

Prof .Dr. Wirsing, Harald Störrle, Alexander Knapp

Wintersemester 2000/2001

19. Januar 2001

Bearbeitet von: Konstantin Kirsch

## Aspekte der Semantik von UML: UML State Machines

### Zusammenfassung:

Ziel dieser Ausarbeitung ist es die Semantik der UML State Machines näher zu bringen. Dabei werden zwei spezielle Ansätze vorgestellt, analysiert und kritisiert, die sich zum einen damit befassen eine eindeutige Formalisierung der UML State Machine Semantik zu definieren und zum anderen eine konkrete Überführung der UML Statecharts in eine input language eines Model-Checkers zu vollbringen.

## Inhaltsverzeichnis

Aspekte der Semantik von UML: UML State Machines .....	1
1 Einführung - Motivation.....	2
1.1 UML State Machines .....	2
2 Formalisierung der UML State Machine Semantik .....	4
2.0 Verschiedene Ansätze .....	4
2.1 Erster Ansatz von Johan Lilius und Iván Porres Paltor:.....	5
2.2 Zweiter Ansatz von Gihwon Kwon:.....	8
3 Diskussion – Kritik.....	11
3.1 Eine Tabelle die einen Vergleich bringt soweit das möglich ist.....	11
3.2 Beschreibung zur Tabelle.....	11
3.3 Bewertung .....	12
4 Schlusswort .....	13
5 Glossar.....	13
6 Appendix .....	14
7 Referenzen und Literatur.....	15

# 1 Einführung - Motivation

## 1.1 UML State Machines

Eine UML State Machine ist ein dynamisches Modell, was bei der objekt-orientierten Programmierung eingesetzt wird. Nach dem Standard hat jede Klasse ihre eigene State Machine, die das Verhalten ihrer Instanzen beschreibt, die sogenannten Objekte. Die State Machine erhält Nachrichten von ihrer Umgebung und reagiert entsprechend auf sie.

Eine State Machine besteht aus folgenden Komponenten:

- Aus einer event queue die einkommende Ereignis Instanzen aufbewahrt, bis sie abgearbeitet werden.
- Aus einem event dispatcher mechanism der aus der event queue Ereignis Instanzen selektiert und entnimmt um sie dem event processor zu überreichen.
- Und aus einem event processor der die Ereignis Instanzen bearbeitet und zwar genau nach der allgemeinen Semantik der UML State Machines und nach der speziellen Form der aktuellen State Machine.

**Ereignisse (Events)** sind grundlegend für Statecharts. Event Instanzen oder auch Ereignisinstanzen werden immer dann generiert, wenn eine Aktion innerhalb des Systems oder auch von der System-Umgebung stattfindet. Ein Event wird an ein oder mehrere Ziel übertragen. Events werden immer einzeln von der State Machine abgearbeitet.

**Transitionen** sind passive Elemente eines Statecharts, die die Struktur des Statecharts widerspiegeln. Jede Transition besitzt fünf Eigenschaften:

Einen Event-Trigger, der festlegt, bei welchem Ereignis ein Zustandswechsel entlang dieser Transition stattfindet.

Einen Quellzustand, der festlegt, aus welchem Zustand ein Zustandswechsel über diese Transition stattfinden kann. Eine Transition wird immer nur dann benötigt, wenn der aktuelle Zustand ihr Quellzustand ist.

Einen Zielzustand, der festlegt, wer der neue aktuelle Zustand der StateMachine wird, wenn ein Zustandswechsel über diese Transition stattfindet.

Eine Guard-Condition, d.h. ein logischer Ausdruck, der logisch wahr ergeben muss, damit ein Zustandswechsel über diese Transition erfolgen kann. Defaultmässig sind alle Guards auf true gesetzt.

Eine Aktion, die während des Zustandswechsels ausgeführt wird. (Diese Aktion ist atomar, dass heisst sie kann nicht unterbrochen werden.)

**Zustände (states):** sind wie Transitionen passive Elemente eines Statecharts, die ebenfalls die Struktur des Statecharts widerspiegeln. Zustände besitzen folgende Bestandteile:

Einen Namen, Eintritts- und Austrittsaktionen (Entry und Exit), interne Zustandsübergänge, Teilzustände (Substates), verzögerte Ereignisse, eine nicht-atomare Aktivität.

Es gibt die klassischen Statecharts, auch Harel Statecharts genannt, und die Object State Machines. Die beiden Konzepte unterscheiden sich wesentlich und der wahrscheinlich wichtigste Unterschied ist:

Im UML Standard wird nicht die Reihenfolge der auszuführenden orthogonalen Regionen im RTC Step ermittelt sondern sie bleibt immer sequentiell.

In den Harel Statecharts werden alle Events, die in einem Schritt generiert wurden gleichzeitig auch dem Empfänger zur Verfügung gestellt.

