Hochschule Wismar
Fachhochschule für Technik, Wirtschaft und Gestaltung
Fakultät der Ingenieurwissenschaften

Master-Thesis

DEVS-basierte Steuerung flexibler Produktions- und Robotersysteme

Ba Eng Tobias Schwatinski, M.-Nr.: 103759

1. Prüfer	Prof. Dr.-Ing. Thorsten Pawletta, FG CEA
2. Prüfer	Prof. Dr.-Ing. Sven Pawletta, FG CEA Hochschule Wismar

Ausgabedatum	19.10.2009
Abgabedatum	25.01.2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen, Formelzeichen und Indizes

Abkürzungen

atomic DEVS	atomares DEVS
coupled DEVS	gekoppeltes DEVS
DEVN	DEVS Network
DEVS	Discrete Event System Formalism/Specification
FG CEA	Forschungsgruppe Computational Engineering and Automation
HiL	Hardware in the Loop
message	Nachricht im abstrakten DEVS-Simulator
PDEVN	parallel DEVS Network
PDEVS	parallel atomic DEVS
RCP	Rapid Control Prototyping
RTC	Real Time Clock
RT-DEVS	Real Time DEVS
SCEs	Scientific and Technical Computing and Visualization Environments
SiL	Software in the Loop
Struct	structure

Formelzeichen

A	Menge aller Aktivitäten eines RT-DEVS
a	Aktivität eines RT-DEVS
a_inc_pos	aktuelle Position einer Roboterachse
comps	componets - Struktur deren Felder mit atomic oder coupled DEVS belegt ist
D	Indexmenge
E	Vektor der Endlagentaster
e	time elapsed
ECC	engine control commands
el_number	Elementnummer innerhalb eines Vekors
f_inc_pos	Zielposition einer Roboterachse
I	Menge der Einwirkungen auf ein DEVS

IMM	Imminents
IPorts	Eingangsschnittstellen eines DEVS
layer	Variable zeigt die Verarbeitungsebene an
M	Modellbeschreibung eines DEVS
M1, M2, …, M4	Motorsteuerbefehle
mail	Liste für Eingangsnachrichten innerhalb eines gekoppelten DEVS
N	Bezeichnung eines DEVN
OPorts	Ausgangsschnittstellen eines DEVS
P	Vektor der Positionstaster
p	Schnittstelle/Portbezeichnung eines DEVS
phase	Merker für den aktuellen Aktivierungszustand eines atomic DEVS
Q	total state
q_1, q_2	Bufferinhalt
S	Menge der Gesamtzustände eines DEVS
s	Gesamtzustand eines DEVS
t	Simulationszeit
TA	Zeitdauer bis zum nächsten Aktivierungszeitpunkt eines coupled DEVS
ta	time advance-Funktion
ti	time intervall-Funktion
tl	Zeitpunkt des letzten Ereignisses eines atomic DEVS
tn	Zeitpunkt des nächsten internen Ereignisses eines atomic DEVS
traj	Matrix mit gespeicherten Bahnpunkten
upper_limit	maximaler Verfahrweg einer Roboterachse
v	Eingangswert eines Ports
X	Menge aller Eingangswerte eines DEVS
x	Eingangswerte eines DEVS
Y	Menge aller Ausgangswerte eines DEVS
y	Ausgangswerte eines DEVS
Z	output translation-Funktion
δ	Zustandsüberführungsfunktion eines DEVS
ε	mögliche Abweichung von der gewöhnlich zu erwartenden Ausführungszeit *comp_time* eines atomic RT-DEVS innerhalb von *ti*(s)
λ	Ausgangsfunktion

σ	Zustandsfeld - Zeitdauer bis zum nächsten internen Ereignis eines atomic DEVS
ψ	activity mapping-Funktion

Indizes

conf	confluent
d	Komponente eines coupled DEVS
ext	external
i	Komponente eines coupled DEVS
int	internal
N	gekoppeltes DEVS
p	Port / Schnittstelle

Mathematische Ausdrücke

inf	Unendlich ∞
N	Menge der natürlichen Zahlen
R^+	Menge der positiv reellen Zahlen
$R_{0,\infty}^+$	$R^+ \cup \{0\} \cup \{\infty\}$

Aufgabenstellung der Master-Thesis

Thema: DEVS-basierte Steuerung flexibler Produktions- und Robotersysteme

Der Discrete Event System Formalismus (DEVS) nach Zeigler ist ein allgemeiner systemtheoretischer Ansatz zur Modellierung und Simulation diskret-ereignisorientierter Systeme. In dieser Arbeit ist ein DEVS-Simulatorprototyp für die wissenschaftlich-technische Berechnungsumgebung Matlab/Simulink zu entwickeln. Anschließend sind für den Simulatorprototypen Echtzeitkonzepte und einfache Prozessschnittstellen zu entwickeln. Abschließend sind der Simulatorprototyp und die Echtzeitkonzepte an einer laborativen Montagestraße und einem integrierten Roboter zu erproben.

Schwerpunkte

- Einarbeitung in die DEVS-Theorie (DEVS-Modellierung, Simulationsalgorithmen)
- Entwicklung eines Matlab-DEVS-Simulators auf Grundlage des Parallel-DEVS Ansatzes
- Analyse des Simulationsmodellbasierten Steuerungsansatzes der FG CEA und des Rapid Control Prototyping Ansatzes (RCP) nach Abel
- Erweiterung des Matlab-DEVS-Simulators um Echtzeitfähigkeit und Prozessschnittstellen
- Erprobung des Gesamtkonzeptes an einer Beispielapplikation am Fischertechnik-Roboter

Die Arbeit ist nach den vorgegebenen Richtlinien anzufertigen. Darin eingeschlossen ist die Anfertigung eines Posters und einer WWW-Seite zu den wesentlichen Lösungswegen und Ergebnissen der Arbeit. Neben der geforderten Anzahl schriftlicher Exemplare ist die schriftliche Arbeit (Originaldateien & PDF-Dateien) sowie die entwickelte Software auf einem dokumentierten digitalen Datenträger abzugeben.

Betreuer und Prüfer: Prof. Dr.-Ing. Thorsten Pawletta, HS Wismar, FG CEA
2. Prüfer: Prof. Dr.-Ing. Sven Pawletta, HS Wismar, FG CEA

Ausgabedatum: 19.10.2009
Abgabetermim: 25.01.2010

1 Problemstellung

Der klassische Entwicklungsprozess für Automatisierungen folgt zumeist dem V-Modell. Dieser lässt sich nach [AB07] in nachfolgend genannte Schritte aufteilen:

- Aufgabenstellung formulieren, Lasten-/Pflichtenheft erstellen
- Analyse und Modellbildung des technischen Prozesses
- Simulation des technischen Prozessen zur Entwicklung und Erprobung einer geeigneten Regelung/Steuerung
- Codierung und Implementierung der entwickelten Algorithmen auf der Zielhardware
- Erprobung der Regler und der Teilsysteme
- Inbetriebnahme und Erprobung der vollständigen Regelung/Steuerung am Realprozess

Für die Mehrzahl der Entwicklungsschritte existieren spezielle Softwarepakete. Das Problem bei dieser Vorgehensweise stellen die Schnittstellen zwischen den einzelnen Arbeitsschritten oder den unterschiedlichen Softwarepaketen dar. So ist es durchaus denkbar, dass das Modell eines branchenüblichen Modellierungswerkzeugs nicht ohne weiteres in ein anderes Modellierungswerkzeug portierbar ist. Außerdem können Informationen zwischen den einzelnen Arbeitsschritten verloren gehen und liegen somit zu einem späteren Arbeitsschritt nicht mehr vor. Weiterhin muss der Entwickler beim klassischen Entwicklungsprozess eine Vielzahl unterschiedlicher Softwarepakete beherrschen. Das *Rapid Control Prototyping* (RCP) Konzept erlaubt den Entwicklungsprozess abzukürzen und zu vereinfachen. So ist es mit dem *Software in the Loop* (SiL) Ansatz möglich, ein Steuerungsmodell auf einem Entwicklungsrechner zu simulieren und damit einen realen Prozess zu steuern. Bei diesem Ansatz wird die Codierung und Implementierung der entwickelten Algorithmen auf der Zielhardware eingespart. Zurzeit wird das RCP-Konzept lediglich in wissenschaftlich-technischen Berechnungsumgebungen wie Matlab/Simulink genutzt. In dieser Arbeit wird untersucht, ob sich RCP und SiL auch auf Grundlage eines allgemeinen systemtheoretischen Ansatzes realisieren lassen. Der Discrete Event System Formalismus (DEVS) nach Zeigler stellt einen solchen Ansatz zur Modellierung und Simulation diskret-ereignisorientierter Systeme dar. Auf dieser Grundlage wird eine Robotersteuerung auf einem Entwicklungsrechner modelliert, simuliert und anschließend zur praktischen Steuerung eines Roboters eingesetzt.

2 Einführung in den Discrete Event System Formalismus (DEVS)

DEVS stellt eine formalisierte Modellierungs- und Simulationsmethodik dar, welche 1976 von Zeigler eingeführt worden ist. Die DEVS Methodik basiert auf einer modular hierarchischen Modellspezifikation. Das dynamische Verhalten der Systemelemente wird beim traditionellen DEVS Ansatz mit atomaren (atomic) DEVS Systemen beschrieben. Daneben gibt es gekoppelte (coupled) DEVS Systeme, welche die Komposition von atomic, bzw. coupled DEVS Systemen beschreiben. Weiterhin kann jedes coupled DEVS wiederum Bestandteil eines coupled DEVS sein. Außerdem gilt beim traditionellen DEVS Ansatz, dass coupled DEVS keine separate Dynamikbeschreibung besitzen. In diesem Abschnitt wird auf der Grundlage von [ZPK00] eine Einleitung in die Spezifikation und Abarbeitung (Simulation) von dynamischen Systemen auf Basis dieser Theorie gegeben.

2.1 Atomic DEVS Spezifikationen

Den elementarsten Baustein innerhalb dieser Theorie stellt das atomic DEVS dar. Grundsätzlich kann es als eine „Black Box“ mit Ein- und Ausgängen verstanden werden, deren Verhalten durch eine Fülle von eingeprägten Funktionen charakterisiert wird. Im Folgenden werden drei Spezifikationen von atomic DEVS vorgestellt.

Classic atomic DEVS System Specification

Ein classic atomic DEVS ist wie folgt definiert:

$DEVS = \{X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\}$

X	Menge der Eingangswerte
S	Menge der Zustandwerte
Y	Menge der Ausgangswerte
δ_{int}	δ_{int}: S→S, *internal transition function*

δ_{ext} δ_{ext}: Q x X→S, *external transition function*

Q={(s,e)|s ∈ S, 0≤e≤*ta(s)}*, *total state*

e verstrichene Zeit seit letzter Transition, *time elapsed*

λ λ: S→Y, *output function*

ta *ta*: S→$R_{0,\infty}^{+}$, *time advance function*

Jedes atomic DEVS besitzt einen inneren Zustand s. Mittels der time advance function *ta*(s) lässt sich aus diesem die Zeitspanne bis zum nächsten internen Ereignis bestimmen. Sobald diese Zeitspanne abgelaufen ist, wird die Ausgangsfunktion λ(s) ausgeführt und auf Grundlage des inneren Zustands s ein Ausgangsereignis auf dem Ausgang Y erzeugt. Danach wird die interne Zustandsüberführungsfunktion δ_{int}(s) ausgelöst, welche auf Grundlage des inneren Zustands s einen neuen inneren Zustand s' ermittelt. Sobald ein externes Ereignis x auf den Eingang X gelegt wird, muss die externe Zustandsüberführungsfunktion δ_{ext}(s,e,x) ausgelöst werden. Diese bestimmt auf Grundlage des inneren Zustands s, der verstrichenen Zeit e seit dem letzten internen Ereignis und dem externen Ereignis x den neuen inneren Zustand s'.

Das dynamische Verhalten eines classic atomic DEVS ist in Abbildung 1 dargestellt.

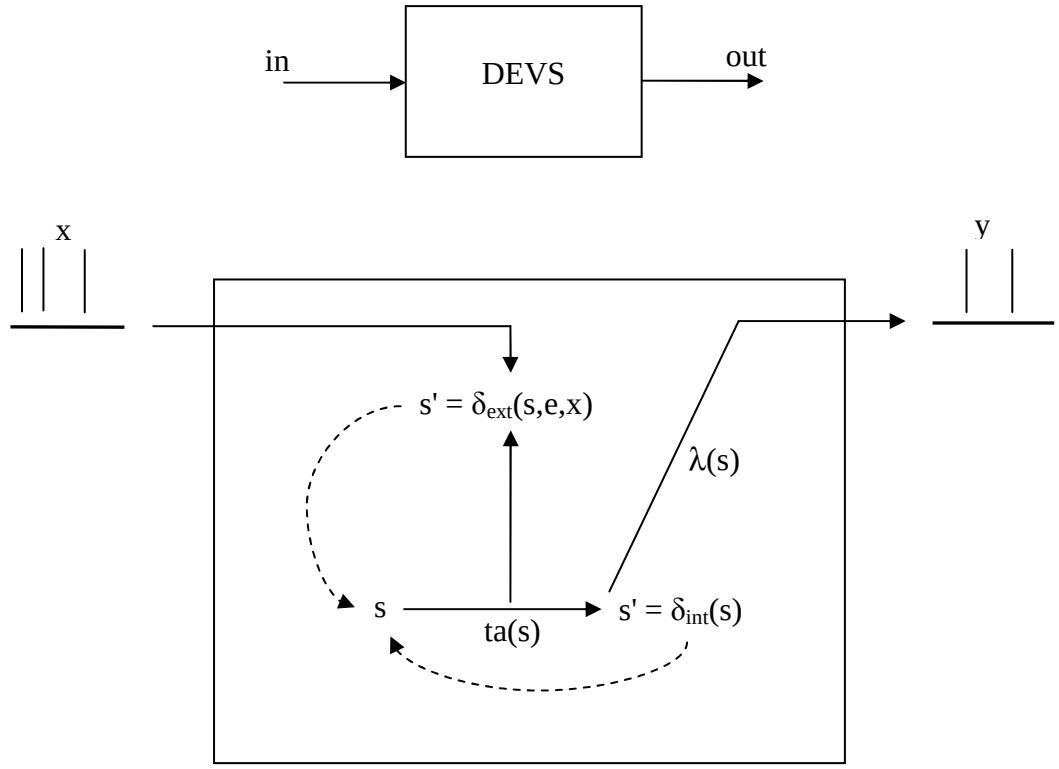


Abb. 1 : Dynamisches Verhalten eines atomic DEVS nach [ZPK00]

Classic atomic DEVS mit Ports

Classic atomic DEVS mit Ports definieren erweiterte Ein- und Ausgangsschnittstellen (Ports) auf Basis der *classic atomic DEVS*. Diese Erweiterung erlaubt die Vereinfachung des Modellierungsprozesses.

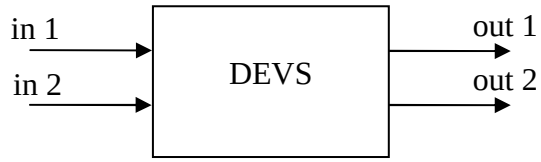


Abb. 2: Struktur eines classic atomic DEVS mit Ports

Die grundsätzliche Struktur entspricht ansonsten einem classic atomic DEVS.

$DEVS_{ports} = \{X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\}$

X $X=\{(p,v)|p \in IPorts, v \in X_p\}$, Menge aller Eingangswerte

Y $Y=\{(p,v)|p \in OPorts, v \in Y_p\}$, Menge aller Ausgangswerte

S, δ_{int}, δ_{ext}, *ta*, λ analog classic atomic DEVS

In diesem Zusammenhang stellt *p* die Bezeichnung des Ports und *v* den ihm zugeordneten Wert dar. Dabei lassen sich in jedem atomic DEVS beliebig viele Ports definieren. Allerdings gibt es bei den classic atomic DEVS mit Ports eine wichtige Einschränkung; es darf zu einem Simulationszeitpunkt jeweils nur an einem Eingangsport ein Ereignis anliegen. Diese Einschränkung wurde mit der Entwicklung von parallelen DEVS beseitigt.

Parallel atomic DEVS

Classic atomic DEVS können externe und interne Ereignisse verarbeiten. Dabei werden der Zeitpunkt eines jeden internen Ereignisses und somit auch der Zeitpunkt des Ausgangsereignisses vom atomic DEVS selbst bestimmt. Das Ausgangsereignis wird entsprechend den Koppelbeziehungen zu einem Eingangsereignis eines anderen atomic DEVS, welches daraufhin das externe Ereignis verarbeiten muss. Folglich könnte es während der Simulation zum gleichzeitigen Auftreten von externen und internen Ereignissen an einem atomic DEVS kommen. Der traditionelle DEVS Ansatz löst diesen Konflikt auf der gekoppelten Ebene. Der Parallel DEVS Ansatz löst diesen Konflikt an genau der Stelle, wo die Kollision von externen und internen Ereignis auftritt, nämlich beim atomic DEVS selbst. Zu diesem Zwecke ist das parallel atomic DEVS entwickelt worden.
Allgemein muss der Parallel DEVS Ansatz nach [ZPK00] drei wesentliche Eigenschaften erfüllen:

- Es werden sämtliche internen Ereignisse von den parallel atomic DEVS zur selben Simulationszeit ausgeführt und das jeweilige Ausgangsereignis erzeugt.
- Jedes erzeugte Ausgangsereignis wird entsprechend den Koppelbeziehungen zum Eingangsereignis weiterer parallel atomic DEVS.
- Die Zustandsüberführung wird im parallel atomic DEVS definiert.

Die grundsätzliche Struktur vom parallel atomic DEVS lautet:

$PDEVS = \{X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta\}$

δ_{con}	$\delta_{con}: Q \times X \rightarrow S$	*confluent transition function*
$X, Y, S, \delta_{int}, \delta_{ext}, ta, \lambda$		analog classic atomic DEVS

Dabei bestimmt die *confluent transition function* $\delta_{con}(s,e,x)$ aus dem aktuellen inneren Zustand s, der verstrichenen Zeit seit dem letzten internen Ereignis e und dem Eingangsereignis x den neuen inneren Zustand s', wenn externes und internes Ereignis zeitgleich auftreten.

Das dynamische Verhalten eines parallel atomic DEVS beim zeitgleichen Eintreten von externen und internen Ereignissen zeigt Abbildung 3.

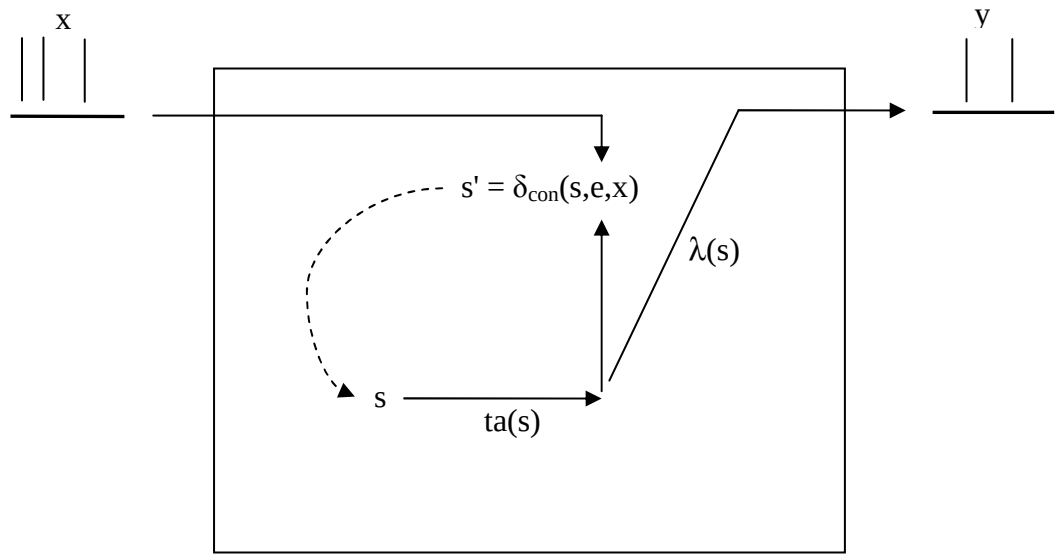


Abb. 3 : Dynamisches Verhalten eines parallel atomic DEVS bei gleichzeitigen externen und internen Ereignissen

2.2 Coupled DEVS Spezifikationen

In einem *coupled DEVS* können atomic DEVS komponiert werden. Auf die gleiche Weise lassen sich atomic DEVS mit coupled DEVS kombinieren und es entstehen hierarchische DEVS-Modelle. Aufgrund der *closure under coupling* Eigenschaft besitzt ein coupled DEVS nach außen die gleichen Eigenschaften wie ein atomic DEVS. Darüber hinaus muss ein coupled DEVS alle Informationen beinhalten, die zur Koordination der Komponenten notwendig sind. Im Folgenden werden zwei mögliche Spezifikationen nach [ZPK00] vorgestellt, welche als *classic coupled DEVS*, bzw. als *parallel coupled DEVS* bezeichnet werden.

Classic coupled DEVS

Die Struktur eines classic coupled DEVS, auch als DEVS Network (DEVN) bezeichnet, ist wie folgt definiert:

$DEVN=\{X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, Select\}$.

X	$X=\{(p,v)\|p \in IPorts, v \in X_p\}$, Menge aller Eingangswerte
Y	$Y=\{(p,v)\|p \in OPorts, v \in Y_p\}$, Menge aller Ausgangswerte
D	Indexmenge
M_d	Modellbeschreibung der Komponenten für $d \in D$ mit $M_d = \{X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\}$ als ein classic atomic DEVS mit Ports
I_d	ist die Menge aller Einwirkungen[1] auf d: $I_d \subseteq D \cup \{N\}$, $d \notin I_d$
$Z_{i,d}$	ist die *i-to-d-output translation* mit $i \in I_d$, für die gilt: $Z_{i,d}: X \rightarrow X_d$, wenn i=N $Z_{i,d}: Y_i \rightarrow Y$, wenn d=N $Z_{i,d}: Y_i \rightarrow X_d$, wenn $d \neq N$ und $i \neq N$
Select	*tie-breaking-fuction* (wählt beim Vorhandensein mehrerer *Imminents* genau das $d \in D$ mit der höchsten Priorität)

Die *Select* Funktion regelt die Auflösung von Konflikten in einem classic coupled DEVS aufgrund zeitgleicher interner Ereignisse.

[1] N ist eine Abkürzung für DEVN und stellt das coupled DEVS selbst dar

Komponenten mit simultanen internen Ereignissen werden als *Imminents* bezeichnet. Abbildung 4 zeigt die Problemstellung am Beispiel eines Pipeline-Modells. Die Pipeline besteht aus drei atomaren Server-Modellen ohne Queues. Sind zum Beispiel Server 0 und Server 1 Imminents, entscheidet die *Select* Funktion welche der beiden Komponenten zuerst das interne Ereignis durchführt und somit auch das Ausgangsereignis erzeugt. Es findet keine parallele Verarbeitung der Komponenten statt.

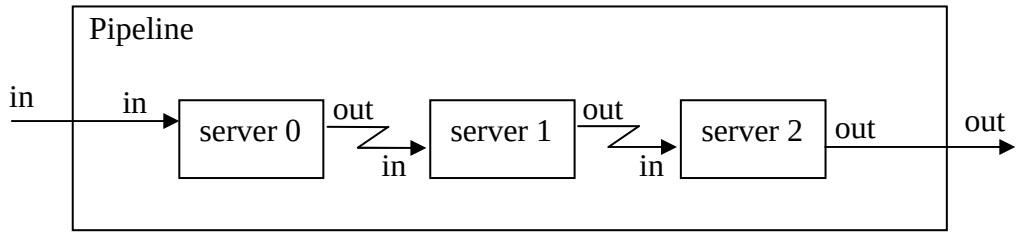


Abb. 4: Struktur eines Pipeline Modells

Parallel coupled DEVS

Die Struktur eines parallel coupled DEVS, im Folgenden auch parallel DEVS Network (PDEVN) bezeichnet, ist wie folgt definiert:

PDEVN={X, Y, D, {M_d}, {I_d}, {$Z_{i,d}$}}

M_d Modellbeschreibung der Komponenten für d ∈ D mit

M_d = {X, Y, S, δ_{int}, δ_{ext}, δ_{con}, λ, *ta}* als ein parallel atomic DEVS

X, Y, D, I_d, $Z_{i,d}$ analog DEVN

Da die Auflösung von Konflikten in einem PDEVN aufgrund zeitgleicher interner Ereignisse auf atomarer Ebene (siehe parallel atomic DEVS) erfolgt, muss keine *Select* Funktion definiert werden.

Im Hinblick auf die Simulation müssen PDEVN und DEVN die arrangierten Komponenten in 6 Gruppen einteilen. Diese Einteilung bildet eine wichtige Grundlage für die Simulationsalgorithmen. Die Ablaufsteuerung eines gekoppelten Modells, Koordinator genannt, muss auf Basis seiner Komponenten d ∈ D seinen nächsten Aktivierungszeitpunkt t_{next} = t + TA bestimmen. Die Zeitdauer bis zum nächsten Aktivierungszeitpunkt ist durch *TA* bestimmt.

$TA = \text{minimum}\{\sigma_d \mid d \in D\}$, mit $\sigma_d = ta(s_d) - e_d$ und $s_d \in S_d$

Es kann nachfolgende Unterscheidung getroffen werden:

1) $IMM = \{d \mid \sigma_d = TA\}$ imminent components

2) $INF = \{d \mid i \in I_d \wedge i \in IMM \wedge x_d^b \neq \varnothing\}$, components about to receive inputs

mit $x_d^b = \{ Z_{i,d}(\lambda_i(s_i)) \mid i \in IMM \wedge i \in I_d \}$

3) $CONF = IMM \cap INF$ confluent components

4) $INT = IMM - INF$ imminent components receiving no input

5) $EXT = INF - IMM$ components receiving input but not imminent

6) $UN = D - IMM - INF$ remaining components

Alle Komponenten, an denen ein internes Ereignis vorliegt, werden der Menge IMM zugeordnet. Jede Komponente, welche eine Einwirkung einer imminent-Komponente besitzt, wird der Menge INF zugeordnet. Aus diesen beiden Mengen lassen sich Untermengen ableiten. Die Menge CONF ist die Schnittmenge von IMM und INF und beinhaltet die Komponenten bei denen ein externes und internes Ereignis zeitgleich auftritt. Die Mengen INT und EXT beinhalten die Komponenten an denen entweder nur ein internes Ereignis oder ein externes Ereignis vorliegt. Alle Komponenten eines gekoppelten Modells, die nicht den Mengen CONF, INT oder EXT zugeordnet werden können, werden der Menge UN zugeordnet. Aufgrund dieser Einteilung ist es möglich, den Komponenten innerhalb von CONF, INT und EXT die korrekte Zustandsüberführungsfunktionen (δ_{conf}, δ_{int} oder δ_{ext}) zuzuordnen.

