-J. Kreowski und G. Rozenberg (Herausgeber): *4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, Band 532 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[ENR83] Ehrig, H., M. Nagl und G. Rozenberg (Herausgeber): *2nd Int. Workshop on Graph Grammars and Their Application to Computer Science*, Band 153. Springer Verlag, 1983.

[EPS73] Ehrig, H., M. Pfender und H. Schneider: *Graph grammars: an algebraic approach*. In: *14th Annual IEEE Symposium on Switching and Automata Theory*, Seiten 167–180. IEEE, 1973.

[ER97] Engelfriet, J. und G. Rozenberg: *Node Replacement Graph Grammars*. In: Rozenberg, Grzegorz [Roz97].

[ETA02] ETAS GmbH: *ASCET-SD V4.2 Benutzerhandbuch*. Stuttgart, 2002.

[GGN91] Göttler, H., J. Günther und G. Nieskens: *Use Graph Grammars to Design CAD-Systems*. In: Ehrig, H. et al. [EKR91], Seiten 396–410.

[Gog89] Goguen, J. A.: *The categorial manifesto*. Technischer Bericht PRG-27, Oxford University Computing Laboratory, 1989.

[Göt83] Göttler, H.: *Attribute Graph Grammars for Graphics*. In: Ehrig, H. et al. [ENR83], Seiten 130–142.

[Göt88] Göttler, H.: *Graphgrammatiken in der Softwaretechnik*. IFB 178. Springer Verlag, 1988.

[GPP98] Gogolla, Martin und Francesco Parisi-Presicce: *State Diagrams in UML: A Formal Semantics using Graph Transformations*. In: Broy, Manfred, Derek Coleman, Tom S. E. Maibaum und Bernhard Rumpe (Herausgeber): *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[GS84] Gécseg, F. und M. Steinby: *Tree Automata*. Akadémiai Kiadó, 1984.

[GS97] Gécseg, F. und M. Steinby: *Tree Languages*. In: Rozenberg, G. und A. Salomaa (Herausgeber): *Handbook of Formal Languages*, Band 3: Beyond Words, Kapitel 1, Seiten 1–68. Springer Verlag, 1997.

[Har87] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 8:231–274, 1987.

[HG96] Harel, David und Eran Gery: *Executable Object Modeling with Statecharts*. In: *Proceedings of 18th Int. Conf. on Software Engineering*, Seiten 246–257, Berlin, 1996. IEEE Computer Society Press.

[HP88] Hatley, Derek und Imtiaz A. Pirbhai: *Strategies for Real-Time-System Specification*. Dorset House Publishing Co., Inc., NY, 1988.

[HW95] Heckel, R. und A. Wagner: *Ensuring Consistency of Conditional Graph Grammars – A Constructive Approach*. In: *Proc. of SEGRAGRA '95 "Graph Rewriting and Computation"*, Band 2. Electronic Notes of TCS, 1995.

[HW98] Heimdahl, Mats P. E. und Michael W. Whalen: *Reduction and Slicing of Hierarchical State Machines*. Technischer Bericht, Institute of Technology, Department of Computer Science, University of Minnesota, 1998.

[Joh99] John, Sebastian: *Zur kompositionalen Semantik von objekt-orientierten Statecharts*. Diplomarbeit, Technische Universität Berlin, 1999.

[KHC+99] Kim, Y. G., H. S. Hing, S. M. Cho, D. H. Bae und S. D. Cha: *Test Cases Generation from UML State Diagrams*. In: *IEE Proceedings – Software*, Band 146, Seiten 187–192, Taejon, Korea, Aug. 1999.

[Kre87] Kreowski, H.-J.: *Part 1: Derivations in graph grammars*. In: *Is parallelism already concurrency?*, Lecture Notes in Computer Science, Seiten 343–360. 1987.

[KW87] Kreowski, H.-J. und A. Wilharm: *Part 2: Non-sequential processes in graph grammars*. In: *Is parallelism already concurrency?*, Lecture Notes in Computer Science, Seiten 361–377. 1987.

[LKW93] Löwe, M., M. Korff und A. Wagner: *An Algebraic Framework for the Transformation of Attributed Graphs*. In: Sleep, M., M. Plasmeijer und M. van Eekelen (Herausgeber): *Term Graph Rewriting: Theory and Practice*, Kapitel 14, Seiten 185–199. John Wiley & Sons Ltd., 1993.

[Löw93] Löwe, M.: *Algebraic Approach to Single-Pushout Graph Transformation*. Theoretical Computer Science, 109(1,2):181–224, 1993.

[LP99] Lilius, Johan und Iván Porres Paltor: *The Semantics of UML State Machines*. TUCS 273, Åbo Akademi University, Turku, 1999.

[Nag79] Nagl, Manfred: *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, Braunschweig, Wiesbaden, 1979.

[OMG01] OMG: *Unified Modeling Language Specification, Version 1.4*. `www.omg.org`, September 2001.

[Pie03] Pietsch, Steffen: *Automatische Überprüfung von UML-Statecharts anhand definierbarer Design-Regeln*. Diplomarbeit, Institut für Software, Abteilung Programmierung, Technische Universität Braunschweig, 2003.

[Plu95] Plumb, D.: *On terminiation of Graph Rewriting*. In: *21st Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, 1995.

[PMP01] Pap, Zsigmond, István Majzik und András Pataricza: *Checking General Safety Criteria on UML Statecharts*. Lecture Notes in Computer Science, 2187:46ff., 2001.

[PR69] Pfaltz, J. L. und A. Rosenfeld: *Web Grammars*. In: *Int. Joint Conference on Artificial Intelligence*, Seiten 609–619, 1969.

[PS91] Pnueli, A. und M. Shalev: *What is in a Step: On the Semantics of Statecharts*. Lecture Notes in Computer Science, 525:244–264, 1991.

[Roz97] Rozenberg, Grzegorz (Herausgeber): *Handbook of Graph Grammars and Computing by Graph Transformation, Volume I: Foundations*. World Scientific, 1997.

[Sal78] Salomaa, A.: *Formale Sprachen*. Springer, Berlin, 1978.

[Sim00] Simons, Anthony J. H.: *On the Compositional Properties of UML Statechart Diagrams*. Technischer Bericht, Department of Computer Science, University of Sheffield, 2000.

[SWZ97] Schürr, A., A. J. Winter und A. Zündorf: *The PROGRES Approach: Language and Environment*. In: Ehrig, H. et al. [EEKR97].

[vdBMS] Beeck, Michael von der, Tiziana Margaria und Bernhard Steffen: *A Formal Requirements Engineering Method for Specification, Synthesis, and Verification*.

[Zün95] Zündorf, A.: *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. Doktorarbeit, RWTH Aachen, 1995.