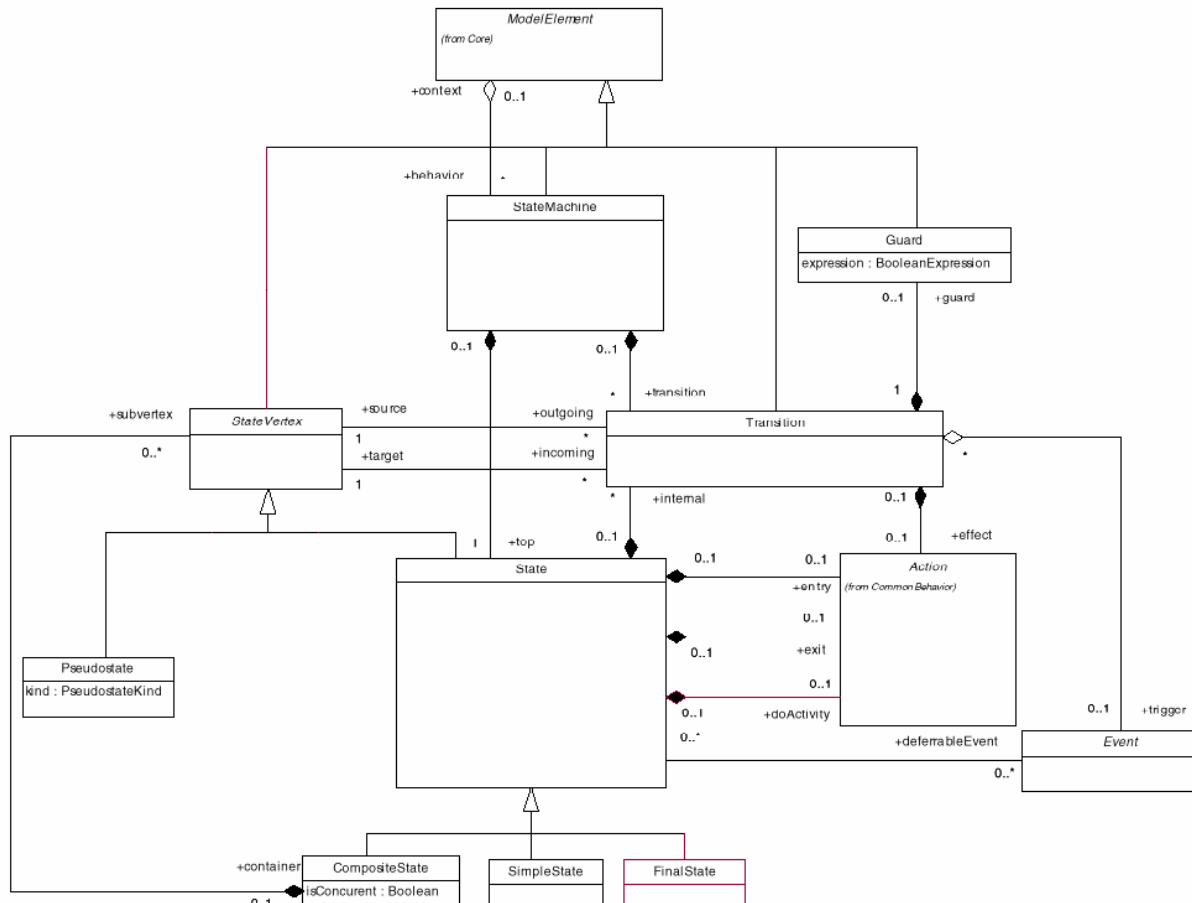
Generell wird von Lilius vorgeschlagen Orthogonality zu vermeiden, da es oft nicht eindeutig ist, in welchen Zustand Objekte geraten.

Wichtig ist noch zu wissen, dass State Machines dazu genutzt werden können, um das Verhalten vieler Elemente die modelliert werden, zu beschreiben. Zum Beispiel kann man das Verhalten von individuellen Events (class instances) modellieren oder auch die Interaktion (collaboration) zwischen den Entities definieren.

Nun eine kleine Zusammenfassung zu den State Machines bezüglich Syntax und Semantik:

- a) Das State Machine Metamodell und seine abstrakte Syntax

Die Basis-Konzepte von State Machines, wie z.B. States, Transitions und Events sind folgendermassen aufgebaut:

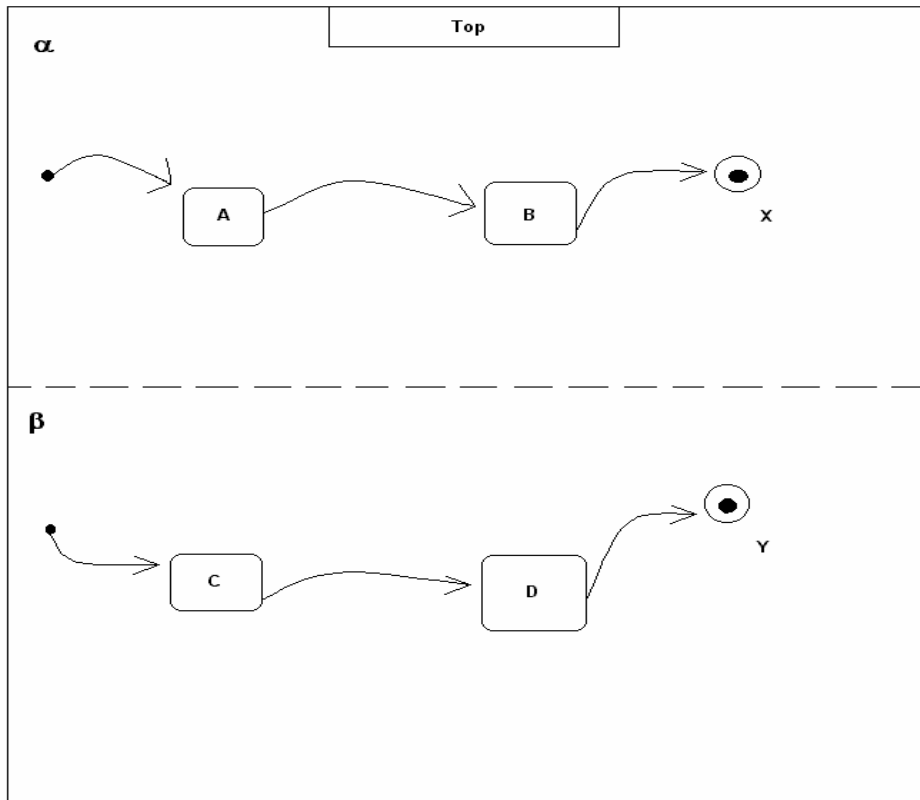


Nun folgt ein Beispiel in dem genau alle States aufgezählt werden um den Unterschied zwischen Syntax und Semantik zu veranschaulichen.