2.3 *Grundlagen DEVS-basierter Simulatoren*

Ein hierarchisches DEVS Modell kann in Form eines *composition-tree* als Graph dargestellt werden. Zur Abarbeitung wird jedem atomaren Modell ein Simulator und jedem gekoppelten Modell ein Koordinator zugeordnet. Außerdem ist allen Koordinatoren und Simulatoren ein *root-coordinator* übergeordnet, welcher solange die Simulationszeit fortschaltet, bis ein zuvor definierter Abbruchzeitpunkt eingetreten ist. Die Überführung eines hierarchischen DEVS Modells in eine Simulationsumgebung soll anhand eines Beispiels verdeutlicht werden und ist in den Abbildungen 5 und 6 dargestellt. Die Komponenten M1, M21 und M22 stellen atomic DEVS und die Komponenten N1 und N2 stellen coupled DEVS dar.

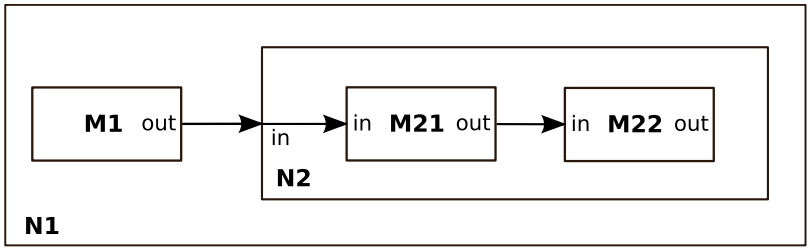


Abb. 5: Beispiel für ein hierarchisches DEVS Modell

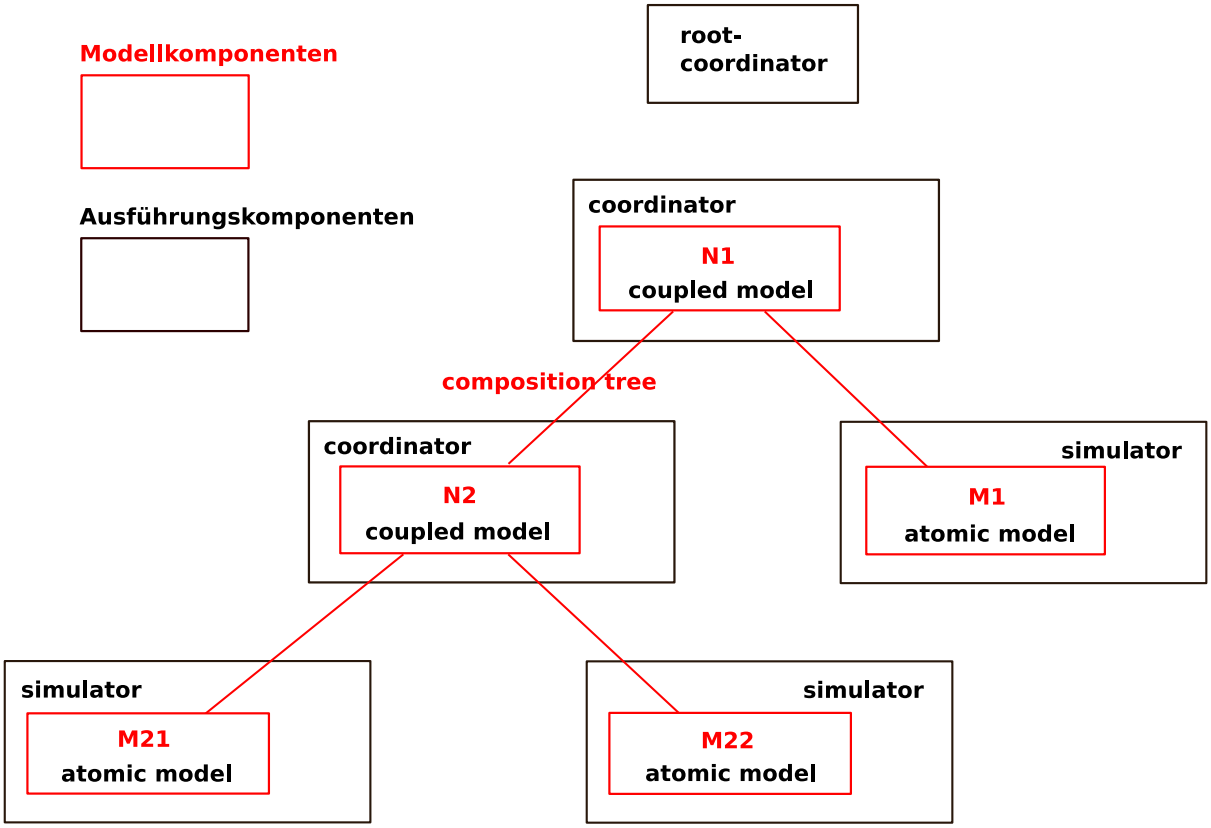


Abb. 6: Überführung eines hierarchischen DEVS Modells in einen composition-tree mit zugeordneten Ausführungskomponenten

Die Simulation erfolgt ereignisorientiert. Während der Simulation kommunizieren die Koordinatoren und Simulatoren mittels verschiedener Nachrichten. Dieses Nachrichtenkonzept zeigt Abbildung 7.

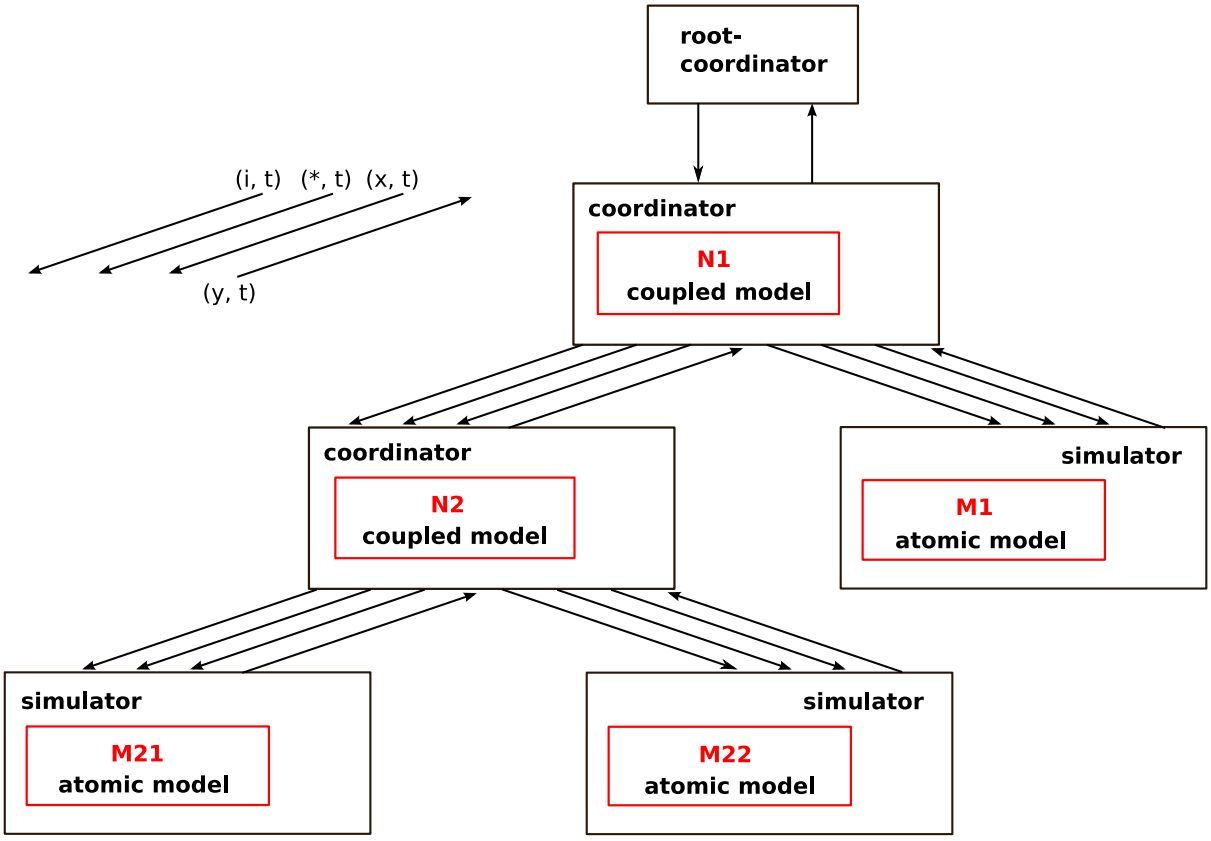


Abb. 7: Nachrichtenkonzept während Simulation

Die nachfolgend genannten Nachrichten können zum Simulationszeitpunkt t, vom Elternkoordinator an alle untergeordneten Simulatoren oder Koordinatoren gesendet werden:

1. Die *Initialisierungsnachricht (i, t)* wird einmalig zu Beginn der Simulation versendet; erst danach beginnt die eigentliche Simulationsphase.
2. Die generelle Ablaufsteuerung wird durch die *Zustandnachricht (*, t)* realisiert. Dazu wählen die Koordinatoren alle Komponenten des zugeordneten coupled DEVS aus ihrer Ereignisliste aus, deren Zeitpunkt *tn* für das nächste interne Ereignis gleich der aktuellen Simulationszeit *t* ist und versenden an diese die Zustandsnachricht.
3. Die *Eingangsnachricht (x, t)* wird zur Übertragung von Ereignissen genutzt und führt beim Empfänger zu einem externen Ereignis.

Außerdem gibt es die *Ausgangsnachricht (y, t)*, welche ebenfalls zur Übertragung von Ereignissen genutzt wird und von Simulatoren und Koordinatoren an ihnen übergeordnete Koordinatoren versendet wird.

Für die Ausführungskomponenten gilt folgender Algorithmus[1]:

- root-coordinator
  - Eine Initialisierungsnachricht wird zum Simulationsstart an den untergeordneten Koordinator gesendet.
  - Es wird geprüft, nachdem alle Nachrichten vollständig abgearbeitet worden sind, ob ein spezielles Abbruchkriterium vorliegt oder das zeitliche Simulationsende erreicht ist. Wenn dies nicht der Fall ist, wird der Zeitpunkt *tn* des nächsten internen Ereignisses des untergeordneten Koordinators ermittelt, die Simulationszeit darauf vorgeschaltet und eine Zustandsnachricht an den untergeordneten Koordinator versendet.
- Simulator
  - Eine Initialisierungsnachricht führt zur Berechnung des letzten Ereigniszeitpunktes *tl* und des nächsten Ereigniszeitpunktes *tn*. Die Berechnung von *tn* erfolgt auf Grundlage der *ta*-Funktion des zugeordneten atomic DEVS.
  - Eine Zustandsnachricht zeigt ein internes Ereignis an. Daraufhin wird mit der Ausgangsfunktion λ des zugeordneten atomic DEVS das Ausgangsereignis berechnet und die interne Zustandsüberführungsfunktion δ_{int} des atomic DEVS ausgeführt. Das Ausgangsereignis wird mittels einer Ausgangsnachricht an den übergeordneten Koordinator gesendet. Danach wird der letzte Ereigniszeitpunkt *tl* auf die aktuelle Simulationszeit gesetzt und der nächste Ereigniszeitpunkt *tn* berechnet.
  - Eine Eingangsnachricht zeigt ein externes Ereignis an. Daraufhin wird die externe Zustandsüberführungsfunktion δ_{ext} des zugeordneten atomic DEVS ausgeführt. Danach wird der letzte Ereigniszeitpunkt *tl* auf die aktuelle Simulationszeit gesetzt und der nächste Ereigniszeitpunkt *tn* berechnet.

[1] Klassischer Simulationsalgorithmus nach [ZPK00]

- Koordinator
    - Eine Initialisierungsnachricht wird an alle Simulatoren und Koordinatoren der Komponenten eines coupled DEVS weitergeleitet. Anschließend wird der letzte Ereigniszeitpunkt *tl* als das Maximum der Ereigniszeitpunkte *tl* aller Kinder ermittelt. Daneben wird der nächste Ereigniszeitpunkt *tn* als das Minimum der Ereigniszeitpunkte *tn* aller Kinder ermittelt.
    - Eine Zustandsnachricht wird an den Simulator oder Koordinator der Komponente eines coupled DEVS mit dem kleinsten nächsten internen Ereigniszeitpunkt *tn* weitergeleitet. Existieren mehrere solcher Komponenten wird beim klassischen Ansatz, durch die *Select-function,* die Abarbeitungsfolge festgelegt. Nachdem das interne Ereignis und alle damit zusammenhängenden externen Ereignisse verarbeitet worden sind, werden die Ereigniszeitpunkte *tl* und *tn* neu berechnet.
    - Eine Eingangsnachricht wird entsprechend den Koppelbeziehungen eines coupled DEVS vom Koordinator an zugeordnete Simulatoren oder Koordinatoren weitergeleitet. Anschließend werden die Ereigniszeitpunkte *tl* und *tn* neu berechnet.
    - Eine Ausgangsnachricht wird in eine Eingangsnachricht umgewandelt und entsprechend den Koppelbeziehungen eines coupled DEVS an zugeordnete Simulatoren oder Koordinatoren versendet. Falls eine Ausgangsnachricht an den übergeordneten Koordinator adressiert ist, wird diese als Ausgangsnachricht an den übergeordneten Koordinator versendet. Anschließend werden die Ereigniszeitpunkte *tl* und *tn* neu berechnet.

Im nächsten Abschnitt werden einige Aspekte von DEVS-basierten Ablaufsteuerungen eingehender diskutiert. Dabei werden auch Probleme bekannter Simulationsalgorithmen analysiert.

3 Entwicklung eines Simulators auf Basis von Parallel DEVS

Aufbauend auf die im Kapitel 2 eingeführten Spezifikationen werden in diesem Kapitel die bekannten Simulationsalgorithmen für Parallel DEVS Modelle nach Chow [Cho96] und nach Zeigler [ZPK00] betrachtet. Der Simulationsalgorithmus von Chow ist für PDEVS Modelle ohne Ports ausgelegt, gemäß der Systemspezifikation im zweiten Kapitel. Der Simulationsalgorithmus nach Zeigler unterstützt PDEVS Modelle mit Ports, welche eine Kombination von PDEVS und DEVS mit Ports gemäß Kapitel 2 darstellen. Nach Einführung der beiden grundlegenden Ablaufsteuerungen wird der Simulationsalgorithmus nach Zeigler eingehend analysiert und ein modifizierter Algorithmus für PDEVS mit Ports aufgezeigt.

3.1 Simulationsalgorithmus nach Chow

Die Ablaufsteuerung orientiert sich an dem im Abschnitt 2.3 dargestellten Prinzip. Die Simulatoren und Koordinatoren kommunizieren mittels 4 unterschiedlichen Nachrichten miteinander.

1. Initialisierungsnachricht (i, t)
2. Zustandsnachricht (*, *x_count*, t)
3. Ein-/Ausgangsnachricht (#, $Z_{i,d}(y_i)$, t), bzw. (#, $\lambda(s)$, t)
4. Abschlussnachricht (*Done*, *tn*)

Für die Ausführungskomponenten gilt folgender Algorithmus:

- root-coordinator
    - Eine Initialisierungsnachricht wird zum Simulationsstart an den untergeordneten Koordinator gesendet.
    - Es wird geprüft, nachdem vom untergeordneten Koordinator eine Abschlussnachricht (*Done*, *tn*) empfangen worden ist, ob entweder ein spezielles Abbruchkriterium vorliegt oder das zeitliche Simulationsende erreicht ist. Wenn dies nicht der Fall ist, wird die Simulationszeit t = *tn* gesetzt und die Zustandsnachricht (*, 0, t) an den untergeordneten Koordinator gesendet.

- Koordinator
  - Eine Initialisierungsnachricht wird an alle Simulatoren und Koordinatoren der Komponenten eines coupled DEVS weitergeleitet. Anschließend wird der nächste Ereigniszeitpunkt *tn* als das Minimum der Ereigniszeitpunkte *tn* aller Kinder ermittelt und eine Abschlussnachricht (*Done*, *tn*) an den übergeordneten Koordinator gesendet.
  - Wenn eine Zustandsnachricht (*, *x_count*, t) empfangen wird, müssen die Komponenten des coupled DEVS gemäß den Definitionen in Abschnitt 2.2 den Mengen IMM, INF, CONF und UN zugeordnet werden. Für jede Komponente innerhalb der Mengen IMM und INF wird die Gesamtanzahl *i_count* der aktuellen Einwirkungen zum Simulationszeitpunkt t ermittelt. Falls eine Komponente durch das gekoppelte System selbst beeinflusst wird, müssen diese Einwirkungen auch durch *i_count* = *i_count* + *x_count* berücksichtigt werden. Danach wird an die Komponenten der Mengen IMM und INF eine Zustandsnachricht (*, *i_count*, t) gesendet. Abbildung 8 zeigt den Vorgang zur Bestimmung von *i_count* und das anschließende Versenden von Zustandsnachrichten durch einen Koordinator an einem Beispiel mit 2 gekoppelten Modellen.

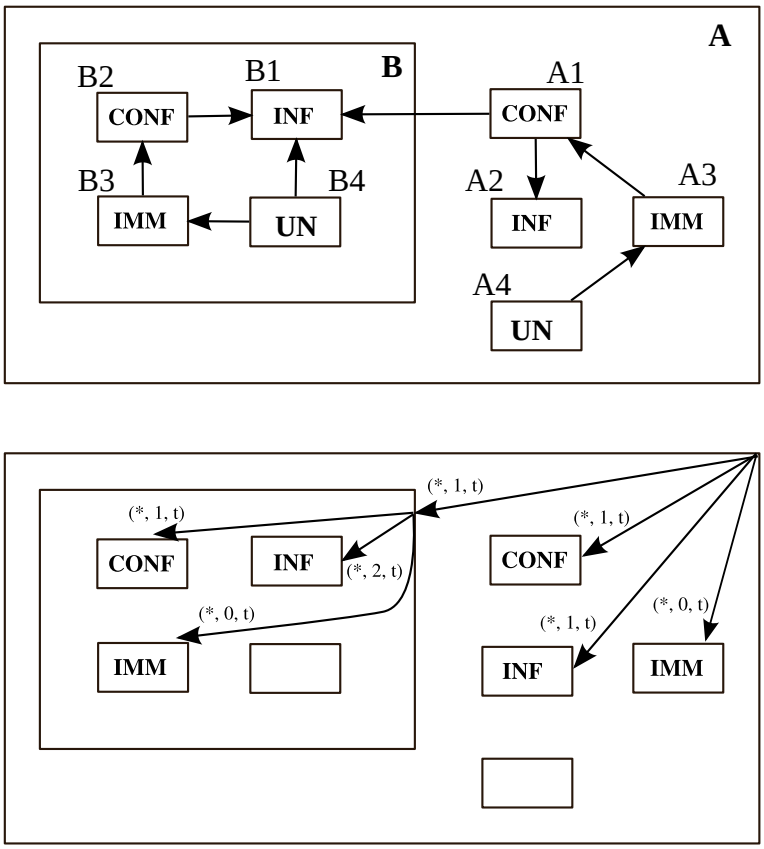


Abb. 8: Ermittlung der Einwirkungen und Versenden von Zustandsnachrichten

Die atomare Komponente A1 von A gehört zur Menge CONF und hat den aktuellen Beeinflusser A3. Demzufolge setzt der Koordinator von A für Komponente A1 *i_count* = 1. Die atomare Komponente B1 von B gehört zur Menge INF und hat den aktuellen Beeinflusser B2 sowie das gekoppelte System selbst. Demzufolge setzt der Koordinator von B für Komponente B1 *i_count=2*.

Eine koordinatorinterne Variable *semaphore_count* wird bei jedem Senden einer Zustandsnachricht um eine Einheit inkrementiert. Das Senden der Abschlussnachricht (*Done*, *tn*) an den übergeordneten Koordinator wird solange hinausgezögert, bis im Rahmen des weiteren Programmablaufs *semaphore_count* = 0 ist.

- Eine Ausgangsnachricht (#, y, t) wird in eine Eingangsnachricht (#, $Z_{i,d}(y_i)$, t) umgewandelt und entsprechend den Koppelbeziehungen eines coupled DEVS an zugeordnete Simulatoren oder Koordinatoren versendet. Falls eine Ausgangsnachricht an den übergeordneten Koordinator adressiert ist, wird diese als Ausgangsnachricht (#, $Z_{i,N}(y_i)$, t) an den übergeordneten Koordinator versendet.
- Eine Eingangsnachricht (#, y, t) wird in eine Eingangsnachricht (#, $Z_{i,d}(y_i)$, t) umgewandelt und entsprechend den Koppelbeziehungen eines coupled DEVS an zugeordnete Simulatoren und Koordinatoren versendet.
- Wenn eine Abschlussnachricht (*Done*, *tn*) empfangen wird, muss der Zeitpunkt für das nächste interne Ereignis *tn* des Absenders in einer Ereignisliste abgespeichert werden. Außerdem wird die interne Variable *semaphore_count* um eine Einheit dekrementiert.

- Simulator
  - Eine Initialisierungsnachricht führt zur Berechnung des nächsten Ereigniszeitpunktes *tn* und zum Senden einer Abschlussnachricht (*Done*, *tn*) an den übergeordneten Koordinator.
  - Eine Zustandsnachricht (*, *x_count*, t) zeigt das interne und/oder externe Ereignis an. Zunächst wird die simulatorinterne Variable *semaphore_count* = *x_count* gesetzt.
    - Für den Fall, dass die Simulationszeit t gleich dem Zeitpunkt des nächsten internen Ereignisses *tn* ist, wird das Ausgangsereignis des zugeordneten atomic DEVS berechnet und mittels einer

Ausgangsnachricht (#, λ(s), t) an den übergeordneten Koordinator gesendet.

- Falls *semaphore_count* = 0 ist, liegt nur ein internes Ereignis vor und es wird die interne Zustandsüberführungsfunktion des zugeordneten atomic DEVS ausgeführt.
- Falls *semaphore_count* ≠ 0 ist, liegen externe und interne Ereignisse zeitgleich vor. Die Ausführung der *confluent function* wird daraufhin solange hinausgezögert, bis im Rahmen des weiteren Programmaublaufs *semaphore_count = 0* ist.

▪ Wenn die Simulationszeit t ungleich dem Zeitpunkt des nächsten internen Ereignisses *tn* ist, liegen externe Ereignisse vor. Die Ausführung der externen Zustandsüberführungsfunktion wird daraufhin solange hinausgezögert, bis im Rahmen des weiteren Programmaublaufs *semaphore_count* = 0 ist.

▪ Wenn *samaphore_count* = 0 ist, muss der nächste Ereigniszeitpunkt *tn* berechnet und eine Abschlussnachricht (*Done*, *tn*) an den übergeordneten Koordinator gesendet werden.

o Als Reaktion auf eine Eingangsnachricht (#, y, t) wird zum einen das Eingangsereignis y in einem Vektor x^b (event bag) abgespeichert und zum anderen die interne Variable *semaphore_count* um eine Einheit dekrementiert. Die Variable *semaphore_count* erreicht den Wert null, nachdem ***alle Eingangsereignisse zum Simulationszeitpunkt t*** abgespeichert worden sind.

Ein äußerst wichtiges Element innerhalb des Simulationsalgorithmus stellt die Variable *x_count* der Zustandsnachricht dar. Sie führt dazu, dass der Simulator eines atomic DEVS genau *x_count* viele Eingangsereignisse in einem Vektor abspeichert, bevor eine Zustandsüberführungsfunktion durchgeführt wird. An dieser Stelle wird auch deutlich, warum Chows' Algorithmus keine Ports beinhaltet, denn die Gesamtzahl der Einwirkungen lässt sich nicht mehr allein aus den Koppelbeziehungen und Mengen herleiten. Diesen Sachverhalt soll Abbildung 9 verdeutlichen. In (a) ermittelt der Koordinator des coupled DEVS N1 die Gesamtzahl der Einwirkungen auf die atomic DEVS A1, A2 und A3 auf Grundlage der Koppelbeziehungen und den Mengen IMM und INF. Danach werden die Simulationszeit und die Gesamtzahl der Einwirkungen mittels der Zustandsnachricht (*, *x_count*, t) an die Simulatoren der betreffenden atomic DEVS gesendet. In (b) sind die Einwirkungen auf ein

atomic DEVS auf der lokalen Simulatorvariablen *semaphore_count* abgespeichert. Da für A1 keine Einwirkungen vorliegen, wird zuerst das Ausgangsereignis dieser Komponente berechnet und danach die interne Zustandsüberführungsfunktion ausgeführt. Im vorliegenden Fall soll das Ausgangsereignis nur am Ausgangsport *out_2* anliegen. Somit wird das Ausgangsereignis von A1 zum Eingangsereignis von A3. Da das atomic DEVS A3 genau ein Eingangsereignis erwartet, kann der zugeordnete Simulator die externe Zustandsüberführungsfunktion ausführen. Der Simulator des atomic DEVS A2 kann hingegen niemals die externe Zustangsüberführungsfunktion ausführen, da am Ausgangsport *out_1* kein Ausgangsereignis vorliegt. Der Simulator von A2 erwartet demnach ein Eingangsereignis, welches nicht von A1 generiert wird. Die Gesamtzahl der tatsächlich vorliegenden Einwirkungen lässt sich folglich nur dann bestimmen, wenn zuvor ein jeder Simulator das Ausgangsereignis generiert hat und somit die exakte Portbelegung mit Ereignissen bekannt ist. Dieses Vorgehen ist bei Chow nicht vorgesehen.

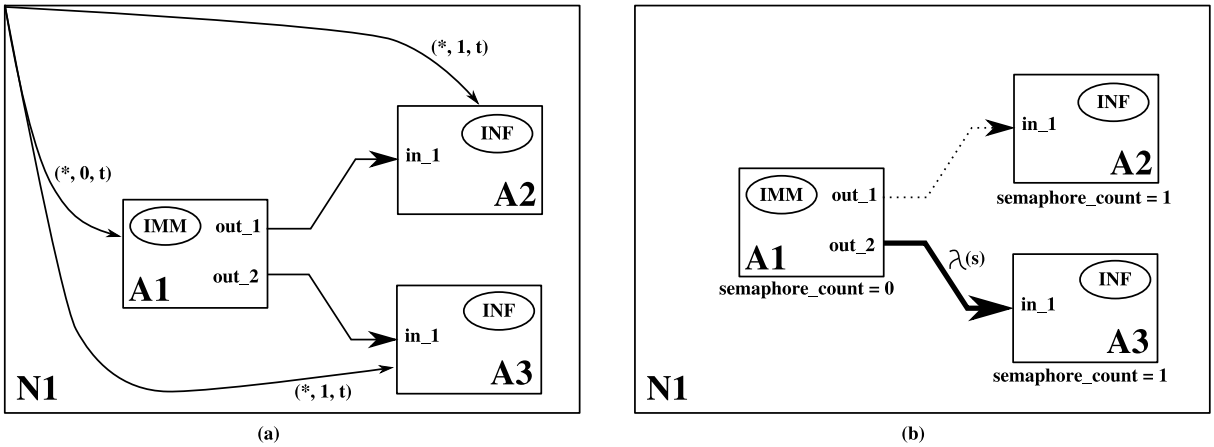


Abb. 9: Ermittlung der Einwirkungen und Versenden von Zustandsnachrichten

3.2 Simulationsalgorithmus nach Zeigler

Die Ablaufsteuerung orientiert sich auch an dem im Abschnitt 2.3 dargestellten Prinzip. Die Simulatoren und Koordinatoren kommunizieren mittels 4 unterschiedlichen Nachrichten.