Aus syntaktischer Sicht sind es 11 Zustände:

Top + initialA + A + B + x + initialB + C + D + y +  $\alpha$  +  $\beta$

Aus semantischer Sicht =  $sem(\alpha) * sem(\beta) = 3 * 3 = 9$  Zustände



## 1.2 Model Checking

Unter **Model Checking** versteht man Verfahren zur automatischen Analyse *reaktiver Systeme*, das sind Systeme, die immer wieder mit ihrer Umgebung interagieren (im Gegensatz zu Algorithmen, die aus gegebenen Eingaben ein Ergebnis berechnen und dann terminieren). Ein typisches Beispiel ist die Prozeßsteuerung (z.B. Airbagauslösung im Auto).

## 2 Formalisierung der UML State Machine Semantik

### 2.0 Verschiedene Ansätze

In der Praxis wurden einige Ansätze gemacht, die UML State Machine Semantik formal aufzufassen. Bei solchen Ansätzen geht es hauptsächlich darum UML Statecharts zu „model checken“.

Einige Ansätze die sich damit befasst haben die klassischen „Harel Statecharts“ und die „UML Statecharts“ zu model checken sind:

Author	Model Checker	Formalism	Statechart-Art
Day, 1993	Voss	Higher-order logic	Harel's
Mikk, 1998	SPIN	Hierarchical automata	Harel's
Damm, 1998	SIEMENS AG	Asynchronous Transition Algorithm	Harel's
Latella, 1999	SPIN	Hierarchical automata	Object State Machine
Gnesi, 1999	JACK	Hierarchical automata	Object State Machine
Lilius, 1999	SPIN	Rewrite Rules	Object State Machine
Heinle, 2000	SMV	Extended temporal logic	Harel's

Hier werden wir nur zwei konkrete Ansätze besprechen. Der eine Ansatz ist von Lilius und Paltor und versucht UML Statecharts vollständig zu formalisieren.

## 2.1 Erster Ansatz von Johan Lilius und Iván Porres Paltor:

### 2.2.0

In diesem Ansatz werden Statechart Diagramme in Terme überführt mit Hilfe von eigenen eingeführten Rewrite Rules. Dann wird noch behauptet, dass es nach diesem Ansatz möglich ist, in linearer Zeit, UML Statecharts zu Model – Checken.

### 2.1.1 SPIN, vUML, PROMELA

Eine Grafik veranschaulicht, wie vUML funktioniert. Dazu wird der schon oben vorgestellte Model-Checker SPIN und seine Eingabesprache PROMELA (= PROTOCOL META LANGUAGE) verwendet. Zuerst wird das UML-Modell in PROMELA übersetzt, die wiederum an den Model-Checker SPIN weitergereicht wird. SPIN analysiert das eingegebene System und im Falle eines Fehlers wird ein sogenanntes counter-example produziert, auch als Fehler – Pfad zu verstehen. Dieses counter-example wird in ein UML Sequenzdiagramm übersetzt und wird dem Benutzer übergeben.

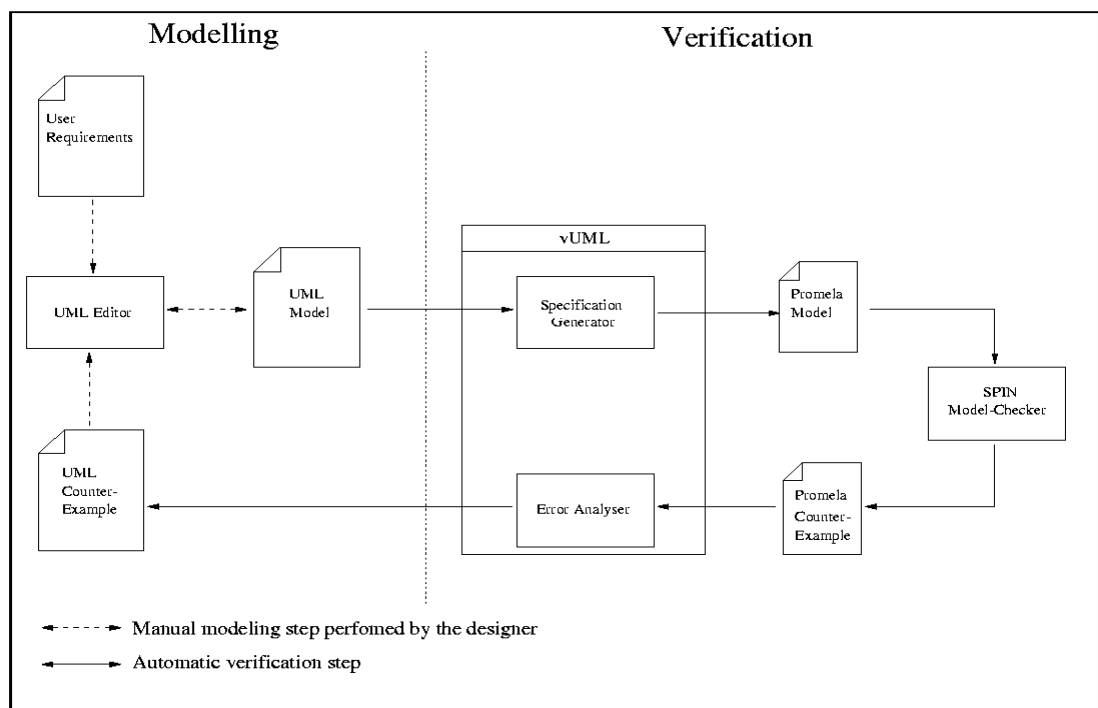


Figure 1: Using vUML

Der Vorteil dabei ist, dass der Benutzer sich weder mit SPIN noch mit PROMELA auskennen muss. Das einzig notwendige ist, dass der Benutzer genau weiss, was für Fehler vUML abfangen kann. Diese Fehler sind:

- i) **Verklemmungen**
- ii) **Ein ungültiger Zustand (invalid State) wird erreicht**

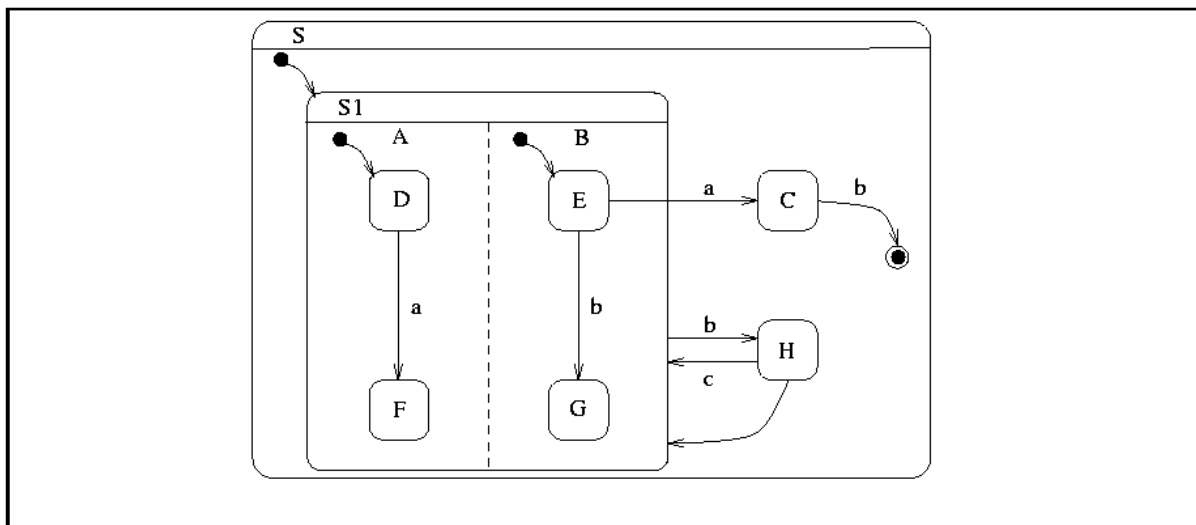
- iii) Constraints eines Objektes werden verletzt
- iv) Ein Event wird an ein terminiertes Objekt verschickt
- v) Überlauf der Input Queue eines Objektes über
- vi) Überlauf der deferred Event Queue
- vii) livelocks

## 2.1.2 FORMALE BESCHREIBUNG DER STATE MACHINES

Nach Lilius können die Zustände  $\Sigma$ , einer State Machine SM, in 5 disjunkte Mengen (sets) aufgeteilt werden: compound states  $\Sigma_{cs}$  sind die zusammengesetzten Zustände, simple states  $\Sigma_{ss}$ , final states  $\Sigma_{fs}$ , synch states  $\Sigma_{syms}$ , history states  $\Sigma_{his}$ .

Es gilt also:  $\Sigma = \Sigma_{cs} \cup \Sigma_{ss} \cup \Sigma_{fs} \cup \Sigma_{syms} \cup \Sigma_{his}$

Zusätzlich gibt es noch initial states  $\Sigma_{is}$ , wobei  $\Sigma_{is}$  ein subset von  $\Sigma_{cs} \cup \Sigma_{ss}$ .



An example statechart

Nach obigem Beispiel haben wir:

$$\Sigma_{ss} = \{ C, D, E, F, G, H \}$$

$$\Sigma_{cs} = \{ S, S_1, A, B \}$$

$$\Sigma_{is} = \{ S, S_1, D, E \} \quad \text{und} \quad \Sigma_{syms} = \Sigma_{his} = \emptyset$$

Es wird noch eine Funktion arity definiert,  $arity: \Sigma_{cs} \cup \Sigma_{ss} \cup \Sigma_{syms} \rightarrow \mathbb{N}$

Wichtig ist nur zu wissen, dass diese Funktion, die Anzahl der orthogonalen Regionen für jedes State abbildet. (Siehe Tabelle)

s	S	S <sub>1</sub>	A	B	C	D	E	F	G	H
arity(s)	1	2	1	1	0	0	0	0	0	0

Sei X eine abzählbare Reihe an Variablen. Mit  $\Sigma$  als Reihensymbol, können wir die Reihe der linearen Terme  $T_{\Sigma}(X)$  bilden:

1.  $\Sigma_{ss} \cup X \subseteq T_{\Sigma}(X)$
2.  $op(t_1, \dots, t_n) \in T_{\Sigma}(X)$ , where  $op \in \Sigma_{cs}$ ,  $n = arity(op)$  and  $t_1, \dots, t_n \in T_{\Sigma}(X)$
3.  $\forall t \in T_{\Sigma}(X), \forall x \in Var(t), \#(x, t) = 1$

**Definition:** Eine **State Configuration** einer State Machine ist ein Term in  $T_\Sigma$ .  
 Jedes State hat analog zum Standard eine entry, exit und activity action. Ein State hat auch interne Transitionen.  
 Transitionen werden analog zum Standard definiert. Es wird nur eine andere Notation verwendet. Es wird ein Triple  $(s, t, s')$  mit  $s, s'$  aus  $T_\Sigma$  und  $t$  aus  $T_n$ .  
 Eine UML State Machine ist dann ein Tupel  $SM = \langle \Sigma, \text{entryAction}, \text{exitAction}, \text{activity}, \text{deferred}, \text{bound}, \text{defaultHistory}, \text{deep}, \Phi, \text{source}, \text{target}, \text{trigger}, \text{guard}, \text{effect} \rangle$