1. Initialisierungsnachricht (i, t)
2. Zustandsnachricht (*, t)
3. Eingangsnachricht (x, t)
4. Ausgangsnachricht (y, t)

Für die Ausführungskomponenten gilt folgender Algorithmus:

- root-coordinator
  - Eine Initialisierungsnachricht wird zum Simulationsstart an den untergeordneten Koordinator gesendet.
  - Es wird geprüft, nachdem alle Nachrichten vollständig abgearbeitet worden sind, ob entweder ein spezielles Abbruchkriterium vorliegt oder das zeitliche Simulationsende erreicht ist. Wenn dies nicht der Fall ist, wird der Zeitpunkt *tn* des nächsten internen Ereignisses des untergeordneten Koordinators ermittelt, die Simulationszeit darauf vorgeschaltet und eine Zustandsnachricht an den untergeordneten Koordinator versendet.
- Simulator
  - Eine Initialisierungsnachricht führt zur Berechnung des letzten Ereigniszeitpunktes *tl* und des nächsten Ereigniszeitpunktes *tn*.
  - Eine Zustandsnachricht (*, t) zeigt das interne Ereignis an. Daraufhin wird das Ausgangsereignis berechnet und dieses mittels einer Ausgangsnachricht (λ(s), t) an den übergeordneten Koordinator gesendet.

- Eine Eingangsnachricht (x, t) zeigt das interne und/oder externe Ereignis an und
  - es wird die entsprechende Zustandsüberführungsfunktion ausgeführt.
    - Interne Zustandsüberführungsfunktion, wenn die Simulationszeit t = *tn* ist und mit dem Eingangsereignis x eine leere Menge übermittelt wird
    - *confluent function*, wenn die Simulationszeit t = *tn* ist und mit dem Eingangsereignis x eine nicht leere Menge von Einzelereignissen übermittel wird (bag of input events)
    - Externe Zustandsüberführungsfunktion, wenn für die Simulationszeit $tl \leq t \leq tn$ gilt und mit dem Eingangsereignis x eine nicht leere Menge von Einzelereignissen übermittelt wird
  - Danach wird der letzte Ereigniszeitpunkt *tl* auf die aktuelle Simulationszeit gesetzt und der nächste Ereigniszeitpunkt *tn* berechnet.

- Koordinator
  - Eine Initialisierungsnachricht wird an alle Simulatoren und Koordinatoren der Komponenten eines coupled DEVS weitergeleitet. Anschließend wird der letzte Ereigniszeitpunkt *tl* als das Maximum der Ereigniszeitpunkte *tl* aller Kinder ermittelt. Daneben wird der nächste Ereigniszeitpunkt *tn* als das Minimum der Ereigniszeitpunkte *tn* aller Kinder ermittelt.
  - Eine Zustandsnachricht (*, t) wird an alle Komponenten der Menge IMM gemäß den Definitionen in Abschnitt 2.2 weitergeleitet.
  - Eine Eingangsnachricht (x, t) wird in eine Eingangsnachricht ($Z_{N,r}(x)$, t) umgewandelt und entsprechend den Koppelbeziehungen eines coupled DEVS an zugeordnete Simulatoren und Koordinatoren gesendet. Zusätzlich wird an alle Komponenten innerhalb der Menge IMM, bei denen zu diesem Zeitpunkt kein Eingangsereignis vorliegt, die Eingangsnachricht ($\varnothing$, t) gesendet. Anschließend werden die Ereigniszeitpunkte *tl* und *tn* neu berechnet.

- Alle eingehenden Ausgangsnachrichten (y, t) werden zusammen mit einer Absenderkennung in einer internen Liste *mail* abgespeichert. Erst nachdem alle Komponenten innerhalb der Menge IMM des Koordinators ihre Ausgangsnachricht versendet haben, erfolgt eine Verteilung der Ereignisse aus *mail* an ihre Empfänger gemäß den Koppelbeziehungen des coupled DEVS.
  - Ausgangsnachricht (y_N, t) an den übergeordneten Koordinator, für alle $r \in I_N$, für die ein Ausgangsereignis von Komponente r in *mail* abgespeichert ist
  - Ausgangsnachricht ($\varnothing$, t) an den übergeordneten Koordinator, falls für kein $r \in I_N$ ein Ausgangsereignis von r in *mail* abgespeichert ist
  - Eingangsnachricht (x_d, t) an eine Komponente d, für alle $r \in I_d$, für die ein Ausgangsereignis von Komponente r in *mail* abgespeichert ist

  Zusätzlich wird an alle Komponenten innerhalb der Menge IMM, bei denen zu diesem Zeitpunkt kein Eingangsereignis vorliegt, die Eingangsnachricht ($\varnothing$, t) gesendet. Danach werden die Ereigniszeitpunkte *tl* und *tn* neu berechnet.

3.3 Kritik am Simulationsalgorithmus nach Zeigler

Der grundlegende Simulationsalgorithmus nach Zeigler [ZPK00] soll anhand von Message Sequenz Diagrammen detailliert untersucht werden. Zur vereinfachten Darstellung wird eine echte synchrone Arbeitsweise der Koordinatoren und Simulatoren zu Grunde gelegt. Das bedeutet, dass alle Komponenten gleichzeitig Ausgangsereignisse erzeugen oder Zustandsüberführungsfunktionen ausführen und auch gleichzeitig Nachrichten versenden können. Das Augenmerk soll auf den versendeten Nachrichten liegen.
Das erste Beispiel besteht aus drei atomic DEVS (M1, M21 und M22), welche im coupled DEVS N1 komponiert sind.

Den zugehörigen Modellaufbau, bzw. den composition-tree mit den Modellkomponenten und den zugeordneten Ausführungskomponenten zeigen die Abbildungen 10 und 11.

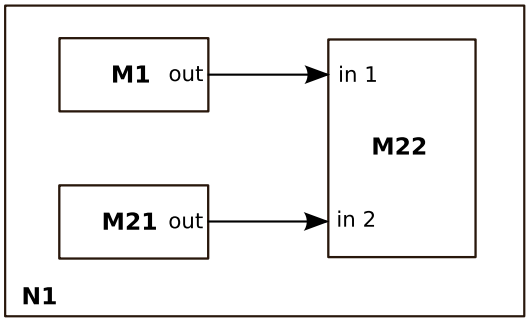


Abb. 10: Diskussionsbeispiel 1, Modellaufbau

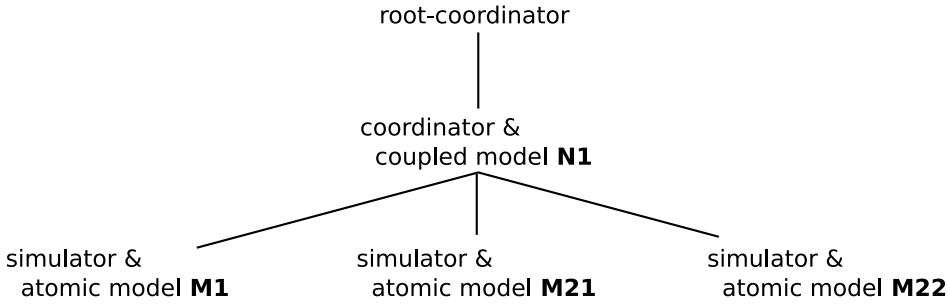


Abb. 11: Diskussionsbeispiel 1, composition-tree mit Modell- und zugeordneten Ausführungskomponenten

Nach der Initialisierung sollen die Komponenten M1 und M21 ihr internes Ereignis auslösen. Das zugehörige Message Sequenz Diagramm zeigt Abbildung 12.

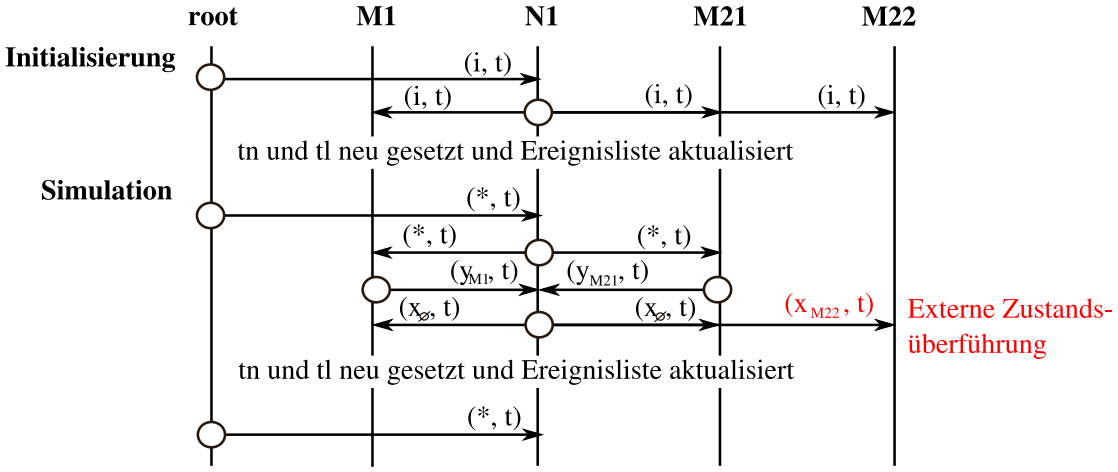


Abb. 12: Diskussionsbeispiel 1, Message Sequenz Diagram

Es zeigt sich, dass die Ausgangsereignisse y_{M1} und y_{M21} vom Koordinator N1 in x_{M22} zusammengefasst werden und mittels einer einzigen Eingangsnachricht an M22 versendet werden. An der Komponente M22 liegt somit ein externes Ereignis vor. Die Komponenten

M1 und M21 erhalten vom Koordinator N1 ebenfalls eine Eingangsnachricht ($x_{\varnothing}$ = leere Menge) und führen daraufhin ihre interne Zustandsüberführungsfunktion durch. Die „grundlegende Abarbeitungsfolge“ ist bei diesem Beispiel bei den Algorithmen nach Chow und Zeigler identisch.

Wenn man allerdings die atomic DEVS M21 und M22 in einem coupled DEVS N2 komponiert, ergibt sich bei Zeigler eine andere Abarbeitungsfolge. Den neuen Modellaufbau, bzw. den composition tree mit den Modellkomponenten und den zugeordneten Ausführungskomponenten zeigen die Abbildungen 13 und 14.

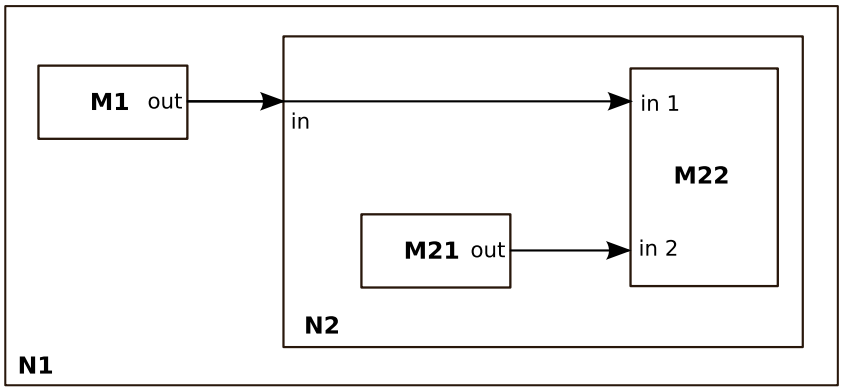


Abb. 13: Diskussionsbeispiel 2, Modellaufbau

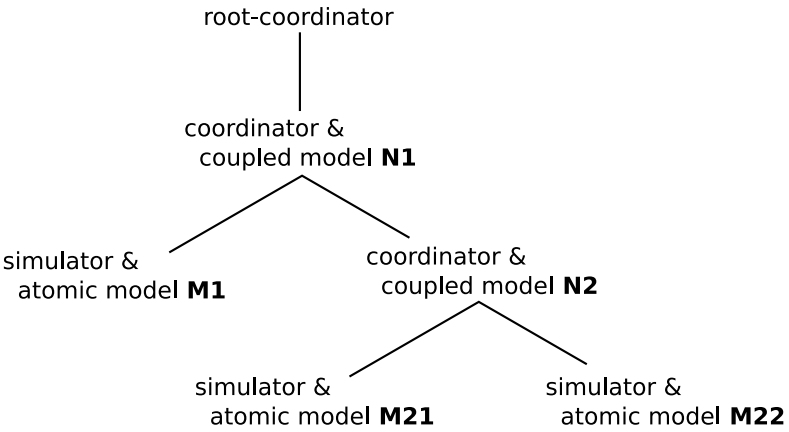


Abb. 14: Diskussionsbeispiel 2 und 3, composition-tree mit Modell- und zugeordneten Ausführungskomponenten

Auch in diesem Fall sollen die Komponenten M1 und M21 nach der Initialisierung simultan ein internes Ereignis ausführen. Das zugehörige Message Sequenz Diagramm zeigt Abbildung 15.

Es zeigt sich, dass die beiden Ausgangsereignisse von M1 und M21 nacheinander an M22 übermittelt werden und dort zweimal die externe Zustandsüberführungsfunktion ausgeführt wird.

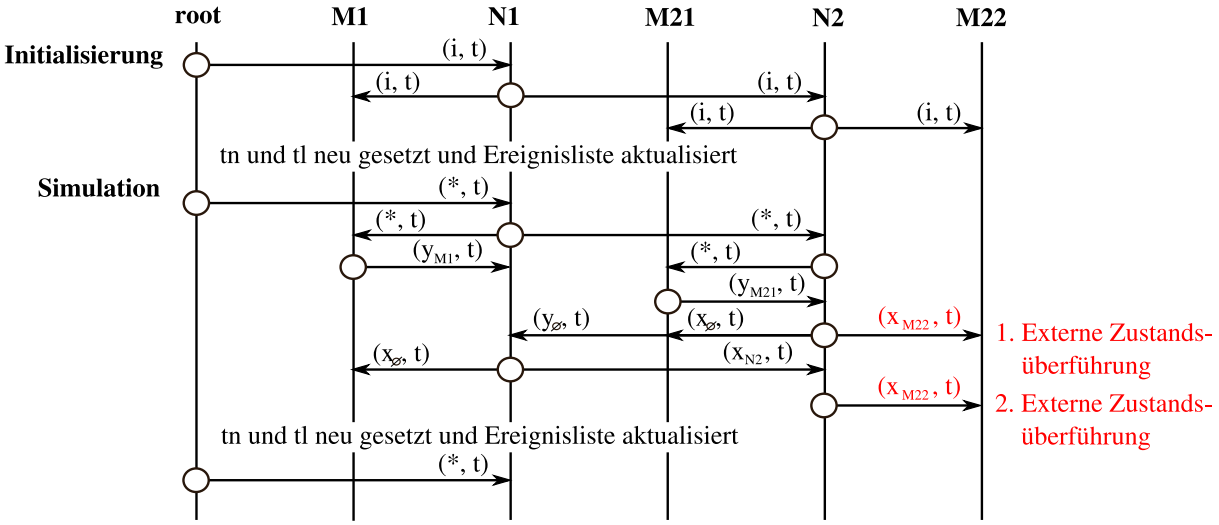


Abb. 15: Diskussionsbeispiel 2, Message Sequenz Diagram

Zuerst sendet der Koordinator N2 die Ausgangsnachricht von M21 an die Komponente M22. Erst zwei Abarbeitungsschritte später sendet er die ursprüngliche Ausgangsnachricht von M1 an M22.

Im dritten Beispiel soll das Verhalten eines Pipe-Line-Modells untersucht werden. Den zugehörigen Modellaufbau zeigt Abbildung 16 und den composition tree mit den Modellkomponenten und den zugeordneten Ausführungskomponenten zeigt Abbildungen 14.

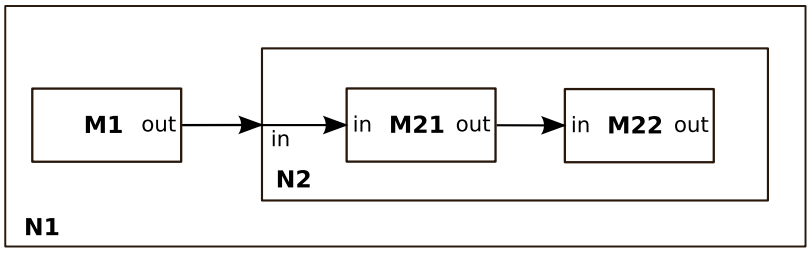


Abb. 16: Diskussionsbeispiel 3, Modellaufbau

In diesem Fall sollen die Komponenten M1 und M21 nach der Initialisierung simultan ein internes Ereignis ausführen.

Das zugehörige Message Sequenz Diagramm zeigt Abbildung 17.

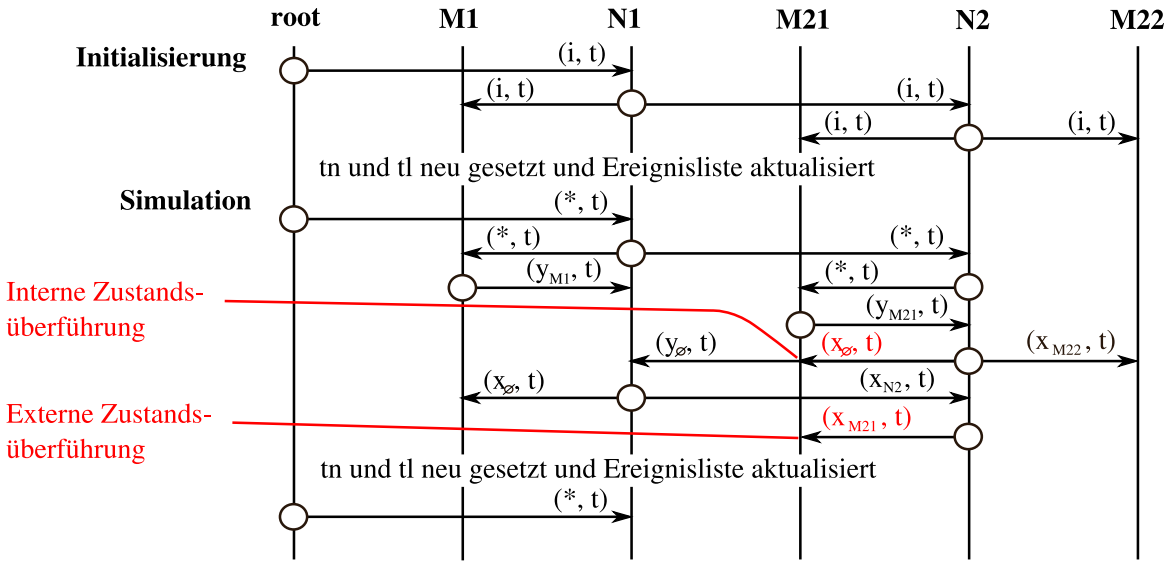


Abb. 17: Diskussionsbeispiel 3, Message Sequenz Diagram

An der Komponente M21 liegen zeitgleich das externe und das interne Ereignis vor. Deshalb muss die *confluent function* zur Ausführung kommen. Abbildung 17 zeigt aber, dass diese Zustandsüberführungsfunktion nicht ausgeführt wird. Zuerst sendet der Koordinator N2 eine Eingangsnachricht ($x_{\varnothing}$ = leere Menge) an den Simulator von M21. Daraufhin führt der Simulator die interne Zustandsüberführungsfunktion aus. Erst zwei Abarbeitungsschritte später sendet der Koordinator N2 die ursprüngliche Ausgangsnachricht von M1 mittels einer Eingangsnachricht an den Simulator von M21. Jetzt führt der Simulator die externe Zustandsüberführungsfunktion aus.

Die Beispiele in diesem Abschnitt zeigen, dass bei einer synchronen Abarbeitung durch den Algorithmus nach Zeigler, nicht in jedem Fall die Zustandsüberführungen gemäß der PDEVS Definition im Abschnitt 2 ausgeführt werden.

3.4 Modifizierter Simulationsalgorithmus für PDEVS mit Pors

Aufbauend auf den Algorithmen von Chow und Zeigler wird im Folgenden ein modifizierter Simulationsalgorithmus für PDEVS Modelle mit Ports eingeführt, welcher eine exakte Abarbeitung simultaner Ereignisse gemäß der PDEVS Definitionen im Abschnitt 2 unterstützt. Dies wird im Wesentlichen durch eine Modifikation des Nachrichtenkonzeptes und des Simulationsalgorithmus selbst erreicht. Die Simulatoren und Koordinatoren kommunizieren mittels 5 unterschiedlichen Nachrichten miteinander.

1. Initialisierungsnachricht (i, t)
2. Zustandsnachricht (*, t)
3. Eingangsnachricht (x, t)
4. Ausgangsnachricht
    a. Gewöhnliche Ausgangsnachricht (y, t)
    b. Interpellationsnachricht (’Y’, t)

Die ersten vier Nachrichten haben dieselbe Bedeutung wie bei Zeigler’s Algorithmus. Die Interpellationsnachricht nimmt hingegen eine besondere Stellung ein. Sie kann nur von einem übergeordneten Koordinator an einen untergeordneten Simulator oder Koordinator versendet werden. Als Reaktion auf eine Interpellationsnachricht (’Y’, t) führt ein Simulator die Ausgangsfunktion $y = \lambda(s)$ aus und sendet das Ausgangsereignis y mittels einer *gewöhnlichen Ausgangsnachricht* (y, t) an den übergeordneten Koordinator zurück. Ein Koordinator leitet die Interpellationsnachricht (’Y’, t) an alle Ausführungskomponenten innerhalb der Menge IMM gemäß Abschnitt 2.2 weiter.
Außerdem wird im Koordinatoralgorithmus eine interne Variable *layer* eingeführt. Sie beeinflusst das Verhalten des Koordinators beim Auftreten einer Zustandsnachricht (*, t) oder einer Ausgangsnachricht (y, t). Bei einer Zustandsnachricht (*, t) kann *layer* der Wert *active* und aufgrund einer bestimmten Ausgangsnachricht (y, t) der Wert *passive* zugewiesen werden. Die nachfolgenden abstrakten Simulatoralgorithmen zeigen das vollständige Abarbeitungsprinzip, welches anschließend an zwei Beispielen nochmals vertieft wird.

```
Parallel-DEVS-simulator

variables:
  parent  // parent coordinator
  tl      // time of last event
  tn      // time of next event
  DEVS    // associated model
     with total state (s, e)
  y       // current output value of the associated model

when receive i-message (i, t) at time t
  tl = t – e
  tn = tl + ta(s)

when receive *-message (*, t) at time t
  if t != tn then
    error: bad synchronisation
  s = δint(s)
  tl = t
  tn = tl + ta(s)

when receive y-message ('Y', t) at time t //'Y' is from typ string
  if t != tn then
    error: bad synchronization
  y = λ(s)
  send y-message (y, t) to parent coordinator

when receive x-message (x, t) at time t with input value x
  if (x != ϕ and t == tn)
    s = δcon(s, x)
    tl = t
    tn = tl + ta(s)
  else if (x != ϕ and (tl ≤ t ≤ tn))
    e = t – tl
    s = δext(s, e, x)
    tl = t
    tn = tl + ta(s)

end Parallel-DEVS-simulator
```

Parallel-DEVS-coordinator

variables:

```
  DEVN = (X, Y, D, {M_d}, {I_d}, {Z_i,d})
                // associated coupled model
  parent        // parent coordinator
  tl            // time of last event
  tn            // time of next event
  event-list    // list of elements (d, tn_d) sorted by tn_d
  IMM           // imminent children
  EXT&UN        // children receiving input but not
                   imminient plus remaining children
  layer         // layer ∈ {active, passive} – define current
                                                 layer
  mail          // output mail bag
  y_parent      // output message bag to parent
  {y_d}         // set of output message bags for each
                   child d
  'Y'           // string in output message from parent
                   to child
```

when receive i-message (i, t) at time t

```
  for-each d ∈ D do
    send i-message (i, t) to child d
  sort event-list according to tn_d
  tl = max {tl_d | d ∈ D}
  tn = min {tn_d | d ∈ D}
  layer = passive
  mail = ϕ
```

when receive *-message (*, t)

```
  if t != tn
    error: bad synchronization
  IMM = {d | (d, tn_d) ∈ (event-list & tn_d == tn)}
  if mail == ϕ
      layer = active
      for-each r ∈ IMM
        send y-message ('Y', t) to r
      //wait till each r ∈ IMM has answered
      block until layer == passive
  //check external input couplings and internal input
    couplings to get children r who receive sub-bag of y
  for each child r ∈ D with some d ∈ I_r &
  (y_d, d) ∈ mail & Z_d,r(y_d) ≠ ϕ
    for each d ∈ I_r & (y_d, d) ∈ mail & Z_d,r(y_d) ≠ ϕ
      add Z_d,r(y_d) to x_r
  EXT&UN = {d | (d, tn_d) ∈ (event-list - IMM)}
  for each r ∈ EXT&UN & x_r ≠ ϕ
    send x-message (x_r, t) to r
```

```
  for each r ∈ IMM
    if x_r != ϕ
      send x-message (x_r, t) to r
    else
      send *-message (*, t) to r
  tl = t
  tn = min {tn_d | d ∈ D}
  sort event-list according to tn_d
  mail = ϕ
```

when receive y-message (y_d, t) with output y_d from d ∈ D or parent

```
  IMM = {d | (d, tn_d) ∈ (event-list & tn_d == tn)}
  if y-message from parent    //y_d == 'Y'
    for-each r ∈ IMM
      send y-message ('Y', t) to r
  else if y-message from d    //y_d ∈ {ϕ, {y_d}}
    if this is not the last d in IMM
       add (y_d, d) to mail
       mark d as reported
    else if this is the last d in IMM
      if layer == passive
        add (y_d, d) to mail
        mark d as reported
        //check external output coupling to form sub-bag of
          parent output
        y_parent = ϕ
        for each d ∈ I_N & d has reported
          if Z_d,N(y_d) ≠ ϕ then
            add Z_d,N(y_d) to y_parent
        send y-message (y_parent, t) to parent
      else if layer == active
        add (y_d, d) to mail
        layer = passive
```

```
when receive x-message (x, t) at time t with input value x
  IMM = {d | (d, tn_d) ∈ (event-list & tn_d == tn)}
  if IMM == ϕ
    //consult external input coupling to get children
      influenced by the input
    receivers = {r | r ∈ D, N ∈ I_r, Z_N,r(x) ≠ ϕ}
    for each r in receivers
      send x-message (Z_N,r(x), t) to r
    sort event-list according to tn_d
    tl = t
    tn = min {tn_d | d ∈ D}

  else
    add (x, N) to mail
    send *-message (*, t) to N    //N is coupled DEVS itself

end Parallel-DEVS-coordinator
```

```
DEVS-root-coordinator

variables:
  t       // current simulation time
  child   // direct subordinate DEVS-simulator
             or DEVS–coordinator

t= t_0
send initialization message (i, t) to child
t = tn of its child
loop
  send (*, t) message to child
  t = tn of its child
until end of simulation

end DEVS-root-coordinator
```

Anhand des Beispiels in Abbildung 18 soll in die Funktionsweise der zuvor dargestellten Algorithmen eingeführt werden. Das Simulationsmodell besteht aus den vier atomic DEVS A1, A2, A3, A4 und aus den 3 coupled DEVS N1, N2 sowie MODEL. Abbildung 19 zeigt den composition-tree mit den Modellkomponenten und den zugeordneten Ausführungskomponenten.