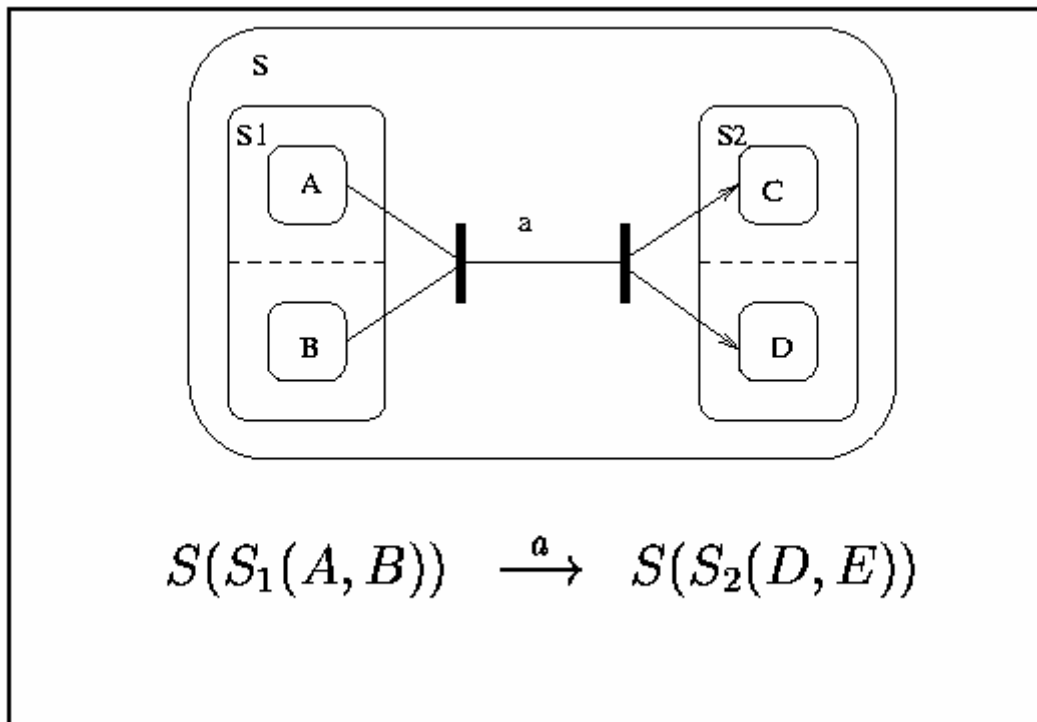
**Ein Vorteil:**

Während der UML Standard weitere **Pseudostates** definiert wie z.B. initial, join vertices, fork vertices und junctions, wird das in diesem Ansatz anders gelöst. Die Pseudostates werden im UML Standard dazu genutzt, um mehrfache Transitionen miteinander zu verbinden und komplexere State Transition Pfade zu bilden. Diese Erweiterungen werden nur deshalb eingeführt, weil die primitiven Konstrukte nicht ausreichen solch ein Verhalten auszudrücken. Es ist jedoch gelungen diese Pseudostates direkt auszudrücken:

Initial States:

Join und Fork Vertices:

Junction Vertices:



Der Run-To-Completion Step ist genauso definiert wie im Standard. Im Standard wurde jedoch keine Semantik zur event queue definiert. Hier wird die queue als ADT konzipiert mit Operationen enqueue und dequeue von Ereignissen. So kann man herausfinden, welche Transitionen aktiviert sind.

Dann einige mathematische Begriffe eingeführt für: aktive transitions, mode einer transition, eine Transitionsinstanz, enabled active transition.

Der Konflikt zwischen Transitionen wird auch eingeführt durch eine Operation:

conflict  $(t, t', s)$ . Die Operation wird durch weitere Operationen definiert. Wichtig ist an dieser Stelle zu erwähnen, dass ein Konflikt manchmal aufgelöst werden kann, wenn man die **priority relation** beachtet:

Falls  $t_1$  eine Transition ist mit  $\text{source}(t_1) = s_1$  und  $t_2$  eine Transition ist mit  $\text{source}(t_2) = s_2$  dann:

- Falls  $s_1$  ein direktes oder transitiv enthaltenes substate von  $s_2$ , dann hat  $t_1$  höhere Priorität als  $t_2$
- Falls jedoch  $s_1$  und  $s_2$  nicht in der selben State Configuration sind, dann gibt es keinen Prioritätsunterschied.

### Noch ein Vorteil:

Dieser Ansatz ist einer der wirklich wenigen, der alle Features von UML beachtet. Lilius / Paltor haben versucht alle möglichen, vom Standard eingeführten Konstrukte, in ihrem Konzept nicht nur zu besprechen, sondern auch eine direkte Lösung anzugeben.

### Nachteile:

Obwohl dieser Ansatz eine Anstrengung macht alle Features der UML State Machines wie sie im UML Standard definiert wurden abzudecken, gibt es jedoch einige Mankos:

- a) Es ist leider nicht möglich neue Objekte zu kreieren. Das kann zu inkonsistentem und undefiniertem Verhalten führen, welches auch unmöglich mit einem Model-Checker automatisch verifiziert werden kann.
- b) Es werden sehr viele theoretische Aspekte vorgestellt, die an sich sehr logisch wirken. Die Argumente und die eingeführten Definitionen sind alle vollständig. Es wird jedoch nicht beschrieben, wie man konkret ein Statechart Diagramm in eine geeignete Eingabe Sprache (wie PROMELA) eines Modell Checkers übersetzt.
- c) Beim zweiten Algorithmus (FullRTC) gibt es eine Ausnahme in Zeile 29, die dazu führen kann, dass der RTC blockiert werden kann, im Falle, dass man auf eine Antwort eines angesprochenen call events wartet, welches aber keine zurückliefert.