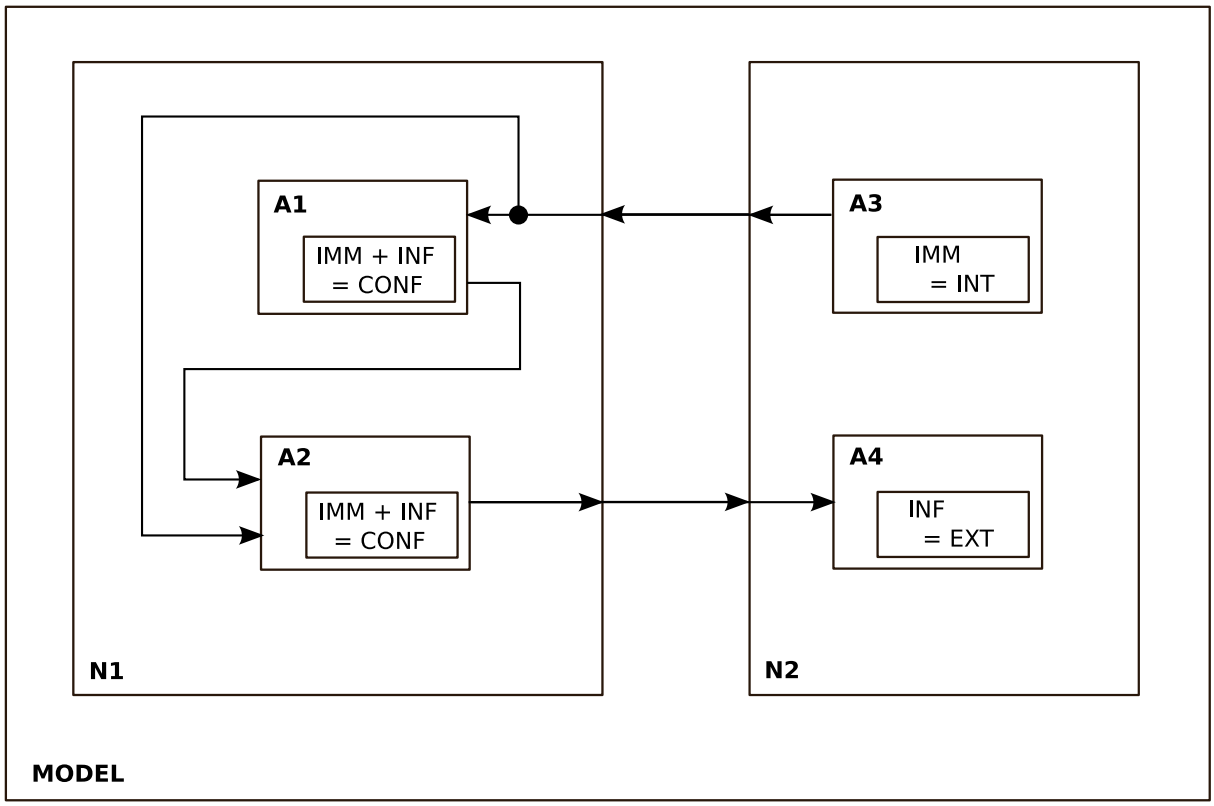


Abb. 18: Diskussionsbeispiel 4, Modellaufbau

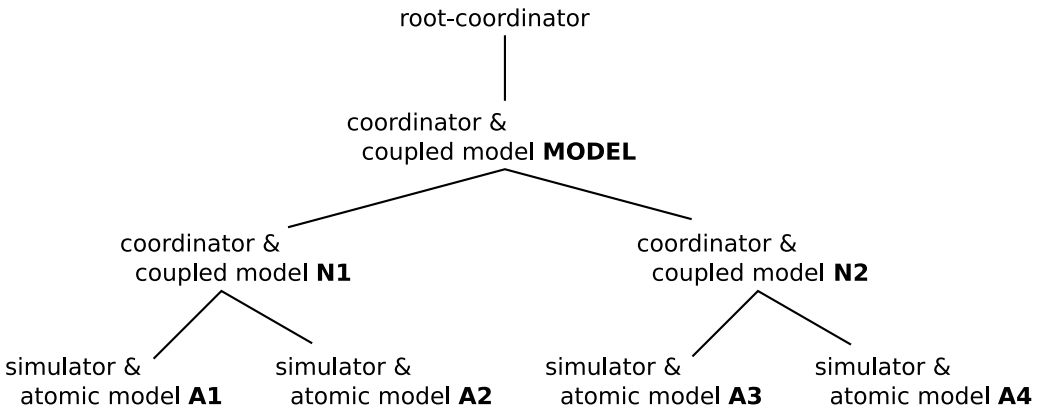


Abb. 19: Diskussionsbeispiel 4, composition-tree mit Modell- und zugeordneten Ausführungskomponenten

Nach der Initialisierung des Modells liegt bei den atomic DEVS A1, A2 und A3 simultan ein internes Ereignis vor. Die vollständige parallele Abarbeitung des Modells zeigt das Message Sequenz Diagram in Abbildung 20. Die Nachrichten in Abbildung 20 werden ausschließlich zwischen Koordinatoren und Simulatoren ausgetauscht. Zum besseren Verständnis werden für die folgenden Erläuterungen aber nur noch die Komponentennamen genutzt.

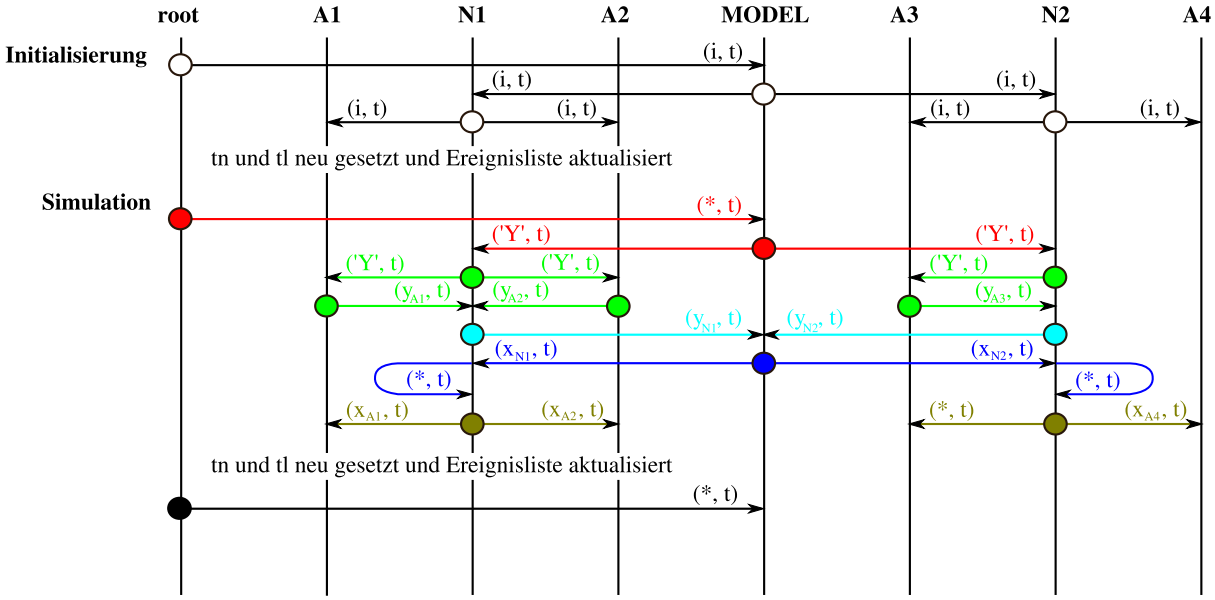


Abb. 20: Diskussionsbeispiel 4, Message Sequenz Diagram

Die Nachricht (y_{A1}, t) bedeutet, dass eine Ausgangsnachricht von Komponente A1, mit den in y_{A1} gespeicherten Ausgangsereignissen, zur Simulationszeit t versendet wird. Die Nachricht (x_{A1}, t) bedeutet, dass eine Eingangsnachricht an eine Komponente A1, mit den in x_{A1} gespeicherten Eingangsereignissen, zur Simulationszeit t versendet wird. Die Nachrichten in Abbildung 20 sind auf 5 Abarbeitungsschritte aufgeteilt und farblich voneinander abgegrenzt. Die rot markierten Nachrichten beziehen sich auf den 1. Abarbeitungsschritt, die grünen Nachrichten auf den 2. Abarbeitungsschritt, bis hin zu den grauen Nachrichten, welche sich auf den 5. Abarbeitungsschritt beziehen.

Zum Zeitpunkt t = tn_{model} sind die atomic DEVS A1, A2 und A3 imminent. Dies gilt dann auch für die coupled DEVS N1, N2 und MODEL. Zunächst sendet der root-coordinator eine *-message (*, t) an MODEL. Daraufhin werden alle IMM innerhalb MODEL ermittelt und an diese nacheinander eine y-message ('Y', t) gesendet. Abbildung 21 zeigt die ersten Abarbeitungsschritte.

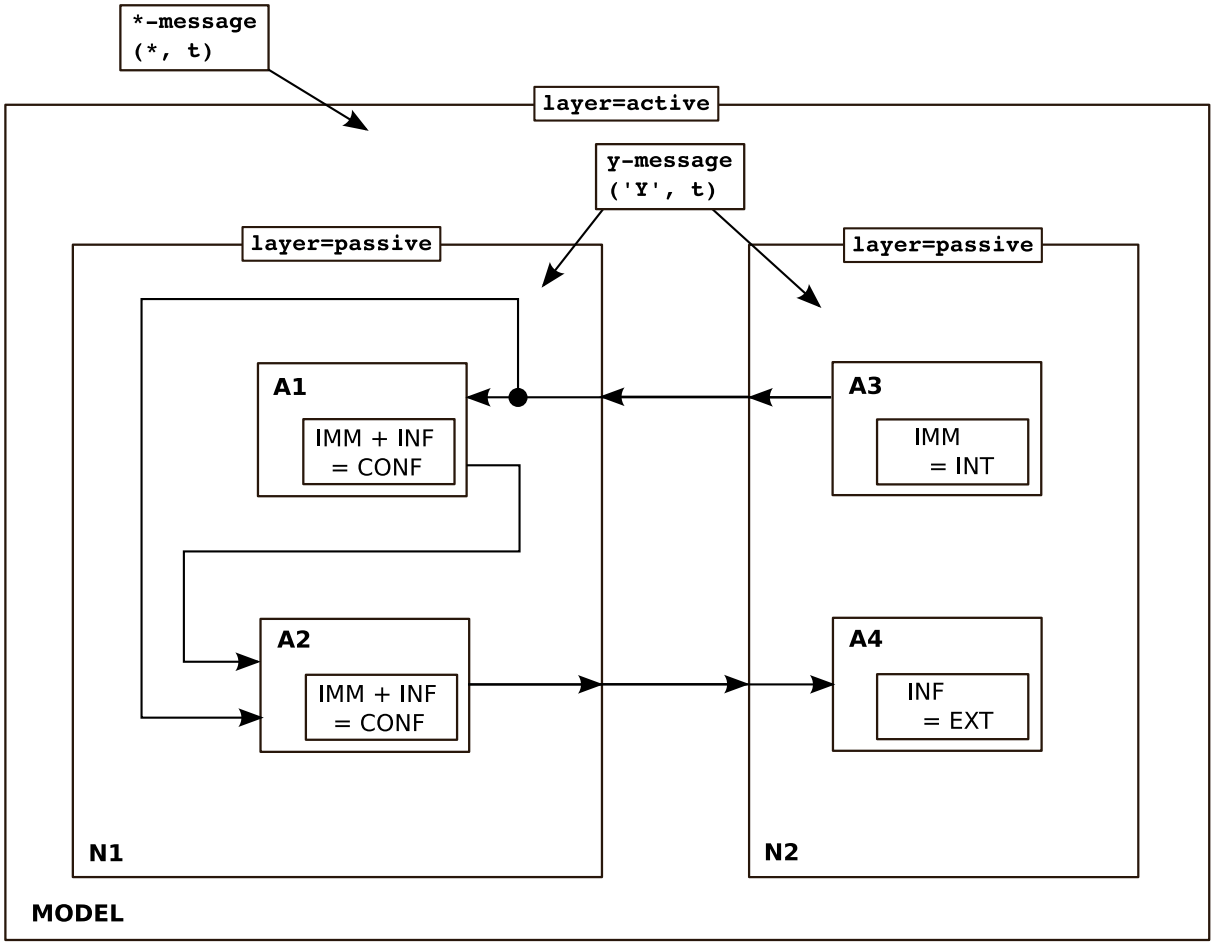


Abb. 21 : Abarbeitungsschritt 1

Da N1 und N2 ihre y-message vom *parent* (in diesem Fall MODEL) erhalten haben, senden sie ihrerseits weitere y-messages ('Y', t) an alle ihnen untergeordneten IMM und ein eventuell vorhandenes coupled DEVS wiederum eine y-message ('Y', t) an die ihm untergeordneten IMM. Dieses Verhalten setzt sich solange fort, bis man ein coupled DEVS erreicht, innerhalb dem nur noch atomic DEVS komponiert worden sind. Der Simulator des atomic DEVS führt daraufhin die λ-function aus und sendet das Ausgangsereignis mittels einer y-message (y, t) an den jeweiligen parent-coordinator zurück.

Jeder Koordinator eines coupled DEVS sammelt die Daten aller eintreffenden y-messages (y, t) in einem Containerfeld namens *mail*, wie in Abbildung 22 und 23 dargestellt.

Erst nachdem alle IMM innerhalb eines coupled DEVS (z.B. A1 und A2 innerhalb N1) ihre Ausgangsereignisse generiert haben und diese im Containerfeld *mail* abgespeichert sind,

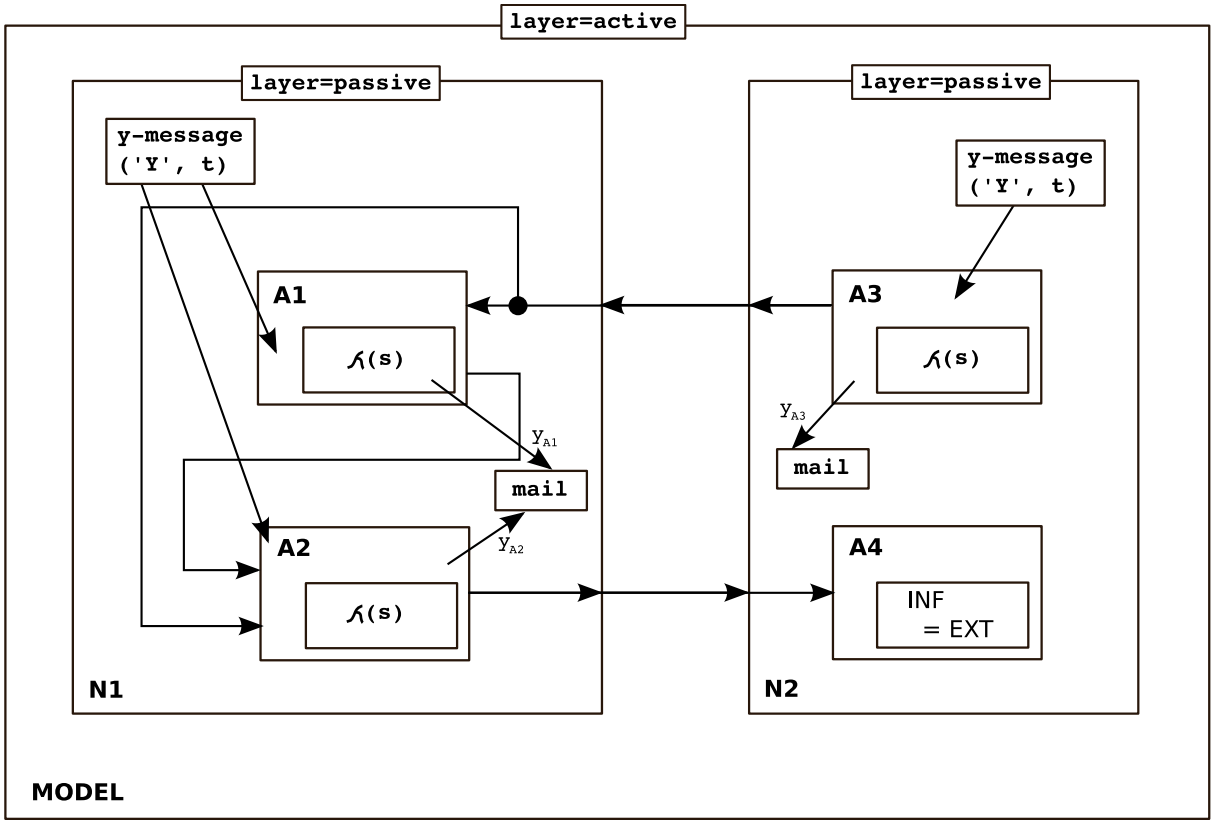


Abb. 22: Abarbeitungsschritt 2

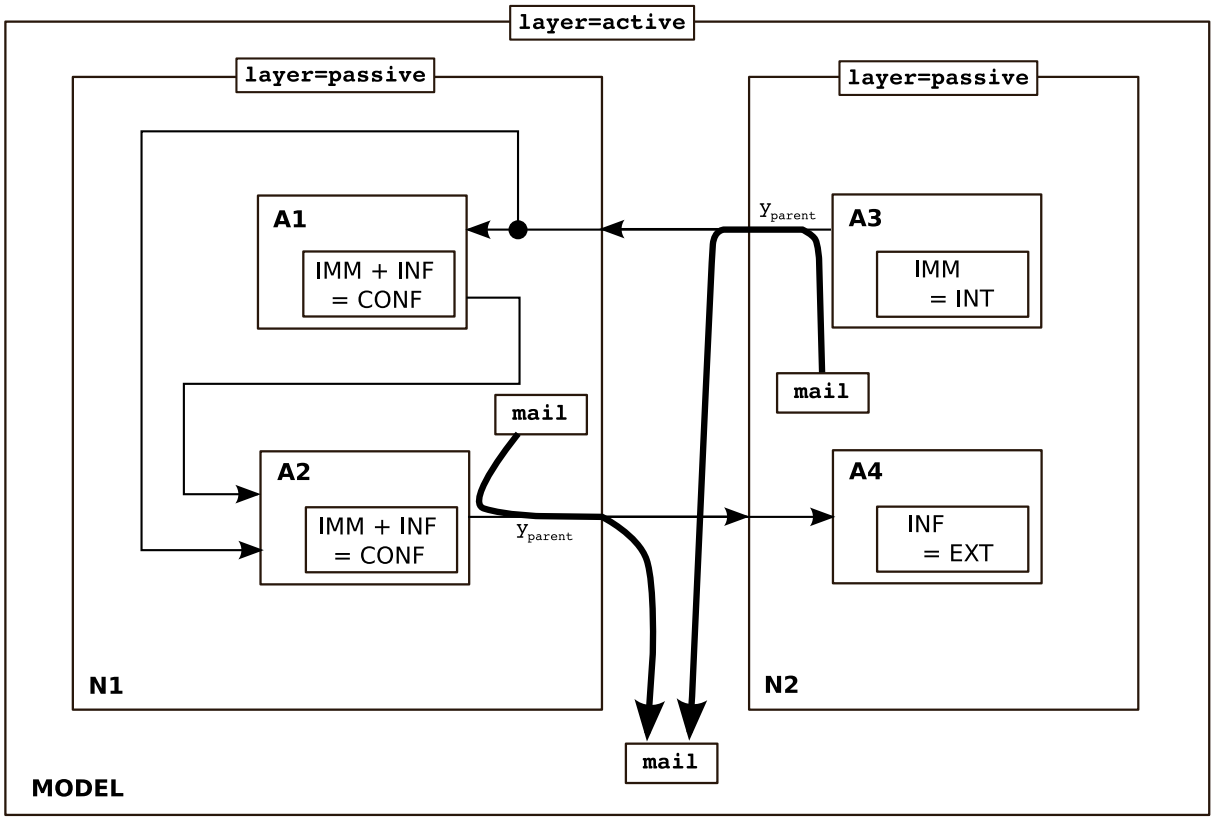


Abb. 23: Abarbeitungsschritt 3

können genau die Ausgangsereignisse, welche den parent des betrachteten coupled DEVS als Übertragungsziel haben (für N1 ist dies MODEL), in der Variablen y_{parent} zusammengefasst und an das übergordnete coupled DEVS mittels einer y-message (y_{parent}, t) versendet werden. Auf diese Art und Weise werden alle relevanten Ausgangsereignisse bis in die oberste Verarbeitungsebene, charakterisiert durch die Variable *layer* = active weitergeleitet. Somit liegen alle Ausgangsereignisse vor, welche zur Verarbeitung von MODEL notwendig sind. Zunächst einmal werden auch innerhalb von MODEL alle eingehenden y-messages (y, t) in einem Containerfeld namens *mail* abgespeichert. Danach werden die notwendigen x-messages und *-messages gesendet, wie in Abbildung 24 dargestellt. Es müssen z.B. die Ausgangsereignisse von N1 an N2 mittels einer x-message versendet und innerhalb von N2 im Containerfeld *mail* abgespeichert werden. Anschließend können N1 und N2 unabhängig voneinander verarbeitet werden und beide Koordinatoren senden sich selbst eine *-message. Damit ist das coupled DEVS MODEL zum Simulationszeitpunkt t vollständig berechnet. Es muss lediglich noch die Ereignisliste aktualisiert werden. Dazu ist es allerdings notwendig, dass auch N1 und N2 vollständig berechnet sind.

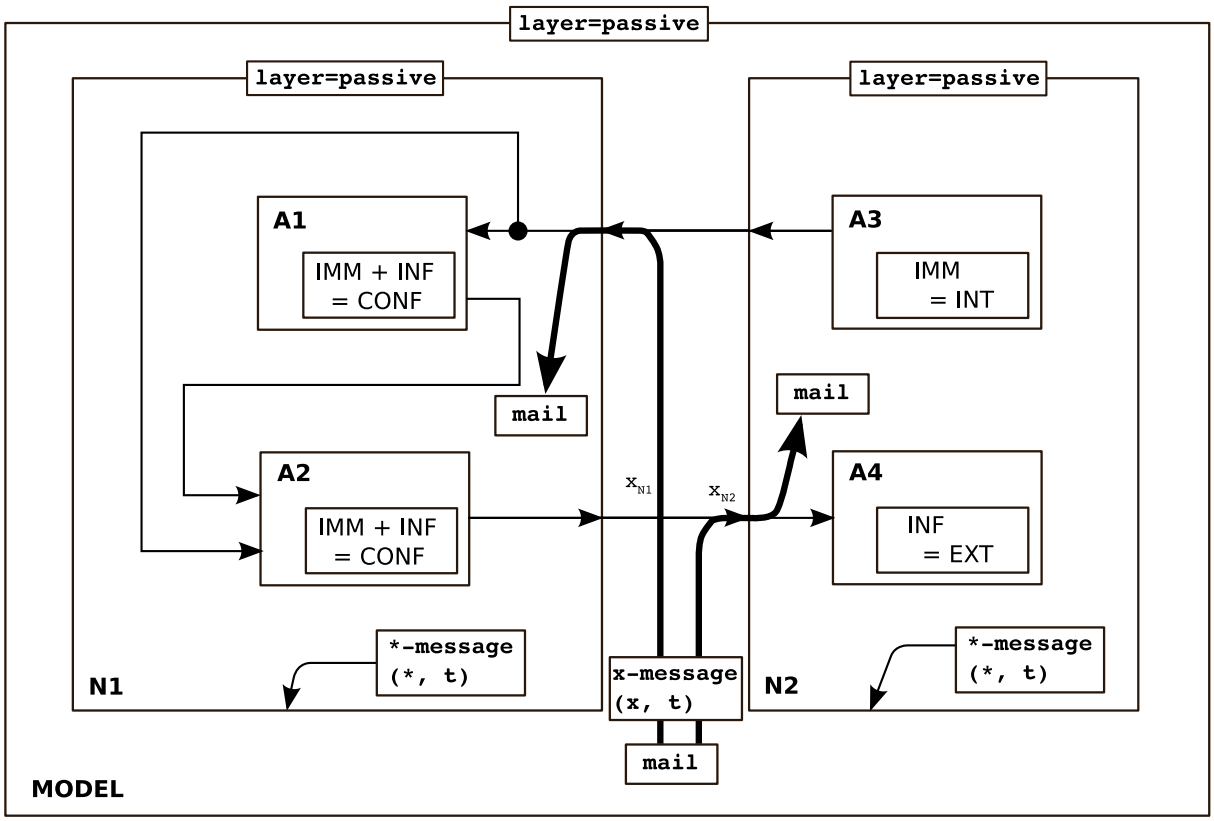


Abb. 24: Abarbeitungsschritt 4

Abschließend werden innerhalb von N1 und N2 die notwendigen x-messages und *-messages versendet und die Simulatoren der atomic DEVS können die entsprechenden Zustandsüberführungsfunktionen $\delta_{ext}(s, e, x)$, $\delta_{int}(s)$ oder $\delta_{con}(s, e, x)$ ausführen, wie in Abbildung 25 gezeigt. Damit sind auch die coupled DEVS N1 und N2 vollständig berechnet und es können in allen coupled und atomic DEVS die Ereigniszeitpunkte und in allen coupled DEVS die Ereignislisten aktualisiert werden.

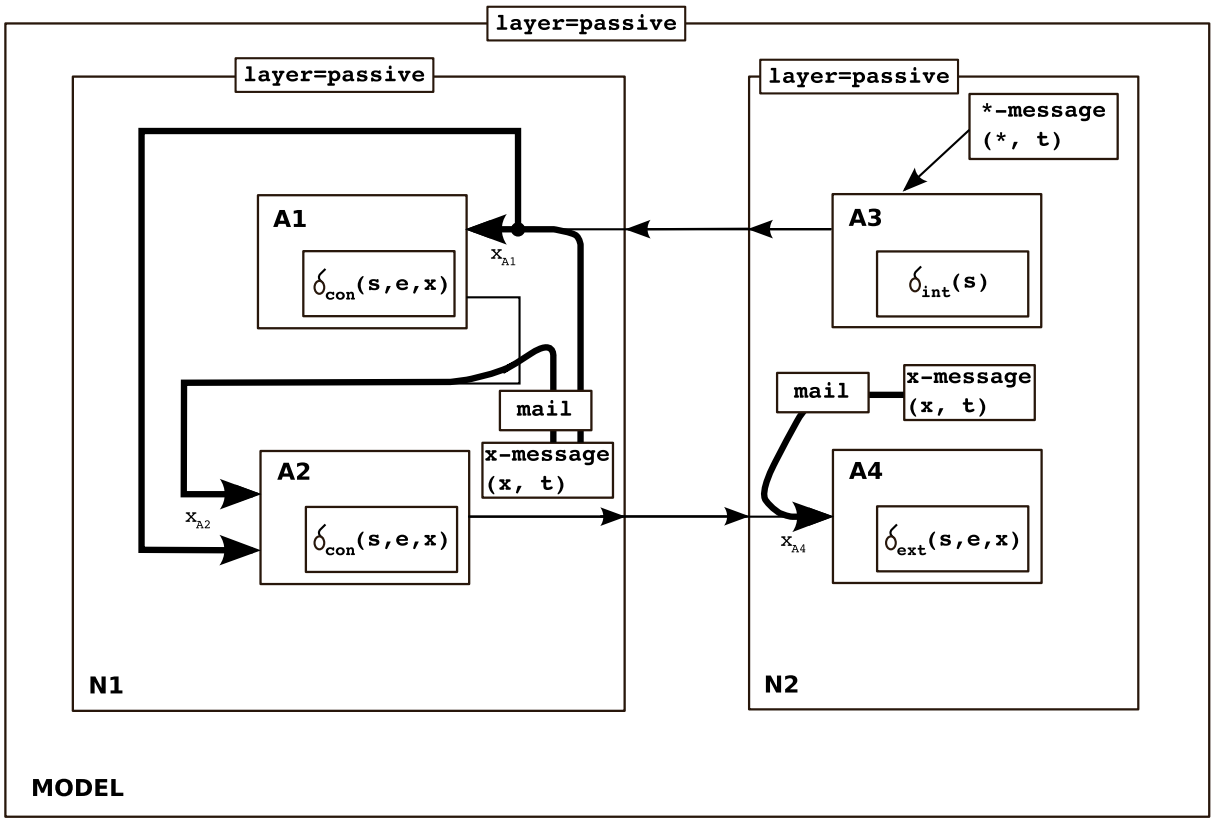


Abb. 25: Abarbeitungsschritt 5

Im nächsten Beispiel soll die Funktionsweise des modifizierten Simulationsalgorithmus für PDEVS mit Ports detaillierter dargestellt werden. Der Schwerpunkt liegt dabei auf der koordinatorinternen Liste *mail*, der Variablen *layer* und allgemein auf der Verarbeitung von atomic PDEVS mit Ports. Dafür wird das Diskussionsbeispiel 2 wieder aufgegriffen. Das zugrunde liegende Modell ist in Abbildung 13 und 14 dargestellt. Dazu zeigt Abbildung 26 die vollständige parallele Abarbeitung anhand eines Message Sequenz Diagramms. Die Komponenten M1 und M21 sollen nach der Initialisierung simultan ein internes Ereignis ausführen. Es zeigt sich, dass im Gegensatz zum Algorithmus nach Zeigler, beide Ausgangsereignisse von M1 und M12 gleichzeitig an M22 übermittelt werden. Damit wird die externe Zustangsüberführungsfunktion genau einmal bei M22 ausgelöst.

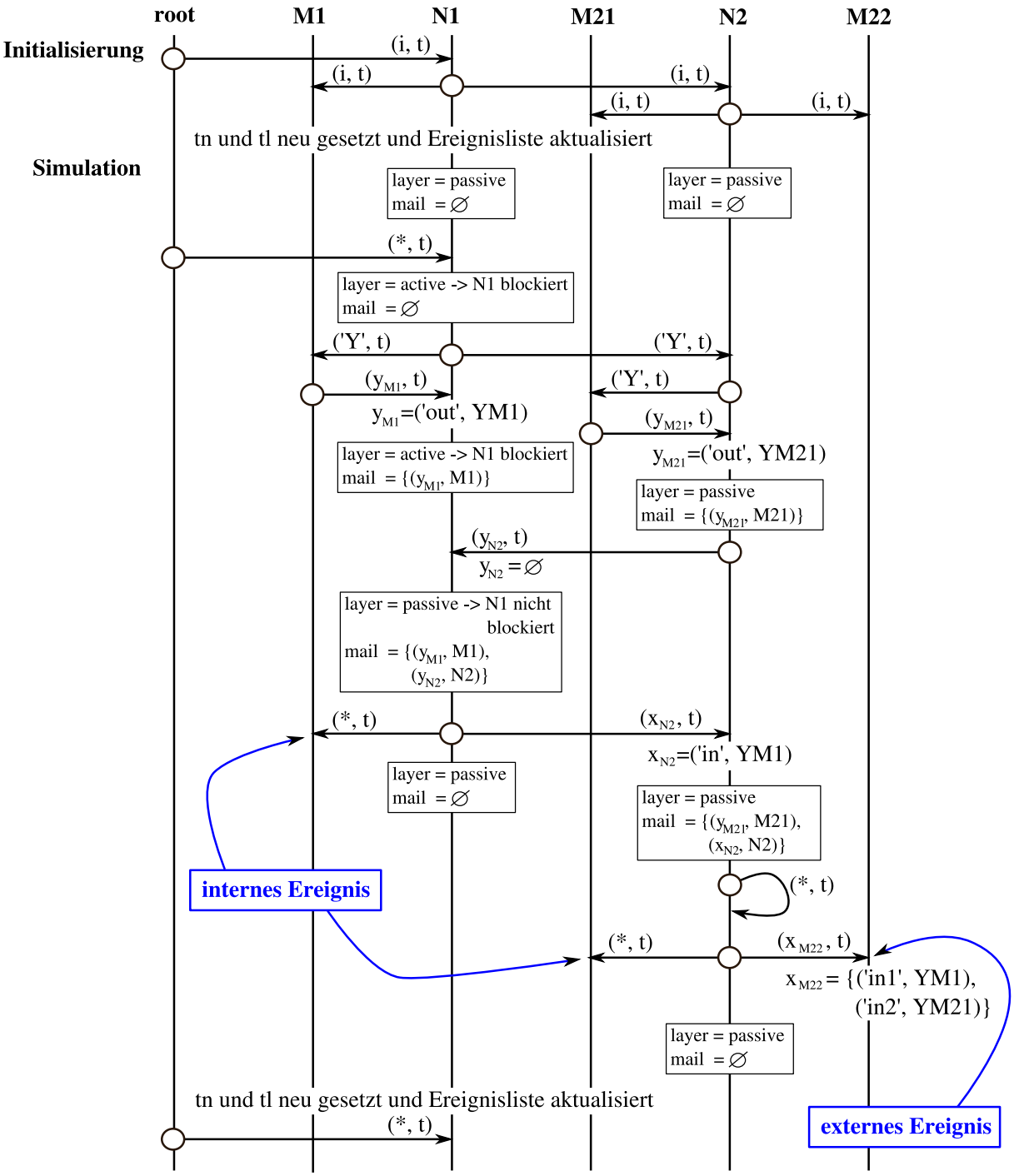


Abb. 26: Diskussionsbeispiel 6, Message Sequenz Diagramm

Zusammenfassend beinhaltet der in diesem Abschnitt eingeführte Algorithmus die Nutzung von Ports und ist ausnahmslos in der Lage simultane Ereignisse gemäß der PDEVS Definitionen in Abschnitt 2 zu verarbeiten.

4 Matlab Parallel DEVS Simulator Prototyp

In diesem Kapitel wird die grundlegende Implementierung eines DEVS Simulator Prototypen auf Basis der im Abschnitt 3.4 dargestellten Algorithmen aufgezeigt. Als Implementierungssprache wird Matlab[1] einschließlich der seit Matlab 5.3 verfügbaren OOP-Erweiterung benutzt.

4.1 Allgemeiner Aufbau

Die Implementationen des PDEVS-Simulators und des PDEVN-Koordinators basieren auf der objektorientierten Matlabprogrammierung. Die grundlegende Klassenstruktur ist in Abbildung 27 dargestellt.

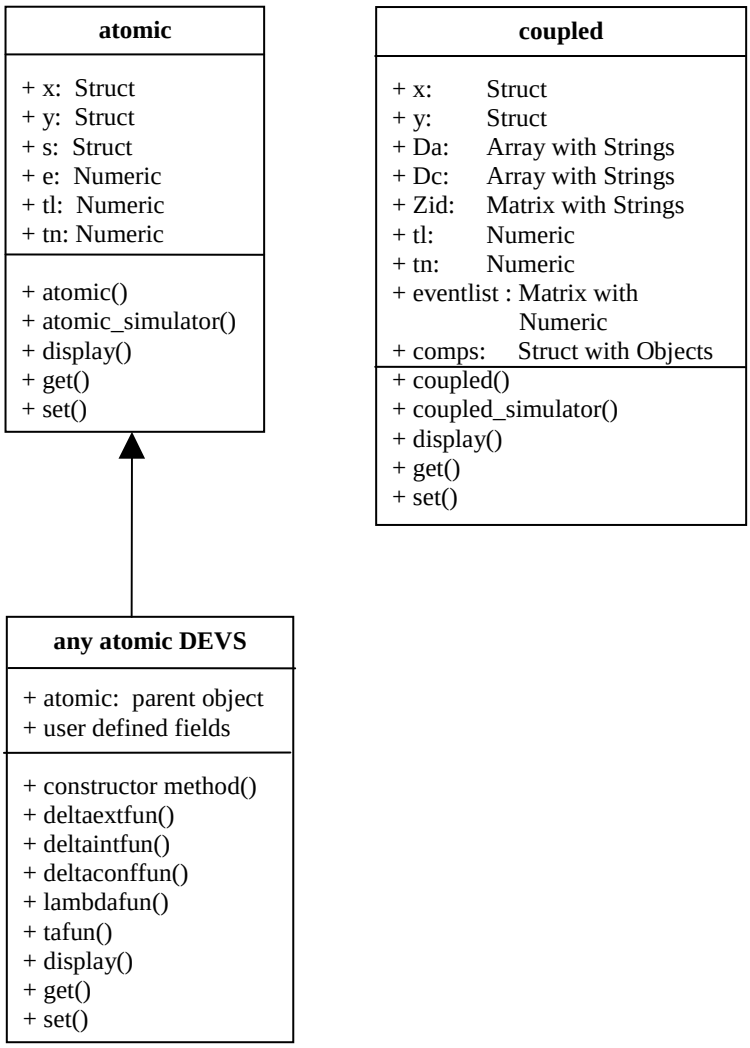


Abb. 27: Klassenstruktur im Simulator

[1] Matlab ist eingetragenes Warenzeichen von The Math Works Inc., USA

Die Klasse *atomic* stellt eine abstrakte Basisklasse dar, welche sowohl grundlegende Datenstrukturen eines atomic PDEVS mit Ports, als auch eines atomic PDEVS Simulators definiert. Die Dynamikbeschreibung eines bestimmten atomic PDEVS, mit den zugeordneten Funktionen δ_{ext}, δ_{int}, δ_{con}, λ und *ta*, wird als abgeleitete Klasse der abstrakten Basisklasse *atomic* definiert, in Abbildung 27 *any atomic DEVS* genannt.

Ein für den composition-tree nutzbares atomic DEVS muss als Inkarnation von *any atomic DEVS* erzeugt werden. Auf diese Art und Weise wird einem Objekt die Dynamikbeschreibung zugewiesen. Weiterhin muss diesem atomic-DEVS-Objekt immer eine Inkarnation von *atomic* als Elternobjekt zugewiesen werden. Auf diese Art und Weise werden die Algorithmen der Ausführungskomponenten und weitere wichtige Felder und Methoden der Elternklasse vererbt. Die Klasse *coupled* stellt ebenfalls eine Basisklasse dar, welche alle grundlegenden Datenstrukturen eines PDEVN mit Ports, als auch eines PDEVN Simulators definiert. Ein für den composition-tree nutzbares coupled DEVS muss als Inkarnation von *coupled* erzeugt werden.

Die Grundlage der Simulation stellt das objektorientierte Simulationsmodell analog zum composition-tree dar. Dieser wird stückweise, beginnend von den Zweigen bis hinauf zur Wurzel, aus atomic und coupled DEVS Objekten erzeugt. Der root-coordinator ist als Matlab-function definiert. Eine Simulation wird durch Aufruf der Funktion *devs_root_coordinator* am Matlab-Prompt ausgeführt. Der Funktion ist das Wurzelobjekt des Objektbaumes als Argument zu übergeben. Das Modellierungsprinzip und der Start einer Simulation sind in Abbildung 28 abgebildet.

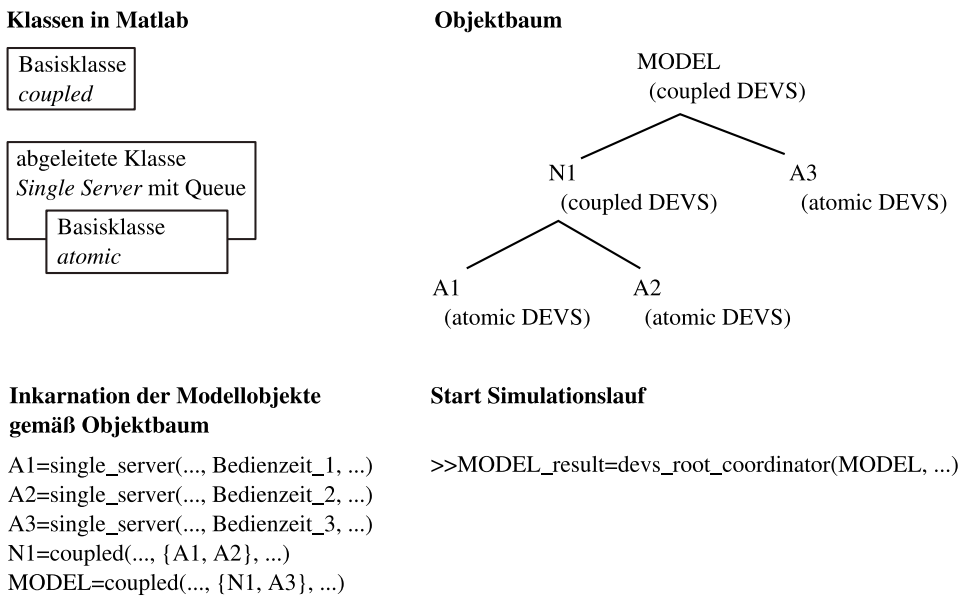


Abb. 28: Modellierungsprinzip

4.2 Spezifikas des Simulationsalgorithmus

In [UH06] wird ausgeführt, dass viele DEVS-basierte Simulationssysteme, die eine direkte Umsetzung der abstrakten Simulationsalgorithmen beinhalten, keine großen Modelle verarbeiten können. Dies wird mit Leistungsgrenzen bezüglich der Nutzung von Threads als auch Speicher begründet.

Die *parallele* Simulation eines DEVS-Modells ist laut [UH06] nur dann möglich, wenn für jeden Simulator oder Koordinator ein Thread definiert wird. Diese Zuordnung zeigt Abbildung 29.

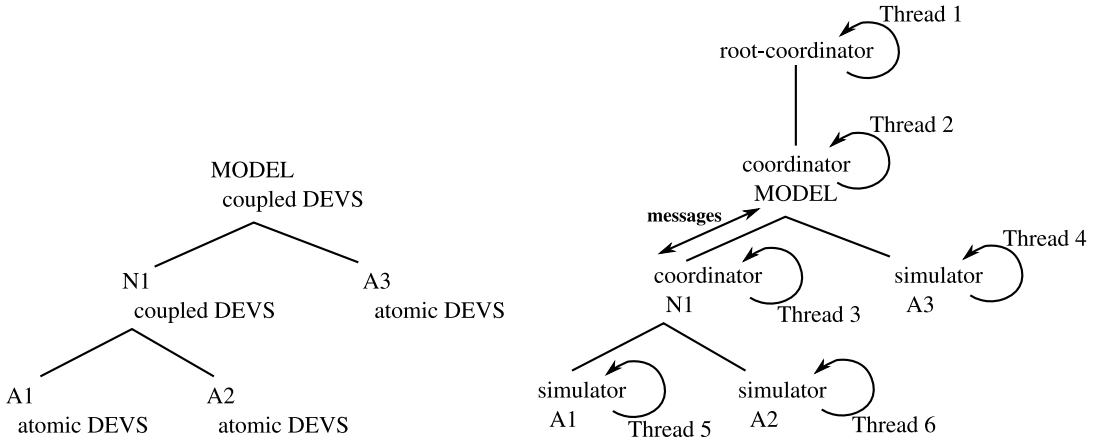


Abb. 29: Zuordnung von Threads zu den Ausführungskomponenten bei paralleler Abarbeitung

Weiterhin wird in [UH06] ausgeführt, dass *sequentielle* DEVS-Simulationsalgorithmen deutlich kürzere Laufzeiten besitzen und weitaus größere Simulationsmodelle verarbeiten können, als die in dieser Arbeit vorgestellten parallelen DEVS-Simulationsalgorithmen. Aus diesem Grund wird der zuvor formulierte parallele Simulationsalgorithmus in einen sequentiell abzuarbeitenden Simulationsalgorithmus umgeformt und implementiert. Die neue Zuordnung der Threads zeigt Abbildung 30.

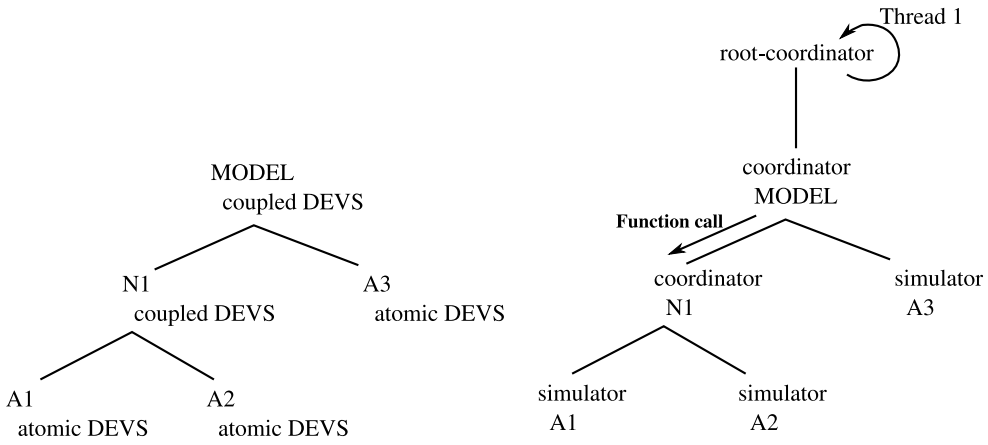


Abb. 30: Zuordnung von Threads zu den Ausführungskomponenten bei sequentieller Abarbeitung

Im Folgenden werden die wichtigsten Inhalte der Methoden *atomic_simulator()* und *coupled_simulator()* der Matlab-Implementierung abstrakt aufgeführt. Dabei wird im Gegensatz zu [UH06] das Nachrichtenkonzept beibehalten.

Parallel DEVS-simulator

function m=atomic_simulator(m_parent,m,flag,gt)

variables:

```
  m       //object any_atomic_DEVS
  m_parent     //parent object atomic
  gt      //simulation time
  tl      //time of last event
  tn      //time of next event
  e       //elapsed time
  flag    //indicates message
```

if strcmp(flag,'i_msg') //when receive i-message

```
  tl = gt – m_parent.e
  tn = tl + ta(m)
  m=set(m,'tl',tl)
  m=set(m,'tn',tn)
```

if strcmp(flag,'S_MSG') //when receive *-message

```
  m=deltaintfun(m)
  tn=gt+tafun(m)
  m=set(m,'tl',gt)
  m=set(m,'tn',tn)
```

if strcmp(flag,'y_msg') //when receive y-message

```
  m=lambdafun(m)
```

if strcmp(flag,'x_msg') //when receive x-message

```
  m=deltaextfun(m)
  tn = tl + ta(m)
  m=set(m,'tl',gt)
  m=set(m,'tn',tn)
```

end Parallel-DEVS-simulator

Parallel-DEVS-coordinator

function parent=coupled_simulation(parent,flag,gt)

variables:

```
  parent       //object coupled
  flag         //indicates message
  gt           //simulation time
  comps        //components - field for saving associated model
                 objects coupled or any_atomic_DEVS in parent
  d            //object any_atomic_DEVS or coupled
  d_parent     //object atomic
  imminents    //same as IMM
  ext&un       //same as EXT&UN
```

if strcmp(flag,'i_msg') //when receive i-message

```
  for-each d ∈ comps do
    if d is an atomic DEVS
      calculate objects d and d_parent from comps
      d=atomic_simulator(d_parent,d,'i_msg',gt)
      store d in comps
    if d is a coupled DEVS
      calculate object d from comps
      d=coupled_simulator(d,'i_msg',gt)
      store d in comps
  set eventlist, tn and tl for parent
```

if strcmp(flag,'s_msg') or strcmp(flag,'S_MSG')
//when receive *-message

```
  //carry y-message out, set and delete output/input ports
  if strcmp(flag,'s_msg') //*-message from root-coordiantor
    for-each d ∈ imminents do
      if d is an atomic DEVS
        calculate objects d and d_parent from comps
        d=atomic_simulator(d_parent,d,'y_msg',gt)
        y=get(d,'y')    //get output events from d
        transmit output events y from d to other components at
         comps
      if d is a coupled DEVS
        calculate object d from comps
        d=couplded_simulator(d,'y_msg',gt)
        y=get(d,'y')    //get output events from d
        transmit output events y from d to other components at
         comps
        delete output events y from component d and store d at
         comps
```

```
  //transmit relevant input ports from parent to its childs
  parentx=parent.x
  transmit input events parentx from parent to other
   components at comps
  //carry x-message out for relevant components in ext&un
  for each d ∈ ext&un do
    if d is an atomic DEVS
      calculate objects d and d_parent from comps
      if d has an input event
        d=atomic_simulator(d_parent,parent,'x_msg',gt)
        store d in comps
    if d is a coupled DEVS
      calculate object d from comps
      if d has an input event
        d=coupled_simulator(d,'x_msg',gt)
        store d in comps
  //carry *-message or confluent-function out for relevant
    compoents in imminents
  for each d ∈ imminents do
    if d is an atomic DEVS
      calculate objects d and d_parent from comps
      if d has no input event
        d=atomic_simulator(d_parent,d,'S_MSG',gt)
        store d in comps
      if d has input events
        d=deltaconffun(d,gt)  //confluent function
        store d in comps
    if d is a coupled DEVS
      calculate object d from comps
      d=coupled_simulator(d,'S_MSG',gt)
      store d in comps
  set eventlist, tn and tl for parent

if strcmp(flag,'y_msg')  //when receive y-message
  for each d ∈ imminents do
    if d is an atomic DEVS
      calculate objects d and d_parent from comps
      d=atomic_simulator(d_parent,d,'y_msg',gt)
      y=get(d,'y')  //get output events from d
      if y is addressed to parent
        store output events y in parent.y
      else
        transmit output events y from d to other components at
         comps
    if d is a coupled DEVS
      calculate object d from comps
      d=couplded_simulator(d,'y_msg',gt)
      y=get(d,'y')  //get output events from d
      if y is addressed to parent
        store output events y in parent.y
```

```
      else
        transmit output events y from d to other components at
         comps
      delete output events y from component d and store d at
       comps
```

```
if strcmp(flag,'x_msg')  //when receive x-message
  transmit input events parent.x to relevant childs d in comps
  for each d with input events
    if d is an atomic DEVS
      calculate objects d and d_parent from comps
      d=atomic_simulator(d_parent,d,'x_msg',gt)
      store d in comps
    if d is a coupled DEVS
      calculate objects d from comps
      d=coupled_simulator(d,'x_msg',gt)
      store d in comps
  set eventlist, tn and tl for parent

end Parallel-DEVS-coordinator
```

5 Beispiele zur Modellierung und Simulation mit DEVS

In diesem Kapitel wird anhand von zwei Beispielen die Modellierung und Simulation mittels dem im Kapitel 4 vorgestellten Simulatorprototypen aufgezeigt. Beim ersten Beispiel handelt es sich um eine Montagestraße. Hier wird die grundlegende Modellierung und Abarbeitung des Modells durch den sequentialisierten Simulationsalgorithmus aufgezeigt. Das zweite Beispiel stellt eine Robotersteuerung dar. Hier wird aufbauend auf dem Simulationsmodellbasierten Steuerungsansatz gezeigt, wie reale diskret-ereignisorientierte Steuerungen mit einem DEVS-Modell umgesetzt werden können.

5.1 Beispiel: Montagestraße

Ausgehend von der grundlegenden Modellstruktur werden nachfolgend die Modellierung der einzelnen Komponenten und ein einfaches Simulationsexperiment aufgezeigt.

5.1.1 Modellstruktur

Mit diesem Beispiel soll die Modellierung von atomic DEVS und coupled DEVS aufgezeigt werden. Die prinzipielle Modellstruktur zeigt Abbildung 31. Es ist eine Montagestraße abgebildet, in der mittels 6 Montageautomaten vom Typ *processing_block* ein Fertigteil aus sieben Grundkomponenten montiert wird. Die Grundkomponenten werden durch 7 Generatoren vom Typ *generator* erzeugt. Die Fertigteile werden durch einen Transducer vom Typ *transducer* gezählt und vernichtet. Die Komponenten werden im Folgenden erläutert.

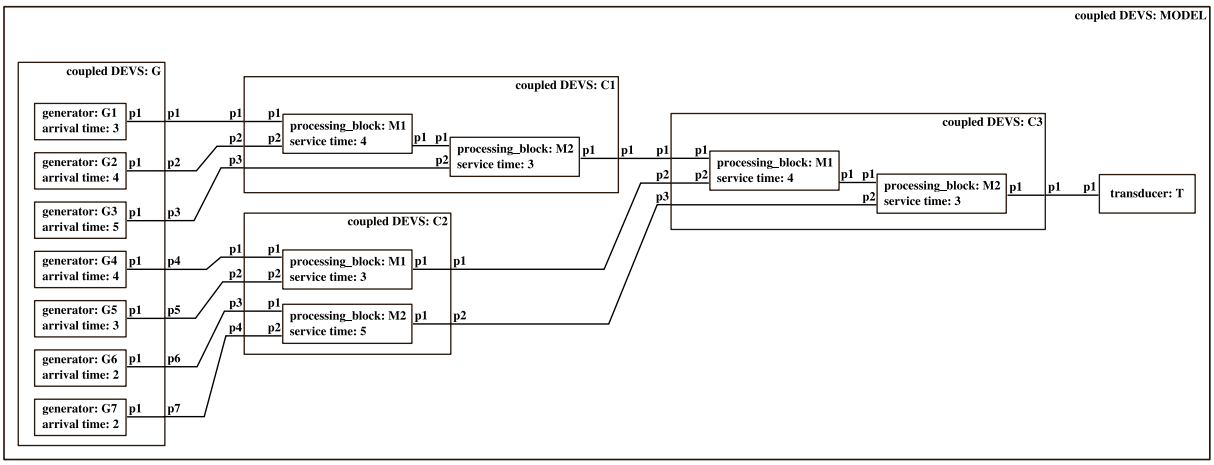


Abb. 31: Modellaufbau Montagestaße

5.1.2 Spezifikation der Modellkomponenten

In diesem Abschnitt wird die Systemspezifikation aller verwendeten atomic DEVS definiert. Auf Grundlage dieser Spezifikationen müssen innerhalb Matlab spezielle Objektklassen angelegt werden.