## 2.2 Zweiter Ansatz von Gihwon Kwon:

### 2.2.1 Der Ansatz von Kwon als Ergänzung zum Vorgehen von Lilius

Dieser zweite Ansatz versucht den ersten Ansatz zu vervollständigen, indem eine Ergänzung eingeführt wird: Lilius transformiert zwar die UML Statechart Diagramme mit Hilfe gewisser Rewrite Rules, spricht aber nie genau davon wie man so ein Diagramm konkret in eine Eingabesprache eines Model Checkers überführt.

Kwon geht in drei wesentlichen Schritten vor:

Zuerst werden seine sogenannten „normalisierten“ Statecharts als input charakterisiert.

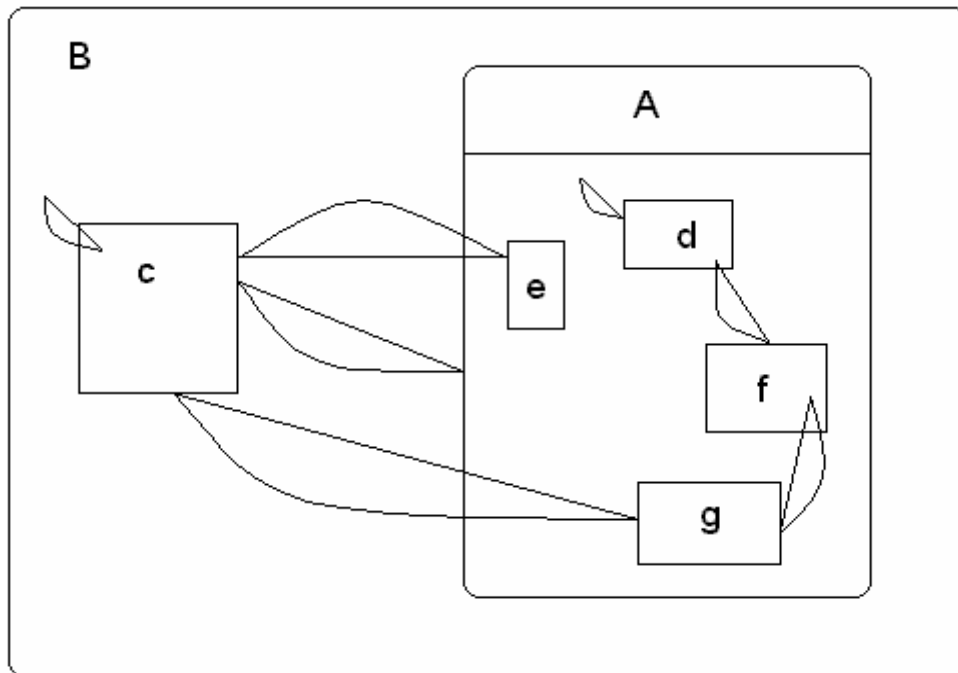
Denn definiert er eine operationale Semantik die seine Rewrite Rules charakterisiert.

Und zuletzt werden die Statecharts in die SMV input language konvertiert.

Kwon hat sich entschlossen **SMV** als Model Checker zu benutzen und UML Statecharts in die input language von SMV zu übersetzen. Die Übersetzung basiert auf ein Term Rewriting System das weiter unten erklärt wird.

SMV wird aus folgenden Vorteilen als (Target) Model Checker gewählt:

- 1) SMV kann mit synchronen und asynchronen Systemen umgehen.
- 2) SMV deckt ein grosses Spektrum an Systemen ab, von abstrakten bis zu konkreten Systemen, von deterministischen bis zu nicht deterministischen Systemen.
- 3) SMV kommt mit vielen Zuständen klar (state space) mehr als  $10^{100}$  Zuständen. Dazu benutzt sie symbolische model checking Techniken mit der Hilfe von BDD (Binary Decision Diagram) Repräsentationen. Das sind effiziente Datenstrukturen und werden oft bei partiellen binären Strukturen eingesetzt, wobei UND -, ODER -, NICHT - Operationen eingesetzt werden. Um den Umfang jedoch nicht zu sprengen, wollen wir hier auf BDD nicht näher eingehen.



Ein Beispiel zum Statechart von Kwon

### Allgemeine Beschreibung des Vorgehens von Kwon:

Zuerst wird auf **normalisierte Statecharts** eingeschränkt, damit das Term Muster regulär bleibt. Eine **State Configuration** wird hier als Term kodiert, da States als Funktionen angesehen werden können, die Subcharts als Argumente übergeben bekommen. Terme sind nach Kwon entweder individuelle Variablen oder individuelle Konstanten oder eine Funktion  $f$ , der man  $n$  Argumente übergeben kann. Falls man der Funktion  $f$  die Terme  $t_1, \dots, t_n$  übergibt, ist dann  $f(t_1, \dots, t_n)$  wieder ein Term.

Dann definiert sich Kwon sogenannte **Transition Labels**, die das Verhalten einer Transition beschreiben. Ein solches Label besteht aus einem **5-Tupel (src, evt, con, act, tgt)**, wobei src ein Quellzustand, evt ein Auslösereignis (trigger event), con die Kondition, act eine Aktion, tgt der Zielzustand ist. Transitionen werden unter folgenden Bedingungen aktiviert:

src gehört zur aktuellen Konfiguration (current configuration)

evt passiert (in dem Moment) und con wartet (befindet sich im Wartezustand).

Aktiviert Transitionen werden letztendlich ausgeführt, wenn keine Konflikte auftreten. Die Konsequenzen bei der Ausführung von Transitionen sind:

Die Aktion act wird ausgeführt und der Zielzustand tgt gehört zur nächsten Konfiguration, während der Quellzustand src in der nächsten Konfiguration nicht mehr aktiv ist.