Generator

Der Generator stellt ein sehr einfaches atomic DEVS dar. Er besitzt keine Eingänge und erzeugt periodisch ein Ausgangsereignis. Die Zeitspanne zwischen zwei internen Ereignissen ist mittels der Variablen *arrival time* definiert.

Modellbeschreibung ohne Ports:

$DEVS_{generator} = \{X, Y, S, \delta_{int}, \lambda, ta\}$

$X = \{ \}$

$Y = \{1\}$

$S = R^+$

initial state $s = (\sigma) =$(*arrival time*)

$\delta_{int}(\sigma) = (\sigma)$

$\lambda(\sigma) = 1$

$ta(\sigma) = \sigma$

Transducer

Der Transducer stellt ebenfalls ein sehr einfaches atomic DEVS dar. Er besitzt keine Ausgänge und ist zu jedem Zeitpunkt in der Lage ein Eingangsereignis zu empfangen und zu vernichten. Außerdem zählt er die Anzahl aller vernichteten Eingangsereignisse mittels einer Variablen $q \in N$.

Modellbeschreibung ohne Ports:

$DEVS_{transducer} = \{X, Y, S, \delta_{ext}, ta\}$

$X = N$

$Y = \{ \}$

$S = N$

inital state $s = (q) = (0)$

$\delta_{ext}(q, e, x) = (q + 1)$

$ta(q) = inf$

Processing block

Der Processing block ist ein atomic DEVS und modelliert einen Montageautomaten, welcher innerhalb der Zeitspanne *service time* ein Werkstück aus zwei verschiedenen Komponenten fertigt. Er besitzt zwei voneinander getrennte Eingangsbuffer q1 und q2, in der getrennt beliebig viele Grundkomponenten zwischengespeichert werden können.

Modellbeschreibung mit Ports:

$PDEVS_{processing\_block} = \{X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta\}$

IPorts = {“p1”, “p2”}, where $X_{p1} = X_{p2} = N$

$X = \{(p, v)| p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“p1”}, where $Y_{p1} = N$

$Y = \{(p, v)| p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{\text{“active”, “passive”}\} \times R_{0,\infty}^{+} \times N \times N$

inital state $s = (phase, \sigma, q1, q2) = (\text{"passive"}, inf, 0, 0)$

δ_{ext}(phase, σ, q1, q2, e, (("p1", x1), ("p2", x2)))

= ("passive", *inf*, q1+x1, q2) if phase = = "passive" & ("p1", x1) $\in$ X & ("p2", x2) $\notin$ X & q2 = = 0

= ("passive", *inf*, q1, q2+x2) if phase = = "passive" & ("p1", x1) $\notin$ X & ("p2", x2) $\in$ X & q1 = = 0

= ("active", *service time*, q1+x1-1, q2-1) if phase = ="passive" & ("p1", x1) $\in$ X & ("p2", x2) $\notin$ X & q2 $\neq$ 0

= ("active", *service time*, q1-1, q2+x2-1) if phase = = "passive" & ("p1", x1) $\notin$ X & ("p2", x2) $\in$ X & q1 $\neq$ 0

= ("active", *service time*, q1+x1-1, q2+x2-1) if phase = = "passiv" & ("p1", x1) $\in$ X & ("p2", x2) $\in$ X

= ("active", σ-e, q1+x1, q2) if phase = = "active" & ("p1", x1) $\in$ X & ("p2", x2) $\notin$ X

= ("active", σ-e, q1, q2+x2) if phase = = "active" & ("p1", x1) $\notin$ X & ("p2", x2) $\in$ X

= ("active", σ-e, q1+x1, q2+x2) otherwise

δ_{int}(phase, σ, q1, q2)

= ("active", *service time*, q1-1, q2-1) if q1 $\geq$ 1 & q2 $\geq$ 1

= ("passive", *inf*, q1, q2) otherwise

δ_{con}(phase, σ, q1, q2, e, x) = δ_{ext}(δ_{int}(phase, σ, q1, q2), 0, x)

λ(phase, σ, q1, q2) = ("p1", 1)

ta(phase, σ, q1, q2) = σ

Die Beschreibung der coupled DEVS erfolgt z.T. abweichend zur formalen Theorie nach Zeigler in [ZPK00] und orientiert sich an der Syntax des im Kapitel 4 entwickelten Koordinators.

Komponenten C1 und C3

$PDEVN_{C1} = PDEVN_{C3} = \{X, Y, Da, Dc, Z_{id}, comps\}$

IPorts = {“p1”, “p2”, ”p3”}, where $X_{p1} = X_{p2} = X_{p3} = N$

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“p1”}, where $Y_{p1} = N$

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

Da = {“M1”, “M2”} //atomic DEVS

Dc = ∅ //coupled DEVS

Z_{id} = { (“parent”,”p1”,”M1”,”p1”),
(“parent“,“p2“,“M1“,“p2“),
(“parent“,“p3“,“M2“,“p2“),
(“M1“,“p1“,“M2“,“p1“),
(“M2“,“p1“,“parent“,“p1“)}

comps = {*M1*, *M2*} //where *m1* and *m2* are objects of type *processing block*

Komponente C2

$PDEVN_{C2} = \{X, Y, Da, Dc, Z_{id}, comps\}$

IPorts = {“p1”, “p2”, “p3”, “p4”}, where $X_{p1} = X_{p2} = X_{p3} = X_{p4} = N$

$X = \{(p, v)| p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“p1”, “p2”}, where $Y_{p1} = Y_{p2} = N$

$Y = \{(p, v)| p \in OPorts, v \in Y_p\}$ is the set of output ports and values

Da = {“M1”, “M2”} //atomic DEVS

Dc = ∅ //coupled DEVS

Z_{id} = { (“parent”,”p1”,”M1”,”p1”),
(“parent“,“p2“,“M1“,“p2“),
(“parent“,“p3“,“M2“,“p1“),
(“parent“,“p4“,“M2“,“p2“),
(“M1“,“p1“,“parent“,“p1“),
(“M2“,“p1“,“parent“,“p2“)}

comps = {*M1, M2*} //where *m1* and *m2* are objects of type *processing block*

Komponente G

$PDEVN_G = \{X, Y, Da, Dc, Z_{id}, comps\}$

X = ∅

OPorts = {“p1”, “p2”, “p3”, “p4”, “p5”, “p6”, “p7”}, where $Y_{p1}, \ldots, Y_{p7} = N$

$Y = \{(p, v)| p \in OPorts, v \in Y_p\}$ is the set of output ports and values

Da = {“G1”, “G2”, “G3”, “G4”, “G5”, “G6”, “G7”} //atomic DEVS

Dc = ∅ //coupled DEVS

Z_{id} = { (“G1“,“p1“,“parent“,“p1“),
(“G2“,“p1“,“parent“,“p2“),
(“G3“,“p1“,“parent“,“p3“),
(“G4“,“p1“,“parent“,“p4“),
(“G5“,“p1“,“parent“,“p5“),
(“G6“,“p1“,“parent“,“p6“),
(“G7“,“p1“,“parent“,“p7“)}

comps = {*G1, G2, G3, G4, G5, G6, G7*} //all components are objects of typ *generator*

Komponente MODEL

$PDEVN_{model} = \{X, Y, Da, Dc, Z_{id}, comps\}$

$X_M = \varnothing$

$Y_M = \varnothing$

Da = {“T”} //atomic DEVS

Dc = {“G”, “C1“, “C2“, “C3“} //coupled DEVS

Z_{id} = { (“G“,“p1“,“C1“,“p1“),
(“G“,“p2“,“C1“,“p2“),
(“G“,“p3“,“C1“,“p3“),
(“G“,“p4“,“C2“,“p1“),
(“G“,“p5“,“C2“,“p2“),
(“G“,“p6“,“C2“,“p3“),
(“G“,“p7“,“C2“,“p4“),
(“C1“,“p1“,“C3“,“p1“),
(“C2“,“p1“,“C3“,“p2“),
(“C2“,“p2“,“C3“,“p3“),
(“C3“,“p1“,“T“,“p1“)

comps = {*G, C1, C2, C3, T*} //where *G, C1, C2* and *C3* are objects of typ *coupled* and *T* is an object from typ *transducer*

Der Objektbaum mit den Modell- und Ausführungskomponenten ist für die Montagestraße in Abbildung 32 dargestellt.

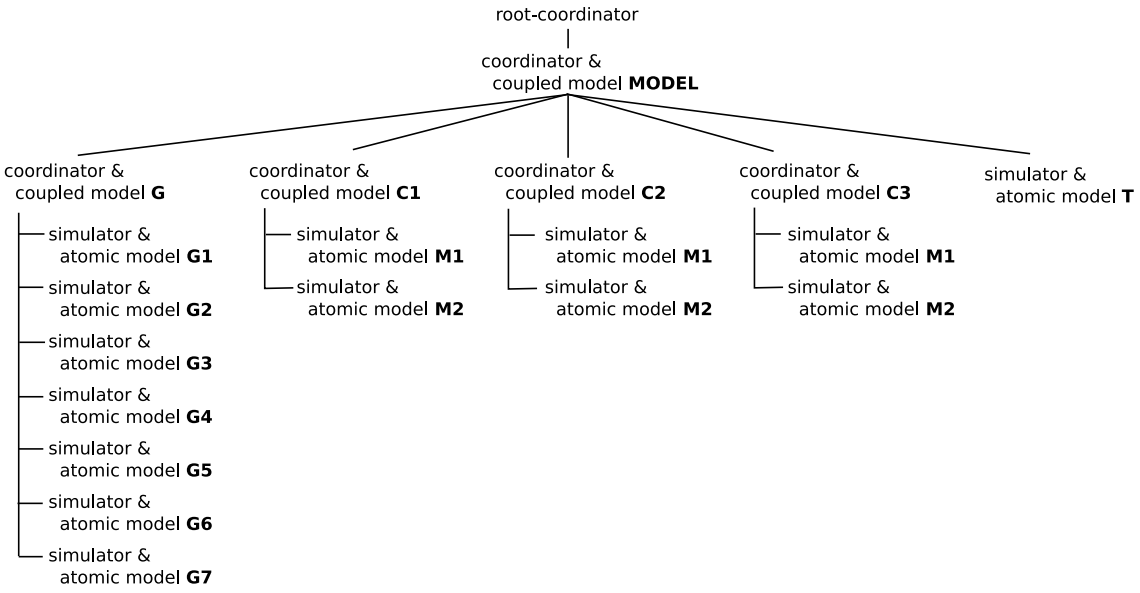


Abb. 32: Objektbaum der Montagestraße

5.1.3 Simulationsergebnisse

Die Simulation des Montagestraßenmodells wurde zum einen von Hand und zum anderen auf dem Rechner durchgeführt und miteinander verglichen. Die Ergebnisse der Handsimulation, für einen Zeitraum von t = 0 bis t = 12, sind für die PDEVN *C1* bis *C3* in Tabelle 1 aufgeführt. Das Eintreten eines externen Ereignisses an Port *p1* oder *p2* eines Montageautomaten ist durch eine gelbe oder blaue Markierung der zugehörigen Zustandsfelder *q1* bzw. *q2* gekennzeichnet. Der Zahlenwert innerhalb dieser Zustandsfelder stellt die aktuelle Belegung der Eingangsbuffer *q1* und *q2* dar. Das Eintreten des internen Ereignisses eines Montageautomaten, welches das Bearbeitungsende anzeigt, wird durch eine grüne Markierung des zugehörigen Zustandsfeldes *sigma* gekennzeichnet. Die genaue Bedeutung der einzelnen Zustände ist der Modellspezifikation für den Processing block zu entnehmen.

Tab. 1: Zustände der Modellkomponenten für das Montagestraßenmodell, Teil 1

simulation time	0	2	3	4	5	6
coupled DEVS C1						
M1 (service time=4)						
phase	passive	passive	passiv	active	active	active
sigma	inf	inf	inf	4	4	2
q1	0	0	1	0	0	1
q2	0	0	0	0	0	0
M2 (service time=3)						
phase	passive	passive	passive	passive	passive	passive
sigma	inf	inf	inf	inf	inf	inf
q1	0	0	0	0	0	0
q2	0	0	0	0	1	1
coupled DEVS C2						
M1 (service time=3)						
phase	passive	passive	passive	active	active	active
sigma	inf	inf	inf	3	3	1
q1	0	0	0	0	0	0
q2	0	0	1	0	0	1
M2 (service time=5)						
phase	passive	active	active	active	active	active
sigma	inf	5	5	3	3	1
q1	0	0	0	1	1	2
q2	0	0	0	1	1	2
coupled DEVS C3						
M1 (service time=4)						
phase	passive	passive	passive	passive	passive	passive
sigma	inf	inf	inf	inf	inf	inf
q1	0	0	0	0	0	0
q2	0	0	0	0	0	0
M2 (service time=3)						
phase	passive	passive	passive	passive	passive	passive
sigma	inf	inf	inf	inf	inf	inf
q1	0	0	0	0	0	0
q2	0	0	0	0	0	0

Tab. 2: Zustände der Modellkomponenten für das Montagestraßenmodell, Teil 2

simulation time	7	8	9	10	11	12
coupled DEVS C1						
M1 (service time=4)						
phase	active	active	active	active	active	active
sigma	2	4	3	3	3	4
q1	1	0	1	1	1	1
q2	0	0	0	0	0	0
M2 (service time=3)						
phase	passive	active	active	active	passive	active
sigma	inf	3	3	1	inf	3
q1	0	0	0	0	0	0
q2	1	0	0	1	1	0
coupled DEVS C2						
M1 (service time=3)						
phase	passive	active	active	active	passive	active
sigma	inf	3	2	2	inf	3
q1	0	0	0	0	0	0
q2	1	0	1	1	1	1
M2 (service time=5)						
phase	active	active	active	active	active	active
sigma	5	4	4	2	2	5
q1	1	2	2	3	3	3
q2	1	2	2	3	3	3
coupled DEVS C3						
M1 (service time=4)						
phase	passive	passive	passive	passive	active	active
sigma	inf	inf	inf	inf	4	4
q1	0	0	0	0	0	0
q2	1	1	1	1	1	1
M2 (service time=3)						
phase	passive	passive	passive	passive	passive	passive
sigma	inf	inf	inf	inf	inf	inf
q1	0	0	0	0	0	0
q2	1	1	1	1	1	2

Weiterhin muss zum Simulationszeitpunkt t = 18 ein Ausgangsreignis vom PDEVN *C3* an den Transducer *T* gesendet werden, da der kritische Pfad der Montagestraße vom Generator G2 zum Transducer T verläuft und sich nach Abbildung 31 zu 4 + 4 + 3 + 4 + 3 = 18 berechnet. Für die Computersimulation muss das im Anhang A1 hinterlegte Initialisierungsskript *init.m* des Montagestraßenbeispiels ausgeführt werden. Dieses Skript erzeugt und parametriert alle notwendigen Komponenten und speichert das vollständige hierarchische Modell auf der Variablen *model*. Anschließend muss mittels der Funktion *devs_root_coordinator* die Simulation gestartet und zum Schluss das Ergebnis auf einer Variablen rückgespeichert werden. Die folgenden Codeausschnitte zeigen den Aufruf, die Durchführung und die Auswertung des Simulationsbeispiels.

```
init
tstart = 0;
tend = 12;
erg_model = devs_root_coordinator(model,tstart,tend);
```

Nun lassen sich mittels der Methode *get()* die Ergebnisse[1] zum Simulationsende vergleichen:

```
    get(erg_model,'comps','am_t')
```
→ für Komponente transducer *T*

oder

```
    get(get(erg_model,'comps','cm_c1'),'comps','am_m1')
```
→ für Komponente processing block *M1* innerhalb *C1*

Ein Vergleich der Ergebnisse der Computersimulation mit den Ergebnissen der Handsimulation zeigt eine vollständige Übereinstimmung.

5.2 Beispiel: Robotersteuerung

Im zweiten Beispiel soll auf Grundlage des Simulationsmodellbasierten Steuerungsansatzes der FG CEA [MPP06], mittels DEVS, eine einfache Robotersteuerung modelliert und simuliert werden. Es wird die vorliegende Prozess- und Steuerungsstruktur aufgezeigt und anschließend die Modellierung sowie ein Steuerungsexperiment erläutert.

5.2.1 Prozess- und Steuerungsstruktur

Der grundlegende Simulationsmodellbasierte Steuerungsansatz nach [MPP06] ist in Abbildung 33 dargestellt.

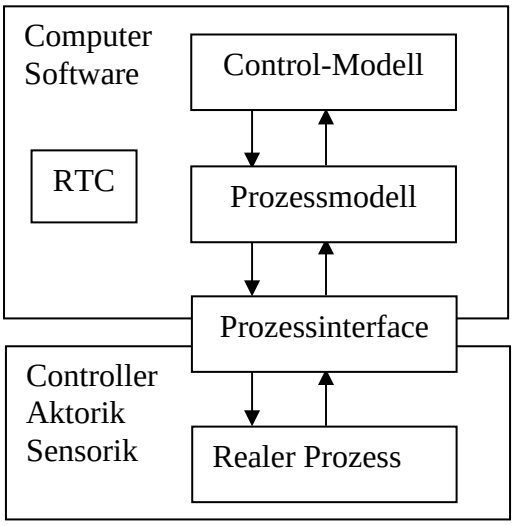


Abb. 33: Impliziter Simulationsmodellbasierter Steuerungsansatz nach [MPP06]

Das Steuerungssystem besteht aus einem Control-Modell, einem Prozessmodell und einem Prozessinterface.

[1] in der Beispielimplementierung sind den Komponentennamen die Bezeichnungen *am* und *cm* für *atomic DEVS* und *coupled DEVS* vorgesetzt

Das *Prozessinterface* stellt die Schnittstelle zwischen dem realen zu steuernden Prozess und dem *Steuerungssystem* dar. Im *Prozessmodell* sind die Elemente und Reaktionen des realen Prozesses modelliert. Das *Control-Modell* bildet die gesamte Steuerungslogik ab. Die einzelnen Ebenen kommunizieren miteinander, wie in Abbildung 33 dargestellt. Außerdem kann das Steuerungssystem eine Real-Time-Clock (*RTC)* beinhalten. Diese wird beim betrachteten Beispiel nicht benötigt, da die zu realisierende Steuerung keine zeitabhängigen Ereignisse beinhaltet und damit nur eine korrekte Ereignisfolge zu realisieren ist.

Der zu steuernde Roboter besitzt vier Freiheitsgrade und hat einen einfachen Fingergreifer als Endeffektor. Die Achsen lassen sich über die serielle Schnittstelle RS232 mittels eines Controllers ansteuern. Abbildung 34 zeigt den Roboter mit den vier Gleichstrommotoren M1… M4 und den acht Endlagentastern, bzw. Positionstastern E1… E8. Die Endlagentaster begrenzen den Verfahrweg jeder Achse in eine Richtung und dienen als Referenz. Die Positionstaster erlauben eine Auflösung des Ankerdrehwinkels von 45° und dienen der Robotersteuerung.

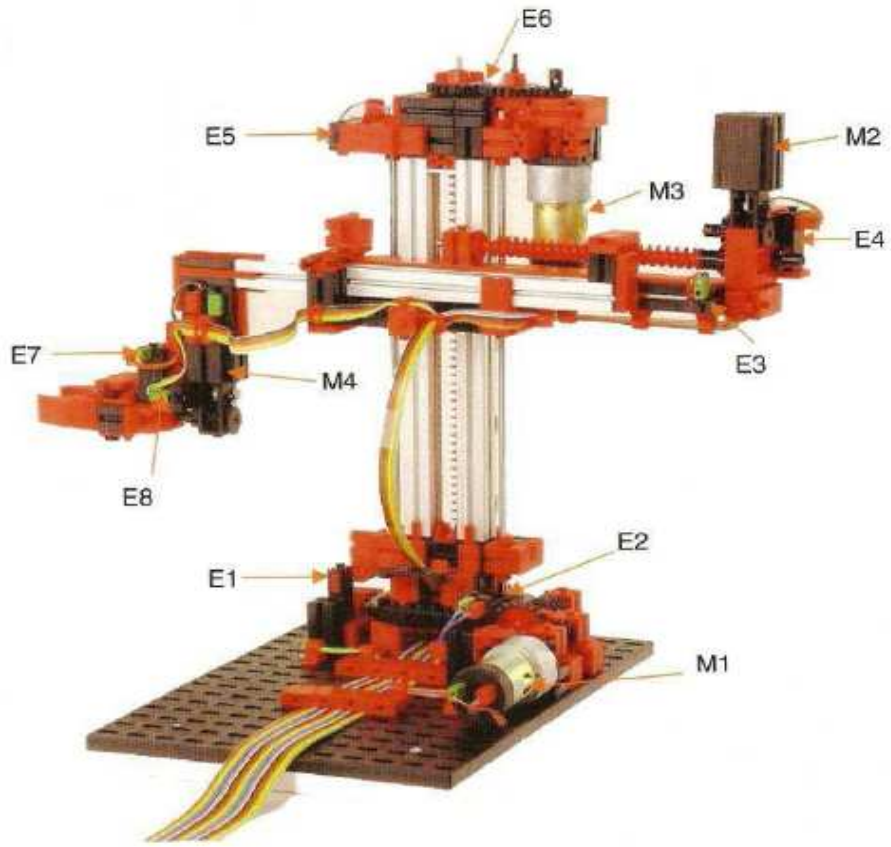


Abb. 34: Fischertechnik Roboter

Das realisierte Steuerungssystem hat den in Abbildung 35 dargestellten Modellaufbau.
Der Objektbaum mit den Modell- und Ausführungskomponenten ist für das Steuerungssystem in Abbildung 36 abgebildet.

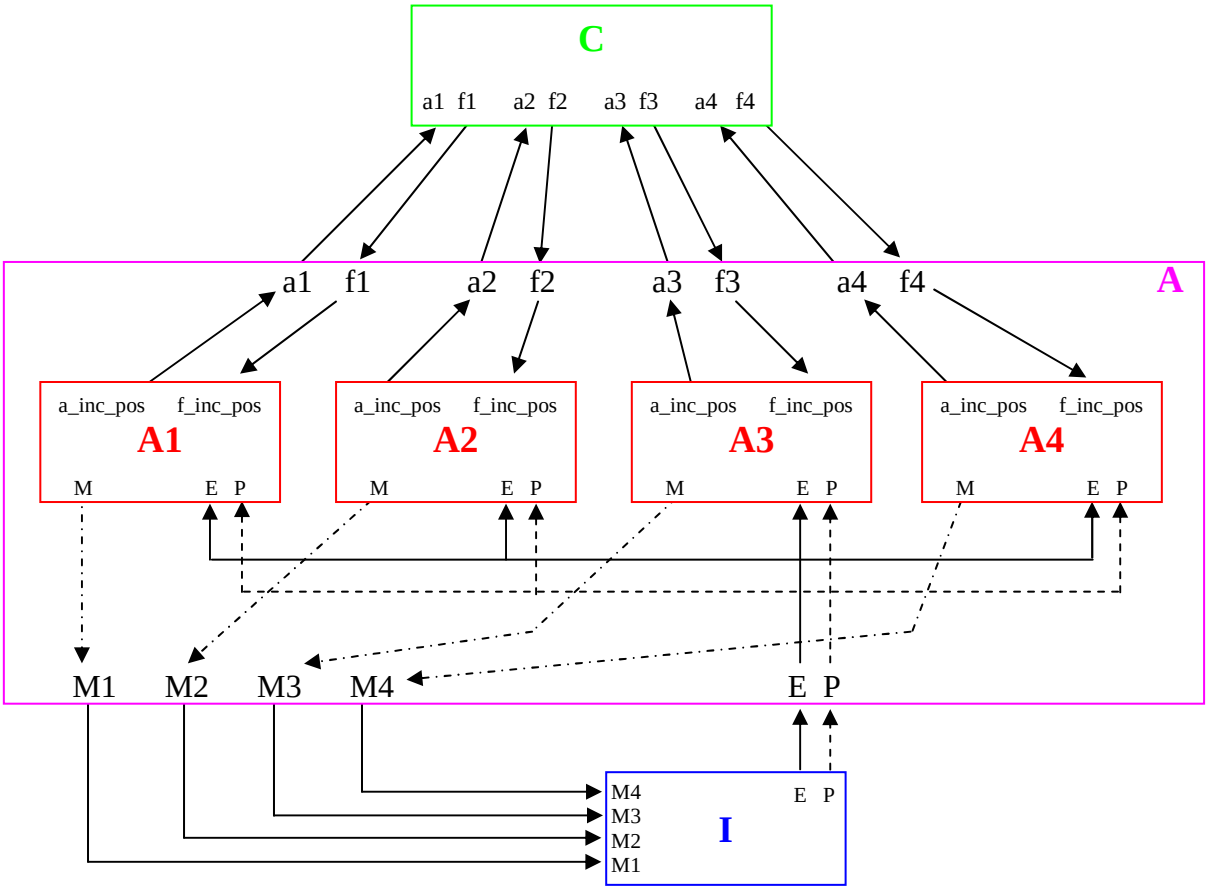


Abb. 35: Modellaufbau Steuerungssystem Roboter

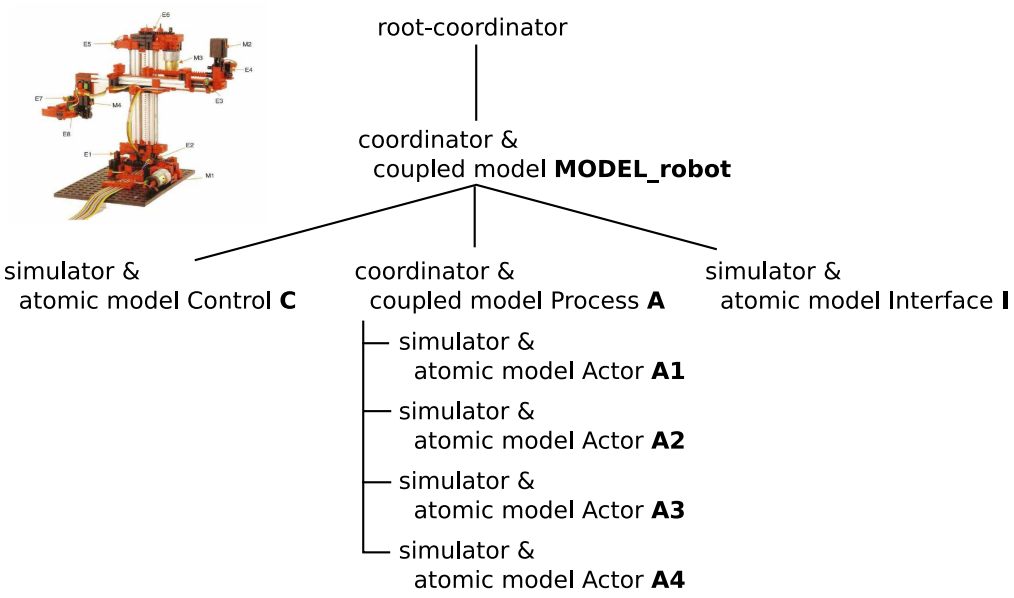


Abb. 36: Objektbaum Steuerungssystem Roboter

5.2.2 Spezifikation der Modellkomponenten

In diesem Abschnitt werden die Grundkomponenten des Beispiels spezifiziert. Das Augenmerk liegt auf der Dynamikbeschreibung der atomic DEVS.