Transitionen sind sehr ähnlich zum UML Standard definiert worden.

Hier die **Rewrite Rules** zum obigen Diagramm:

$r_2: [] B-c(c) \rightarrow B-c(d) \{ \}$

$r_3: [] B-c(d) \rightarrow B-c(e) \{ \}$

$r_4: [u] A-c(b(B-c(X))) \rightarrow A-c(a) \{ \}$

$r_5: [] A-c(b(B-c(e))) \rightarrow A-c(a) \{ \}$

Labels wie  $r_2$  werden nur zur Bequemlichkeit der Referenzierung der Rewrite Rules benutzt.

Danach führt er **Rewriting Relations** ein. Zwei Terme haben eine solche Relation genau dann, wenn der Term  $t_j$  den Term  $t_i$  ersetzt, da irgend ein Rewrite Rule ausgeführt wird. Die Rewrite Relations werden dann durch 3 Regeln charakterisiert, in die ich an dieser Stelle nicht eingehe. Wichtig ist nur zu wissen, dass diese Regeln dazu gedacht sind, um Subterme zu handhaben, das sind Terme die in Termen enthalten sind, und damit im Falle das gleichzeitig mehrere Rewrite Rules ausgeführt werden, die Rewriting Relations definiert sind.

Weiterhin wird eine sogenannte map Struktur vorgestellt. Viele mathematische Operationen werden eingeführt, um dann den „step“ einzuführen. Danach befindet er sich in der Lage, die Semantik für Rewrite Rule Aktivierung zu beschreiben. Der entscheidene und interessante Teil seines Ansatzes kommt aber erst später, wenn dann wirklich konkret die UML Statecharts in die SMV input language mit Hilfe seiner eingeführten operationalen Semantik. Indem Kwon immer wieder auf seine operationale Semantik zurückgreift hat er die Möglichkeit viele Komponenten von Statecharts abzudecken.

Es werden genau folgende Fälle simuliert:

- Subchart activity
- Initialisierung von Statecharts (Startzustände)
- Aktivierung von Rewrite Rules
- State Transitions
- Der Spezialfall, dass ein Chart nur ein State hat
- History States
- Transition conflict

Zu beachten ist, das Kwon (obwohl er es in seiner Ausarbeitung sehr kompliziert formuliert hat) die **Konflikte zwischen Transitionen** völlig analog behandelt, wie es Lilius macht.

Ein kleines Beispiel hierzu:

Angenommen folgende Formel (nach der operationalen Semantik von Kwon) beschreibt das Verhalten von zwei Transitionen, wo aber schon eine Ordnung existiert.

$Conflict(r_1, r_2) \text{ und } r_1 \Phi r_2 \rightarrow (\text{nicht } r_1) \text{ und } r_2$

Diese Formel wird dann in SMV folgendermassen ausgedrückt:

DEFINE

cr2: = ! r1 & r2

wobei cr für einen konfliktfreien rewrite rule steht.

### Nachteile:

- a) Ein Nachteil ist es jedoch, dass Kwon bei der Übersetzung der Statechart Diagramme in die SMV input language keine hierarchischen Strukturen behandelt. Er begründet das ganze mit folgenden Argumenten:
  - i) Interlevel Transitionen (wie goto statements in Programmiersprachen) welche die klare Struktur der Hierarchie verletzen kommen sehr oft vor. (Er meint evtl. das es keine klaren hierarchischen Strukturen in der Praxis gibt, so wie sie im Standard angenommen werden.)
  - ii) Der scope rule von state Namen ist global.
  - iii) Man muss Konflikte zwischen aktivierten Transitionen auflösen, falls welche auftreten.
- b) Ein weiterer wesentlicher Nachteil dieses Ansatzes ist, dass nicht alle Features und Eigenschaften des UML Standards behandelt werden. Es werden zum Beispiel keine Composite States besprochen.

### 3 Diskussion – Kritik

#### 3.1 Eine Tabelle die einen Vergleich bringt soweit das möglich ist

UML Standard		Ansatz von Lilius/Paltor		Ansatz von Kwon	
Hierarchische Strukturen von Statecharts		unterstützt		nicht unterstützt	
State Configuration		wird analog zum Standard definiert, ist ein Term in $T\Sigma$		ist ein Term , weil States =Funktionen mit subcharts als Argumente	
Transitionen		Tupel (t,s,t') , selbe Semantik wie im Standard		transition labels (src, evt, con, act, tgt), selbe Semantik zu UML	
Pseudostates		werden nicht eingeführt, aber durch eigene Konstrukte simuliert		werden überhaupt nicht erwähnt	
RTC Assumption		wird in vUML implementiert hat jedoch einen Problemfall		wird nicht direkt angesprochen. Nicht nötig wegen Normalisierung.	
Transition Conflict		definiert durch conflict(t,t',s), mit priority relation gelöst		wie im Standard mit <b>priority</b> und <b>nondeterminism</b> gelöst	
Features der UML State Machines		alle Features werden behandelt		nicht alle werden behandelt (keine Pseudostates und composite States)	
Model Checker	input language	SPIN	PROMELA	SMV	SMV input language

#### 3.2 Beschreibung zur Tabelle

### 3.3 Bewertung

Meiner Meinung nach ist das Konzept von Lilius/Paltor sehr weit ausgereift, im Gegensatz zum Ansatz von Kwon.

Es werden klare Definitionen gebracht, die durchaus logisch sind.

Es wird auf **alle Features** der UML State Machines Rücksicht genommen.

Ein besonderer Pluspunkt ist die **Vermeidung von Pseudostates**.

Wesentliche Nachteile sind jedoch, dass **kein konkretes Verfahren** angesprochen wird, State Machines in eine Eingabe Sprache eines Model Checkers überführen zu können.

Zusätzlich auch die Tatsache, dass **keine neuen Objekte** erzeugt werden können und auch die mögliche Fehlerquelle der vollständigen Version des RTC Algorithmus, dass das der **Algorithmus bei Zeile 29 blockiert**.

Das Team ist sich auch über die Existenz allgemeiner Probleme, die der UML Standard mit sich bringt, bewusst. (sprich orthogonal regions und concurrent states) Deshalb vielleicht, ist dieser Ansatz ziemlich positiv wirkend.

Dass im 2. Ansatz **keine hierarchischen Strukturen** behandelt werden und es auch unmöglich ist diese ganzen Transformationsregeln und die eingeführte operationale Semantik bei solchen Strukturen einzusetzen ist ein elementarer negativer Punkt. Dieser eine Punkt reicht schon aus, um das ganze Konzept nicht ernst zu nehmen. Weiterhin werden viele, eigentlich intuitiv leicht verstehbare, Konstrukte viel **zu kompliziert** ausgedrückt und formuliert. Man sagt ja, wenn man die Wahl hat, zwischen zwei Algorithmen oder Lösungen eines selben Problems, dann nimmt man immer die einfachere. Es gibt weitere Ansätze auf diesem Gebiet, die bei weitem nicht so umständlich vorgegangen sind. Zusätzlich kommt noch hinzu, dass **nicht alle Features** bezüglich des UML Standards behandelt werden. Obwohl die Idee, die dahinter steckt - auf ein schon vorhandenes Konzept aufzubauen - sehr gut motivierend wirkt, ist der im Paper vorgestellte Ansatz noch nicht ausgereift genug.

## 4 Schlusswort

Nachdem erklärt wurde, wie man Statecharts sinnvoll in der Praxis einsetzen kann, sieht man, dass es immer noch schwer ist alle Features des UML Standards abzudecken. Die vorgestellten Ansätze sind noch nicht ausgereift genug, um sie bei wirklich komplexen Systemen einzusetzen, da es oft zu Inkonsistenzen und undefinierten Zuständen führt. Viele Fehler werden weiterhin beim wirklichen Einsatz des Software Systems auftreten, die dann durch mühsame Fehleranalyse entdeckt werden müssen.

Ein Model Checker kann natürlich dazu beitragen, aber so lange man das System nicht vollständig in eine geeignete input language (wie z.B. SMV oder PROMELA) übersetzen kann, ist es nicht möglich eine automatische Verifikation durchzuführen. Man sieht jedoch, dass schon grosse Schritte in diese Richtung gemacht wurden und das es bestimmt in der Zukunft möglich sein wird, durch verschiedene Werkzeuge, die Arbeit eines Software Entwicklers zu vereinfachen.

## 5 Glossar

**state configuration** = When dealing with composite and concurrent states, the simple term „current state“ can be quite confusing. In a **hierarchical state Machine** more than one state can be active at once. If the state Machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active „state“ is actually represented by a tree of states starting with the single top state at the root down to individual simple states at the leaves. We refer to such a state tree as a **state configuration**.

**run-to-completion processing** = means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed.

**run-to-completion step** = the processing of a single event by a state Machine. It is the passage between two state configurations of the state Machine

**Transition selection algorithm** = The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

- All transitions in the set are enabled
- There are no **conflicting transitions** within the set (Wenn zum Beispiel 2 Transitionen vom selben State kommen und vom selben Ereignis ausgelöst wurden, aber mit verschiedenen Guards. Falls dieses Ereignis auftritt und beide Guard Conditions true sind, denn wird nur eine Transition gefeuert, da sie in Konflikt miteinander sind Es können Transitionen nur dann gleichzeitig ausgelöst werden, wenn sie in gleichen orthogonalen Regionen auftreten.)
- There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out.)

## 6 Appendix

### Beispiele von problematischen Situationen:

---

#### **Deadlock**

---

- gegenseitige Blockade
    - $T_1$  sperrt A
    - $T_2$  sperrt B
    - ...
    - $T_1$  will B sperren, muß warten
    - $T_2$  will A sperren, muß aber auch warten
  - Deadlock-Vermeidung
    - jede Transaktion muß alle Sperren gleichzeitig vornehmen
      - + Lock(A,B),..., Unlock(A),..., Unlock(B)
      - + können nicht alle Locks gesetzt werden, muß die gesamte Transaktion warten
      - + erzeugt viele Sperrkonflikte
    - lineare Anordnung der Items
      - + jede Transaktion muß die Items in vorgegebener Reihenfolge sperren: A vor B vor C ...
- 

#### **Livelock**

---

- Probleme entstehen, wenn mehrere Transaktionen auf ein gesperrtes Element warten
    - $T_1$  sperrt A
    - $T_2$  will A sperren, muß aber warten
    - $T_3$  will A sperren, muß ebenfalls warten
    - $T_1$  gibt den Lock frei
    - $T_3$  bekommt die nächste Zeitscheibe
    - $T_2$  will A sperren, muß aber weiter warten
    - $T_4$  will A sperren, muß warten
    - $T_3$  gibt den Lock frei
    - $T_4$  bekommt die nächste Zeitscheibe
    - ...
  - Lösung: sinnvoller Schedule
  - meist „first-come-first-served“
-

## 7 Referenzen und Literatur

- 1) Unified Modeling Language Specification (UML Specification), Version 1.3 (zu finden auf <http://www.omg.net>)
- 2) The Semantics of UML State Machines by Johan Lilius, Iván Porrer Paltor, Turku Centre for Computer Science (Finnland)
- 3) Rewrite Rules and Operational Semantics for Model Checking UML Statecharts by Gihwon Kwon, Kyonggi University (Korea):