Interface *I*

Das Prozessinterface des Robotersystems besteht aus einem Roboter-Controller und einem atomic DEVS Interface, wie in Abbildung 37 dargestellt. Das atomic DEVS Interface *I* empfängt die vom übergeordneten Prozessmodell gesetzten Motorsteuerbefehle M1… M4, tranformiert diese und sendet sie an den Robotercontroller.

M1, ..., M4 ∈ {1; 0; -1}

1 Bewegung weg von Referenz

0 Stillstand

-1 Bewegung hin zur Referenz

Außerdem werden die Zustände der Endlagentaster *E* und der Positionstaster *P* vom Roboter-Controller, über das atomic DEVS Interface *I*, an das übergeordnete Prozessmodell übergeben.

E=[E1, ..., E4], E1, ..., E4 ∈ {0; 1}

0 Endlagentaster betätigt

1 Endlagentaster nicht betätigt

P=[P1, ..., P4], P1, ..., P4 ∈ {0; 1}

0 Positionstaster nicht betätigt

1 Positionstaster betätigt

Die Kommunikation mit dem Controller des Roboters erfolgt periodisch, mit der konstanten Zykluszeit *sampling_time* = 1.[1]

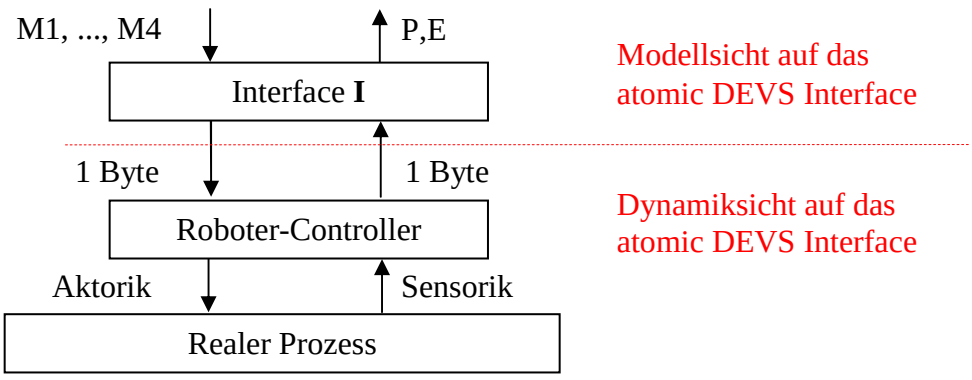


Abb. 37: Dynamik des atomic DEVS Interface

1 Spezifik des Roboter-Controllers

$DEVS_I = \{X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta\}$

IPorts = {“M1”, “M2”, “M3”, “M4”}, where $X_{M1}, ..., X_{M4} \in \{-1, 0, 1\}$

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“E”, “P”}, where Y_E, Y_P are $\{matrix_{(1,4)} | components \in \{0, 1\}\}$

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{matrix_{(1,4)} | components \in \{-1, 0, 1\}\} \times R_0^+$

initial state s = ([engine control commands], σ) = ([0, 0, 0, 0], 0)

δ_{ext}([SM1, SM2, SM3, SM4], σ, x, e)

```
//save engine control commands (ECC) in s
SM1=x1          if  (“M1”, x1) ∈ x
SM2=x2          elseif  (“M2”, x2) ∈ x
SM3=x3          elseif  (“M3”, x3) ∈ x
SM4=x4          elseif  (“M4”, x4) ∈ x
```

=([SM1, SM2, SM3 SM4], σ-e)

δ_{int}([SM1, SM2, SM3, SM4], σ) = ([SM1, SM2, SM3, SM4], *sampling_time*)

$\delta_{con}(s, x, e) = \delta_{int}(\delta_{ext}(s, x, e))$

λ(s)

```
send ECC[1] (SM1, ...,SM4) to controller
waiting for answer from controller
create vectors E and P
```

= ((“E“, *E*), (“P“, *P*))

ta([SM1, SM2, SM3, SM4], σ) = σ

Das Senden der Motorsteuerbefehle ECC an den Roboter-Controller erfolgt in der *λ-function*. Ebenso wird dort die Antwort des Controllers verarbeitet und über eine *y-message* des zugeordneten Simulators an höhere Steuerungsebenen weitergeleitet. Weiterhin muss beachtet werden, dass die Controller-Anwort unter keinen Umständen als Eingangsereignis verstanden werden darf, da dies eine Rückkopplung des Interface-DEVS auf sich selbst bedeuten würde.

[1] ECC – engine control commands (Motorsteuerbefehle)

Actors *A1 – A4*

Die atomic DEVS A1 bis A4 ergeben sich aus den vier Freiheitsgraden des Roboters. In diesen Komponenten werden die jeweiligen Aktoren und Sensoren einer Bewegungsachse modelliert. Daher wird anstelle der in der Robotik üblichen Bezeichnung *Axes* die Bezeichnung *Actor* verwendet. Jede Bewegungsachse wird durch je einen Endlagentaster *E*, einen Positionstaster *P* und dem Bewegungszustand *M* des Gleichstrommotors repräsentiert. Weiterhin muss jeder Komponente eine Indexvariable *el_number* eingeprägt werden, da die vom Interface empfangenen Nachrichten *P* und *E* die Menge aller Zustände, der vorhandenen Taster, in einem Vektor zusammengefasst enthalten. Außerdem gibt es Zustandsfelder für die aktuelle Position *a_inc_pos*, die Zielposition der betrachteten Bewegungsachse *f_inc_pos* und den maximalen Verfahrweg der Bewegungsachse *upper_limit*.

$DEVS_{A1, ...,A4} = \{X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta\}$

IPorts = {“P”, “E“, “f_inc_pos”}, where X_E, X_P are $\{matrix_{(1,4)} |$ components $\in \{0, 1\}\}$

and $X_{f\_inc\_pos} \in N$

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“M”, “a_inc_pos”}, where $Y_M \in \{-1, 0, 1\}$ and $Y_{a\_inc\_pos} \in N$

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{1, 0\} \times N \times N \times R_{0,\infty}^{+}$

initial state s = (P, a_inc_pos, f_inc_pos, σ) = (0, *upper_limit*, 0, *inf*)

δ_{ext}(P, a_inc_pos, f_inc_pos, σ, x, e)

```
//P → pushed = 1, not pushed = 0
//E → pushed = 0, not pushed = 1 (negative logic)
σ = inf
//when receive input from control
if (“f_inc_pos”, x3) ∈ x
  σ = 0
  f_inc_pos = x3   //x3 ≤ upper_limit
```

```
//when input message from interface
if ("E", x1) & ("P", x2) ∈ x
  //normal behaviour
  if x2(el_number) ≠ P   //compute new position
    P = x2(el_number)
    a_inc_pos = a_inc_pos
                + sign(f_inc_pos – a_inc pos)
    if a_inc_pos == f_inc_pos
      σ = 0  //immediately stop motor
  //check for control mistakes
  elseif x1(el_number) == 0
    if a_inc_pos ≠ f_inc_pos
      a_inc_pos = 0  //compute new ECC
      σ = 0
  elseif a_inc_pos == 0 & x1(el_number) == 1
    a_inc_pos = 1
```

=(P, a_inc_pos, f_inc_pos, σ)

δ_{int}(P, a_inc_pos, f_inc_pos, σ) = (P, a_inc_pos, f_inc_pos, *inf*)

δ_{con}(s, x, e) = δ_{ext}(s, x, e)

λ(P, a_inc_pos, f_inc_pos, σ)

```
M = sign(f_inc_pos – a_inc_pos)
```

=(("M", *M*), ("a_inc_pos", a_inc_pos))

ta(P, a_inc_pos, f_inc_pos, σ) = σ

Control *C*

Im atomic DEVS *C* ist die Steuerungslogik abgebildet. In der Matrix $traj_{(N,4)}$ wird eine geordnete Menge aller anzufahrenden Punkte definiert. Die Raumpunkte werden nacheinander durch asynchrone Point-to-Point-Bewegungen angefahren.

traj = [$\mathbf{A_{i,j}}$]$_{(N, 4)}$, $\mathbf{A_{i\,1}, ..., A_{i\,4}}$: Zielposition i der Achsen 1 bis 4

Die Variable *index* kennzeichnet die Position des aktuell anzufahrenden Punktes innerhalb von *traj*. Die Modellkomponente *C* besitzt weiterhin für jeden Aktor *A1 – A4* separate Ein und Ausgänge, damit die gewünschte Zielposition (f1, ..., f4) gesendet und die aktuell vorhandene Achsenposition (a1, ..., a4) empfangen werden kann.

$DEVS_C = \{X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta\}$

IPorts = {“a1”, “a2”, “a3”, “a4”}, where $X_{a1,} ..., X_{a4} = N$

$X = \{(p, v)| p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“f1”, “f2” , “f3” , “f4”}, where $Y_{f1}, ..., Y_{f4} = N$

$Y = \{(p, v)| p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{matrix_{(1,4)} | components \in N\} \times N \times R_{0,\infty}^{+}$

initial state s = ([aktuelle Position der 4 Aktoren], index, σ) = ([1, 1, 1, 1], 1, 1)

δ_{ext}([a_inc1, a_inc2, a_inc3, a_inc4], index, σ, x, e)

```
//save values from actors as states
if (“a1”, x1) ∈ x
  a_inc1 = x1
elseif (“a2”, x2) ∈ x
  a_inc2 = x2
elseif (“a3”, x3) ∈ x
  a_inc3 = x3
elseif (“a4”, x4) ∈ x
  a_inc4 = x4
//arrival at final point?
elseif [a_inc_1,...,a_inc_4] == traj( index, (1, ...,4) )
  if this not the last point in traj
    ++index
    σ = 0
  else
    σ = inf
```

=([a_inc1, a_inc2, a_inc3, a_inc4], index, σ)

δ_{int}(a_inc_pos, index, σ) = (a_inc_pos, index, *inf*)

$\delta_{con}(s, x, e) = \delta_{ext}(s, x, e)$

λ([a_inc1, a_inc2, a_inc3, a_inc4], index, σ)

```
f1 = traj(index, 1)
f2 = traj(index, 2)
f3 = traj(index, 3)
f4 = traj(index, 4)
```

=((“f1“, *f1*), (“f2“, *f2*), (“f3“, *f3*), (“f4“, *f4*))

ta(a_inc_pos, index, σ) = σ

5.2.3 Experimente

Für die Steuerungsinitialisierung muss das im Anhang A2 hinterlegte Initialisierungsskript *init_c.m* ausgeführt werden. Dieses Skript erzeugt und parametriert alle notwendigen Komponenten und speichert das vollständige hierarchische Modell auf der Variablen *model_robot.*

Anschließend kann mittels der Funktion *devs_root_coordinator* die Steuerung gestartet werden und am Bewegungsende der Zustand der Steuerung auf einer Variablen rückgespeichert werden. Die folgenden Codeausschnitte zeigen den Aufruf, die Durchführung und die Auswertung der Robotersteuerung in Matlab.

```
init_c;
tstart = 0;
tend = 2500;
erg_model =devs_root_coordinator(model_robot,tstart,tend);
```

Der Roboter fährt nacheinander alle Punkte gemäß der Matrix *traj* an und bleibt in der letzten Position stehen. Es empfiehlt sich, als ersten Punkt die Referenz [0, 0, 0, 0] anzufahren.

Es können beliebige Punktemengen im Feld *traj* der Komponente *C* abspeichert werden. Dies verdeutlicht der nachfolgende Code.

```
traj = [0, 0, 0, 0; 20, 20,10, 10; 40, 30, 50, 38];
new_model_robot = set(model_robot, 'traj', traj, {'c'});
```

Jetzt kann die neue Bahnkurve abgefahren werden:

```
tstart = 0;
tend = 500;
erg_model =
       devs_root_coordinator(new_model_robot,tstart,tend);
```

Da die Robotersteuerung keinerlei zeitabhängige Ereignisse bewältigen muss und somit die gesamte Steuerung mittels Sensorik realisiert werden kann, gelten besondere Definitionen für die Simulationszeit und somit auch für die Variablen *tstart* und *tend*.
Die Ablaufsteuerung bei parallel DEVS wird mittels einer Ereignisliste realisiert und basiert ausschließlich auf der logischen Zeit. Diese orientieren sich bei der vorliegenden Implementierung an dem Sendezyklus der *Interface*-Komponente. Ein Zeitschritt von $\Delta t = 1$ bedeutet das einmalige Senden von Motorsteuerbefehlen an den Controller des Roboters.

Zusammenfassend lässt sich feststellen, dass die Robotersteuerung mittels DEVS möglich ist. Allerdings gibt es für die benutzte Experimentierplattform zwei limitierende Faktoren:

1. Die hohe Rechenzeit der Steuerung erlaubt keinen harten Echtzeitbetrieb auf einem gewöhnlichen PC.
2. Die Übertragungszeit bei der Kommunikation mittels RS232 ist im Echtzeitbetrieb nicht vernachlässigbar klein.

Aus diesen Gründen müssen alle Steuervorgänge mit harter Echtzeitanforderung auf einen Controller verlagert werden. Weiterhin kann der PDEVS-Simulatorprototyp im Hinblick auf die Laufzeit verbessert werden. Die Verwendung eines leistungsfähigeren Rechners und einer anderen Schnittstelle sollten die limitierenden Faktoren soweit abschwächen können, dass Echtzeitfähigkeit auf einem gewöhnlichen PC realisiert werden kann.

6 Erweiterte Steuerungskonzepte mit DEVS

Es existieren zahlreiche Erweiterungen[1] zu DEVS. Eine bedeutende Erweiterung stellen im Rahmen dieser Arbeit die *Real Time DEVS* (RT-DEVS) dar. Sie unterstützen die Ausführung eines DEVS-Modells in einer Echtzeitumgebung und erlauben somit die Interaktion zwischen einem Softwaresystem und der unmittelbaren Umgebung. RT-DEVS-Modelle können ohne den Modellcode zu transformieren oder zu modifizieren, mittels eines geeigneten *real-time-Simulators*, in Echtzeit ausgeführt werden. In diesem Abschnitt werden die grundlegenden Spezifikationen der RT-DEVS und deren Auswirkungen auf den zuvor dargestellten Simulationsmodellbasierten Steuerungsansatz erläutert.

6.1 Der RT-DEVS Formalismus

RT-DEVS stellen eine Erweiterung der klassischen DEVS Theorie (classic DEVS) dar. Dabei wird die Spezifikation der classic atomic DEVS wesentlich erweitert. Die Spezifikation der DEVN bleibt nahezu unverändert. Es entfällt lediglich die *Select*-Anweisung.
Ein atomic RT-DEVS Modell hat nach [ZPK00] folgende Struktur:

RTDEVS = {X, S, Y, δ_{ext}, δ_{int}, λ, *ti*, ψ, A}

X, S, Y, δ_{int}, λ analog classic atomic DEVS

A Menge der auszuführenden Aktivitäten einschließlich der Randbedingungen

ti: s $\rightarrow$ $R^{+}_{0,\infty}$ x $R^{+}_{0,\infty}$ *interval time advance function*

ψ: s $\rightarrow$ A *activity mapping function*

δ_{ext}: Q x X$\rightarrow$S, *external transition function*

Q={(s,e)|s $\in$ S, 0$\leq$e$\leq$*ti(s)*$|_{max}$}, *total state RT-DEVS*

Jedem Zustand s$\in$ S ist eine Aktivität ψ(s) zugeordnet. Jede Aktivität A stellt eine ausführbare Funktion dar und benötigt die aktuelle Simulationszeit. Weiterhin gilt, dass eine Aktivität weder Ereignisse empfangen, Ereignisse versenden oder gar Zustände verändern darf. Eine Zustandsänderung s$\rightarrow$s' eines RT-DEVS wird erst dann wirksam, wenn die zugehörige Aktivität vollständig durchgeführt worden ist.

[1] Dynamic Structure DEVS, Symbolic DEVS, Real Time DEVS, Fuzzy DEVS, ...

Das Zeitintervall *ti*(s) = [*comp_time* - ε, *comp_time* + ε] spezifiziert die Zeitspanne, innerhalb derer eine Aktivität A vollständig durchgeführt werden sollte. Dabei ist ε die mögliche Abweichung von der gewöhnlich zu erwartenden Ausführungszeit *comp_time*. Weiterhin wird angenommen, dass sämtliche Aktivitäten eine nichtdeterministische Ausführungszeit besitzen. Das dynamische Verhalten eines atomic RT-DEVS ist in Abbildung 38 dargestellt.

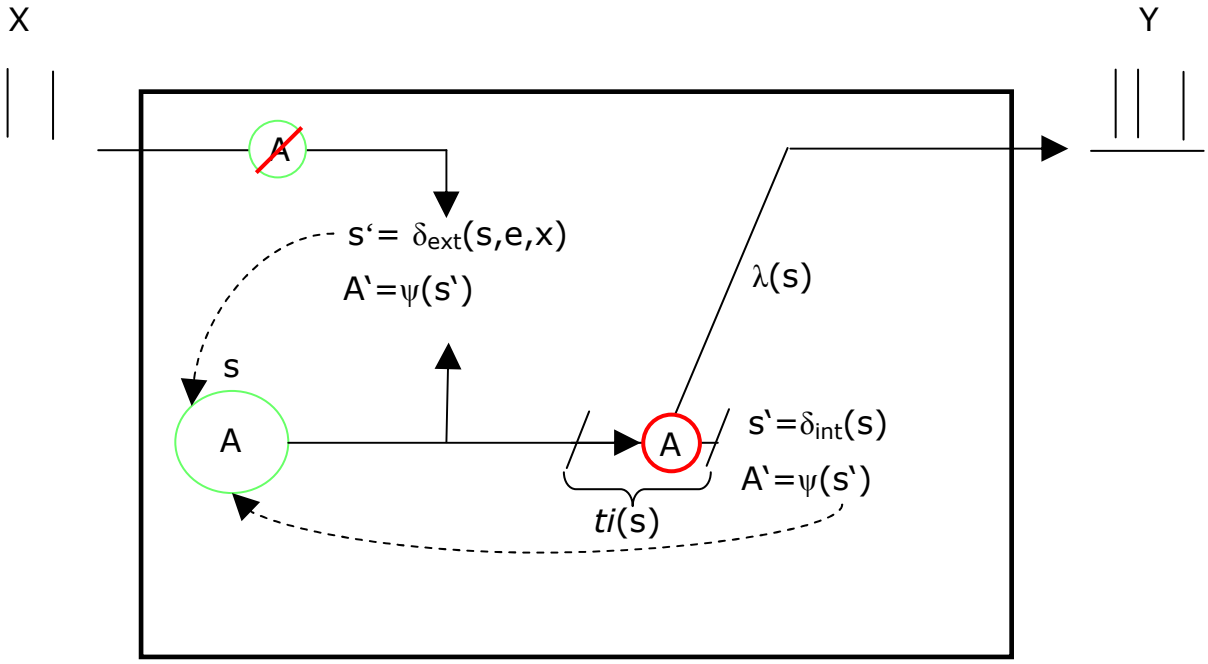


Abb. 38: Dynamisches Verhalten eines atomic RT-DEVS

Das Grundprinzip der Simulation ähnelt der klassischen DEVS Theorie. So wird auch hier jedem atomic DEVS ein Simulator und jedem coupled DEVS ein Koordinator zugeordnet. Die Simulation folgt allerdings nicht der *logischen Zeit*, sondern der Realzeit. Weiterhin gibt es im Simulator lediglich drei Nachrichtenarten:

1. die Zustandsnachricht (*, t) und
2. die Eingangsnachricht (x, t) und
3. die Ausgangsnachricht (y, t).

Zum Verständnis der RT-DEVS Spezifikation ist im Folgenden der Pseudocode für die Zustandsnachricht angegeben.

```
when receive (*, t)
  if ti_N|_min ≤ t ≤ ti_N|_max then
    y = λ(s)
    send y-message (y, t) to parent coordinator
    s = δ_int(s)
    tl= t
    ti_N = [tl + ti(s)|_min, tl + ti(s)|_max]
    a =ψ(s)
  else
    mistakes at *-message
```

Wenn der Simulator eine Eingangsnachricht (x, t) vom übergeordneten Koordinator erhält, wird die aktuelle Aktivität A des atomic RT-DEVS abgebrochen und eine neue Aktivität auf der Grundlage des neuen Zustands $s' = \delta_{ext}(s,e,x)$ bestimmt. Wenn ein atomic RT-DEVS seine Aktivität A beendet hat und kein externes Ereignis vorliegt, dann sendet der übergeordnete Koordinator eine Zustandsnachricht an den Simulator des betrachteten RT-DEVS.

Somit müssen die Koordinatoren zwei Kontrollaufgaben (control threads) parallel bewältigen. Der *monitoring thread* überwacht alle atomic RT-DEVS innerhalb eines coupled DEVS. Außerdem verwaltet der Koordinator je einen *activity thread* für jedes zugeordnete atomic RT-DEVS, in welchem die Aktivität A des zugeordneten RT-DEVS ausführt wird. Wenn beide control threads simultan agieren müssen, wird dieser Konflikt durch das unterlegte Echtzeitbetriebssystem aufgelöst. An dieser Stelle wird auch deutlich, dass RT-DEVS Modelle eigene Simulator- und Koordinatoralgorithmen besitzen, welche sich deutlich von den bisher behandelten Algorithmen unterscheiden.

6.2 Rapid Control Prototyping mit DEVS

Unter dem Begriff Rapid Control Prototyping [AB07] versteht man die effiziente Entwicklung einer Automatisierungsfunktion, beginnend bei der Simulation bis hin zum praktischen Einsatz, mittels einer durchgängigen Werkzeugkette. Zuerst wird innerhalb der Planungsphase ein Simulationsmodell des Realprozesses modelliert. Dieses Modell nennt man Prozessmodell. Auf Grundlage des Prozessmodells kann ein Control-Modell entwickelt werden, welches die Steuerungslogik für den Realprozess beinhaltet. Danach kann die durch Simulation entwickelte Steuerungslogik in „Hardware“ umgesetzt oder auf einem Mikrocontroller programmiert werden. In diesem Zusammenhang unterscheidet man zwei grundsätzliche Arten der Steuerungsentwicklung – *Hardware in the Loop* und *Software in the Loop*. Bei Hardware in the Loop (HiL) steuert ein realer Controller ein simuliertes Prozessmodell und bei Software in the Loop (SiL) steuert das simulierte Control-Modell den realen Prozess. Allgemein verfolgt RCP nach [AB07] analog zum Simulationsmodellbasierten Steuerungsansatz [MPP06] das Ziel, Simulationsmodelle umfassend von der frühen Planungsphase über die Automatisierungsphase bis hin zum Betrieb eines Systems einzusetzen. Einerseits soll dadurch die Systemsicherheit erhöht und andererseits Entwicklungskosten eingespart werden. Letzteres lässt sich erreichen, wenn Reimplementierungen der Simulationsmodelle im Entwicklungsprozess möglichst vermieden werden, das heißt sie sollten sukzessive erweiterbar sein. Dies verdeutlicht Abbildung 39.

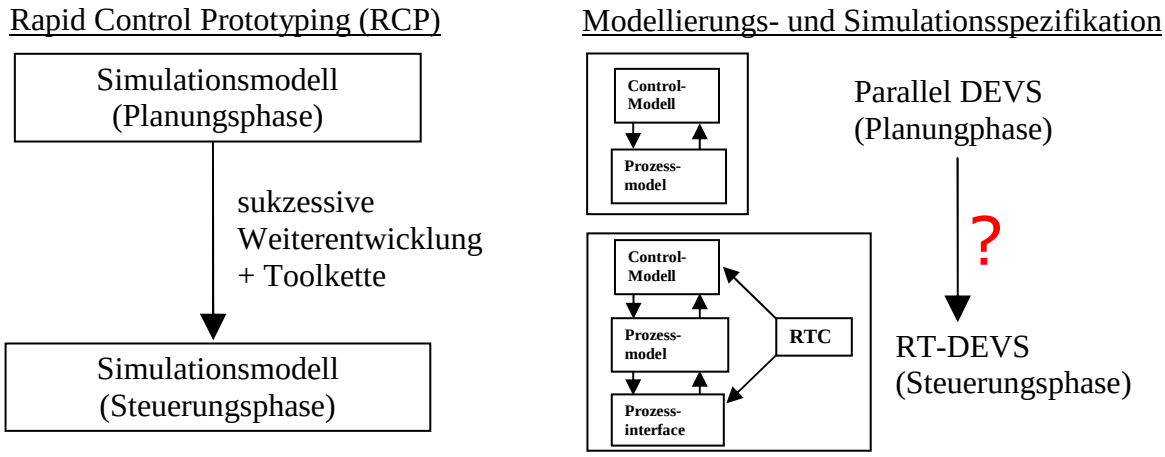


Abb. 39: RCP-Ansatz und DEVS-Spezifikationen

Planungsmodelle können auf Grundlage von Parallel DEVS am Entwicklungsrechner entwickelt und simuliert werden. Steuerungen müssen in der Lage sein Realprozesse in Echtzeit zu führen und könnten nach SiL, auf dem Entwicklungsrechner, auf Grundlage von RT-DEVS simuliert werden. Eine direkte Überführung von Parallel DEVS Modellen in

RT-DEVS Modelle ist dabei ausgeschlossen, da die Formalismen zu unterschiedlich sind. Damit ist die Werkzeugkette unterbrochen und RCP kann nicht realisiert werden! Die Integration des Simulationsmodellbasierten Steuerungsansatzes und RT-DEVS zeigt hingegen eine Möglichkeit auf, parallel DEVS Modelle durch geringfügige Änderungen der realzeitabhängigen Modellkomponenten „echtzeitfähig“ zu machen.

6.3 Integration von PDEVS und RT-DEVS zu RCP-DEVS

In diesem Abschnitt sollen Parallel DEVS und RT-DEVS miteinander kombiniert werden, so dass parallel DEVS Modelle in einer Echtzeitumgebung ausgeführt werden können und sich somit realzeitabhängige Steuerungsaufgaben besser realisieren lassen.
Dazu wird die Spezifikation von parallel atomic DEVS um die *time interval function* erweitert. Allerdings sollen die parallelen DEVS Simulatoren und Koordinatoren weiterhin für die Simulation genutzt werden. Daher erfolgt die Ablaufsteuerung der Simulation auf Basis der logischen Zeit. Auf Grundlage dieser Zeit werden alle vorliegenden internen Ereignisse, mittels der *time advance function*, in den Ereignislisten eingeplant und die Reihenfolge der Nachrichten zwischen den Koordinatoren und Simulatoren abgestimmt. Daneben existiert die Realzeit. Diese stellt die Grundlage für den Echtzeitbetrieb dar und wird neben der *time interval function* insbesondere für interne Ereignisse genutzt. Die Generierung von Ausgangsereignissen und das Eintreten eines internen Ereignisses finden entsprechend der RT-DEVS Spezifikation nur dann statt, wenn die Realzeit innerhalb des Zeitintervalls *ti* liegt. Diese erweiterte Struktur eines parallel DEVS, im Folgenden RCP-DEVS genannt, ist nachfolgend definiert. Das zugehörige dynamische Verhalten zeigt Abbildung 40.

RCP-DEVS = $\{X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}\ \lambda, ti, ta, A\}$

$S, Y, \delta_{ext}, \delta_{int}, \delta_{con}, ta$		analog parallel atomic DEVS mit Ports
ti, A		analog atomic RT-DEVS
$X \in \{X_{model}, X_{clock}\}$		Menge aller Eingangswerte
	$X_{model}=\{(p,v)\|p \in IPorts, v \in X_p\}$	Menge aller Eingangswerte des Simulationsmodells
	$X_{clock}=\{(\text{“clock“},v)\|v \in X_{clock}\}$	spezieller Eingangsport der Realzeit
λ	$\lambda: S \rightarrow Y$, *output function*	Ausgabefunktion
	$\lambda: S \rightarrow A$, *activity mapping function*	innerhalb *output function*

Jedes atomic RCP-DEVS besitzt einen inneren Zustand s und einen diesem Zustand zugeordnete Aktivität A. Die Zeitspanne bis zum nächsten internen Ereignis lässt sich mittels der time advance function *ta*(s) berechnen. Sobald diese Zeitspanne abgelaufen ist, wird die Ausgangsfunktion λ(s) ausgeführt und auf Grundlage des inneren Zustands s die aktuelle Aktivität abgebrochen, eine neue Aktivität initiiert und Ausgangsereignisse auf dem Ausgang Y erzeugt. Eine Aktivität kann analog Abbildung 37, mittels eines Controllers, einen beliebigen Aktor des realen Prozesses steuern. Auf die gleiche Art und Weise können Sensordaten innerhalb der Ausgangsfunktion ermittelt werden. Danach wird die interne Zustandsüberführungsfunktion δ_{int}(s) ausgelöst, welche auf Grundlage des inneren Zustands s den Folgezustand s' ermittelt.

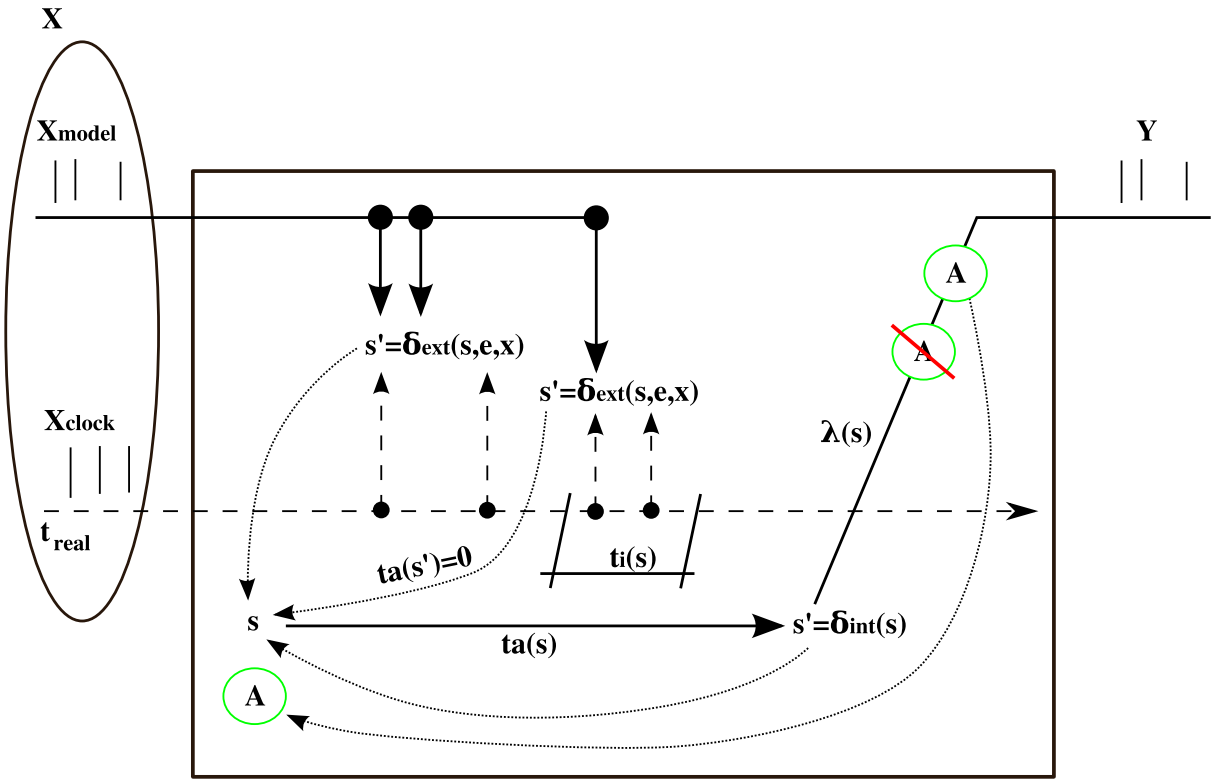


Abb. 40: Dynamisches Verhalten eines atomic RCP-DEVS

Sobald externe Ereignisse x auf dem Eingang X vorliegen, muss die externe Zustandsüberführungsfunktion δ_{ext}(s,e,x) ausgelöst werden. Als Eingangsereignisse x können gewöhnliche Ausgangsereignisse anderer atomic DEVS Komponenten in x_{model} oder das Ausgangsereignis der speziellen Modellkomponente RTC (Real Time Clock) in x_{clock} vorliegen. Am Eingang X_{clock} liegt somit immer die Realzeit an. Solange externe Ereignisse x außerhalb des Zeitintervalls *ti*(s) auftreten, bestimmt die externe Zustandsüberführungsfunktion auf Grundlage des inneren Zustands s, der verstrichenen Zeit e seit dem letzten Ereignis und den externen Ereignissen x den neuen inneren Zustand s'.

Daraufhin müssen auch *ta*(s') und *ti*(s') neu berechnet werden. Sobald externe Ereignisse x innerhalb des Zeitintervalls *ti*(s) auftreten und $x_{clock} \in x$ ist, bestimmt die externe Zustandsüberführungsfunktion auch auf Grundlage des inneren Zustands s, der verstrichenen Zeit e seit dem letzten Ereignis und den externen Ereignissen x den neuen inneren Zustand s'. Der Folgezustand s' muss dabei zu *ta*(s')=0 führen und hat somit unmittelbar ein internes Ereignis zur Folge.

Zentrales Element von RCP-DEVS ist die Integration der Realzeit als Eingangsereignis. Damit lassen sich realzeitabhängige Aufgabenstellungen, systemtheoretisch korrekt, mit einem atomic RCP-DEVS umsetzen, da jeder Trajektorie von Eingangsereignissen eindeutig eine Trajektorie von Ausgangsereignissen zugordnet werden kann. Auf dieser Grundlage werden im Folgenden ereignisorientierte Echtzeitsteuerungssysteme entwickelt. In diesem Zusammenhang bedeutet Echtzeit nicht, dass ein Steuerungssystem verzugsfrei auf eine Folge von Eingangsereignissen mit einer zugeordneten Folge von Ausgangsereignissen antworten muss. Stattdessen ist gefordert, dass eine Reaktion auf ein Ereignis innerhalb einer definierten Zeit eintritt. Dieses deterministische Reaktionsverhalten ist mittels der *time intervall function ti* für jede einzelne realzeitabhängige Modellkomponente realisiert und überprüfbar. Weiterhin sind die Anforderungen an Rechtzeitigkeit und Gleichzeitigkeit nach [AB07] zu berücksichtigen. Rechtzeitigkeit bedeutet, dass Eingabedaten rechtzeitig verfügbar sind und die daraus ermittelten Ausgabedaten rechtzeitig berechnet werden. Die Grenzen der Rechtzeitigkeit werden auf Grundlage des technischen Prozesses definiert. Allgemein lassen sich die so ermittelten Zeitbedingungen in Absolutbedingungen und Relativzeitbedingungen einteilen, welche beim vorliegenden Ansatz innerhalb der externen Zustandsüberführungsfunktion δ_{ext} geprüft werden. Eine realisierbare Relativzeitbedingung könnte sein, dass ein realer Prozess in einen sicheren Zustand fährt, wenn eine eingeplante Prozesszustandsänderung mindestens 10 s überfällig ist. So können einem realzeitabhängigen Control-Modell, nach Abbildung 41, Prozesszustände x_{model} und die Realzeit x_{clock} als Eingangsereignisse während der Simulation übermittelt werden. In der externen Zustandsüberführungsfunktion δ_{ext} des Control-Modells wird die Realzeit t_{real} mit dem Zeitintervall *ti* verglichen und eine eventuell auftretende Relativzeitbedingung erkannt. Daraufhin kann vom Controlmodell ein internes Ereignis durch $ta = 0$ eingeplant und mittels der Ausgangsfunktion in den technischen Prozess eingegriffen werden. Somit lassen sich z.B. fehlertolerante Systeme realisieren. Gleichzeitigkeit bedeutet nach [AB07], dass ein sequentiell verarbeitetes Steuerprogramm rechtzeitig auf gleichzeitig ablaufende Umweltvorgänge reagieren muss. Dafür müssen bestimmte Randbedingungen spezifiziert

werden. Eine dieser Randbedingungen ist eine hinreichend große Abtastfrequenz für wesentliche Zustände des technischen Prozesses. Diese Anforderung lässt sich auch mittels RCP-DEVS realisieren und führt zur Entwicklung eines Prozessinterface nach Abbildung 41. So können einem realzeitabhängigen Prozessinterface Steuerbefehle x_{model} und die Realzeit x_{clock} als Eingangsereignisse während der Simulation übermittelt werden. In der externen Zustandsüberführungsfunktion δ_{ext} des Prozessinterface werden die Steuerbefehle innerhalb der Zustandsfelder des atomic RCP-DEVS abgespeichert. Außerdem wird die Realzeit t_{real} mit dem Zeitintervall *ti* verglichen. Das interne Ereignis wird unmittelbar eingeplant, wenn sich die Realzeit t_{real} innerhalb des Zeitintervalls *ti* befindet. Dann werden in der Ausgangsfunktion λ die Steuerbefehle an einen Controller gesendet, Sensordaten von diesem empfangen und Ausgangsereignisse berechnet. Die Bestimmungsgleichung für *ti* basiert auf der erforderlichen Abtastfrequenz für den technischen Prozess.

6.4 RCP-DEVS und Simulationsmodellbasiertem Steuerungsansatz

In diesem Abschnitt soll die Echtzeitsimulation mit RCP-DEVS anhand eines abstrakten Beispiels verdeutlicht werden. Abbildung 41 zeigt die Struktur des Steuerungssystems, die logische Zeit t, die Realzeit t_{real} und die Ereignisliste des zugeordneten Koordinators zum logischen Steuerungszeitpunkt t = 3.

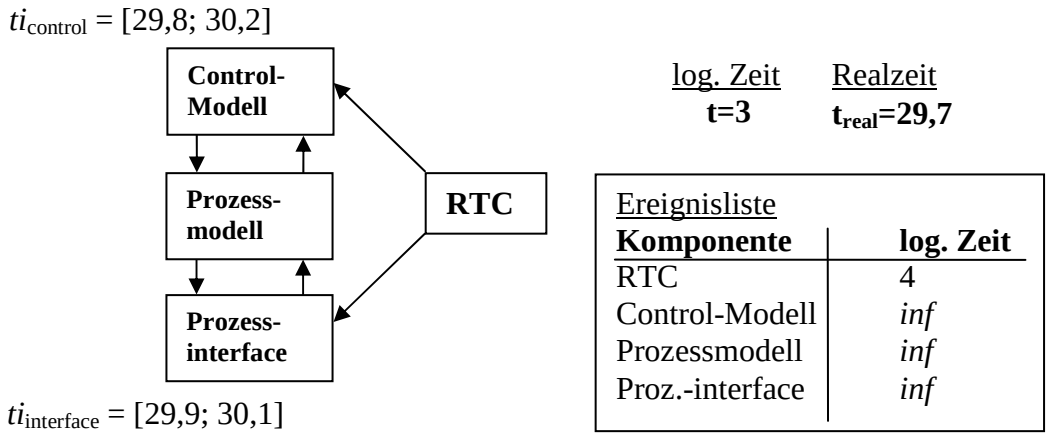


Abb. 41: Echtzeitsteuerung zum logischen Zeitpunkt t=3

Die Modellkomponenten Control-Modell und Prozessinterface stellen die realzeitabhängigen Komponenten des Systems dar. Beim Control-Modell liegt alle 30 ± 0,2 s ein internes Ereignis vor und führt zu einem Eingangsereignis des Prozessmodells. Beim atomic DEVS Prozessinterface liegt alle 1 ± 0,1 s ein internes Ereignis vor. Entsprechend der RCP-DEVS

Spezifikation werden vom Prozessinterface, innerhalb der zugehörigen λ-function, die Steuerbefehle an den Controller des Realprozesses und die aktuellen Sensordaten mittels einer Ausgangsnachricht an das Prozessmodell gesendet.

Das atomic DEVS RTC (Real Time Clock) plant mit der konstanten logischen Schrittweite $ta_{RTC} = 1$ ein internes Ereignis ein. Innerhalb der λ-function ermittelt RTC die Realzeit t_{real}, welche anschließend mittels einer Ausgangsnachricht an die realzeitabhängigen Komponenten versendet wird.

Das nächste interne Ereignis liegt zum logischen Zeitpunkt t = 4 am atomic DEVS RTC vor. Folglich wird wie in Abbildung 42 dargestellt die logische Zeit auf den Zeitpunkt t = 4 vorgeschaltet. Daraufhin versendet RTC die Realzeit t_{real} = 29,7 mittels einer Ausgangsnachricht, an die Komponenten Control-Modell und Prozessinterface. Beide Komponenten überprüfen in der external transition function δ_{ext}, ob sich die von RTC übermittelte Realzeit innerhalb des Intervalls *ti* befindet. Da dies nicht der Fall ist, wird bei beiden Komponenten *ta* = inf gesetzt und damit kein internes Ereignis eingeplant. Da RTC sein nächstes internes Ereignis stets mit konstanter Schrittweite einplant, folgt hier der logische Zeitfortschritt *ta* = 1.

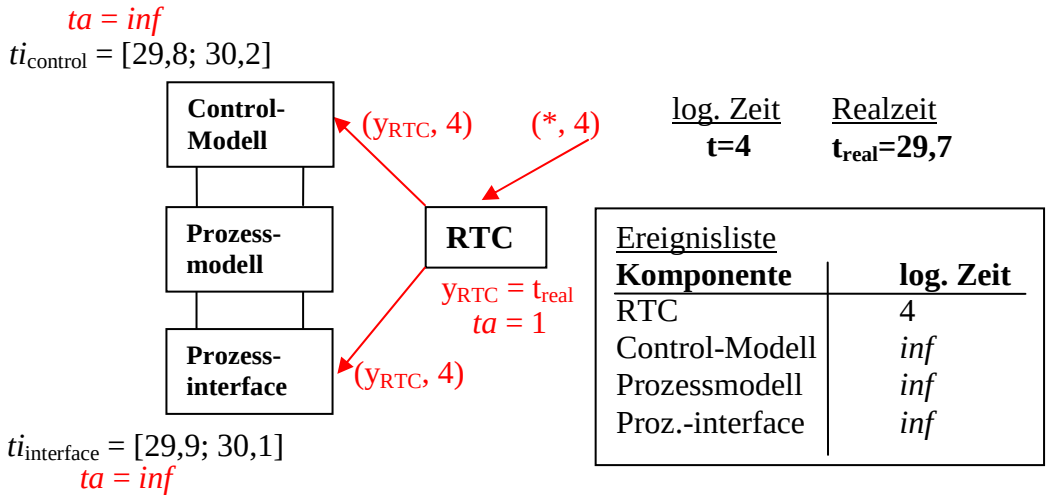


Abb. 42: Echtzeitsteuerung zum logischen Zeitpunkt t=4

Das nächste interne Ereignis liegt zum logischen Zeitpunkt t = 5 beim atomic DEVS RTC vor. Deshalb wird gemäß Abbildung 43 die logische Zeit auf den Zeitpunkt t = 5 fortgeschaltet und es werden je eine Ausgangsnachricht mit der Realzeit t_{real} = 30 von RCP an die realzeitabhängigen Komponenten versendet. Beide Komponenten überprüfen daraufhin in der external transition function δ_{ext}, ob sich die Realzeit innerhalb des Intervalls *ti* befindet. Diesmal liegt die Realzeit innerhalb der Grenzen von *ti*. Daher wird durch *ta* = 0 sofort ein internes Ereignis für die Komponenten Prozessinterface und Control-Modell eingeplant.

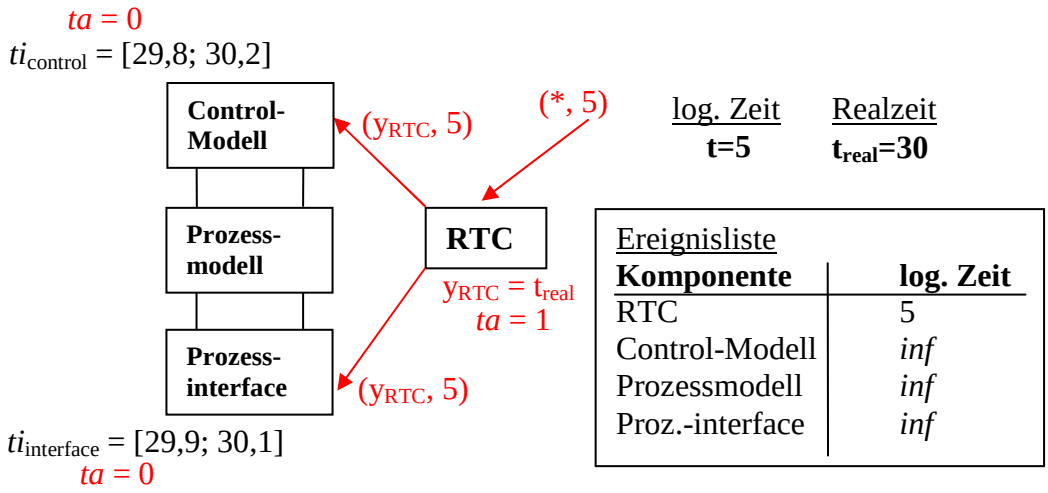


Abb. 43: Echtzeitsteuerung zum logischen Zeitpunkt t=5, Schritt 1

Die nächsten internen Ereignisse liegen für das Prozessinterface und das Control-Modell zum Zeitpunkt = 5 vor. Demzufolge müssen beide Komponenten zum logischen Zeitpunkt t = 5 ihr internes Ereignis ausführen, während das atomic DEVS RTC mit *ta* = 1 ein nächstes internes Ereignis zum logischen Zeitpunkt t = 6 einplant, wobei unabhängig von der logischen Zeit die Realzeit fortschreitet. Abbildung 44 zeigt die damit einhergehenden Abarbeitungsschritte. Das Control-Modell und das Prozessinterface erhalten zu t = 5 eine *-message. Dadurch sendet das atomic RCP-DEVS Prozessinterface im korrekten Realzeitintervall Steuerungsbefehle an den realen Prozess und empfängt in der λ-function (siehe Abschnitt 5.2.2) Sensordaten vom realen Prozess und sendet diese mittels einer Ausgangsnachricht (y_I,5) an das übergeordnete Prozessmodell. Ebenso verarbeitet das Control-Modell zu t = 5 ein internes Ereignis und sendet eine Ausgangsnachricht (y_C,5) an das Prozessmodell.

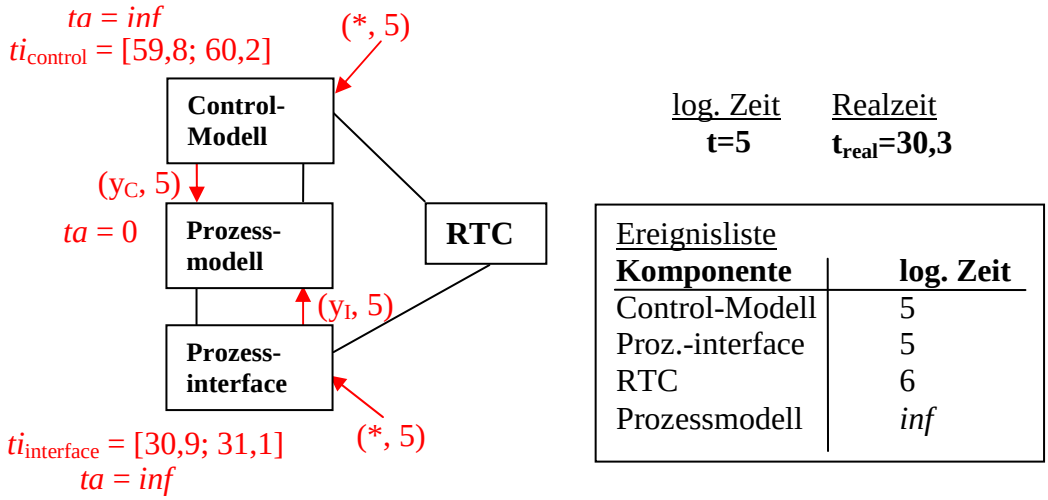


Abb. 44: Echtzeitsteuerung zum logischen Zeitpunkt t=5, Schritt 2

Außerdem berechnen beide realzeitabhängigen Komponenten das neue Zeitintervall *ti* für ihr nächstes internes Ereignis. Das Prozessmodell bestimmt auf Basis der übermittelten Steuerbefehle und den Sensordaten, innerhalb der external transition function δ_{ext}, die neuen Prozesszustände. Im vorliegenden Fall führt das zu neuen Steuerbefehlen, welche umgehend an das Prozessinterface gesendet werden müssen. Daher wird für das Prozessmodell *ta* = 0 berechnet und somit ein internes Ereignis zum logischen Zeitpunkt t = 5 eingeplant, während gemäß Abbildung 45 die Realzeit auf t_{real} = 30,6 fortschreitet. Das interne Ereignis hat eine Ausgangsnachricht (y_P,t) an das Prozessinterface zur Folge, in welcher die neuen Steuerbefehle übermittelt werden. Das Prozessinterface speichert in der external transition function δ_{ext} die übermittelten Steuerbefehle als Zustände ab und wird diese innerhalb des Zeitintervalls *ti* an den realen Prozess senden. Das Prozessmodell plant aufgrund von *ta* = inf kein neues internes Ereignis ein. An dieser Stelle ist der Anfangszustand wieder erreicht und

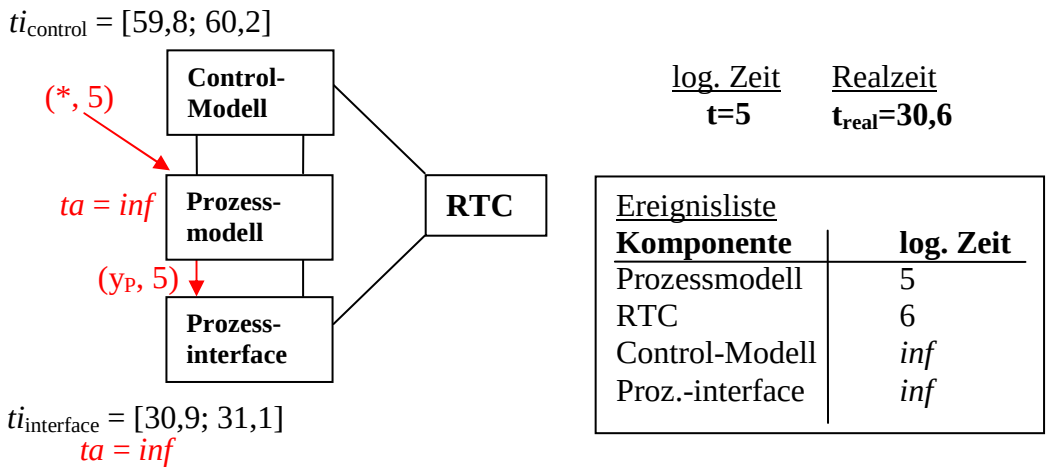


Abb. 45: Echtzeitsteuerung zum logischen Zeitpunkt t=5, Schritt 3

die einzige Komponente mit einem eingeplanten internen Ereigniss ist RTC. Diesen Zustand mit fortgeschrittener Realzeit zeigt Abbildung 46.

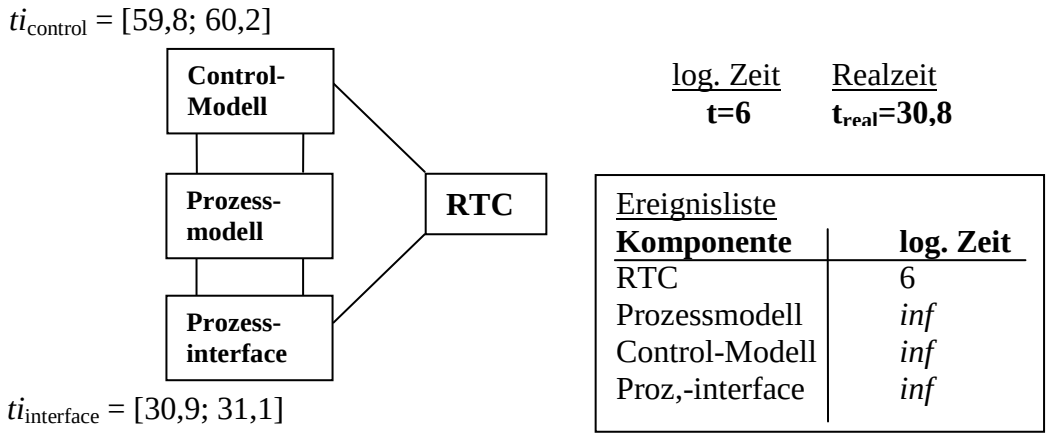


Abb. 46: Echtzeitsteuerung zum logischen Zeitpunkt t=6

6.5 Beispiel Echtzeitsteuerung eines Roboters

Im Folgenden wird die Robotersteuerung aus Abschnitt 5.2 um realzeitabhängiges Verhalten erweitert. Jetzt soll der Roboter bestimmte Raumpunkte mittels asynchroner PTP-Bewegung anfahren (Gleichzeitigkeit) und darüber hinaus definierte Zeitspannen in Ruhe verharren können (Relativzeitbedingung). Das erweiterte Simulationsmodellbasierte Steuerungssystem zeigt Abbildung 47.

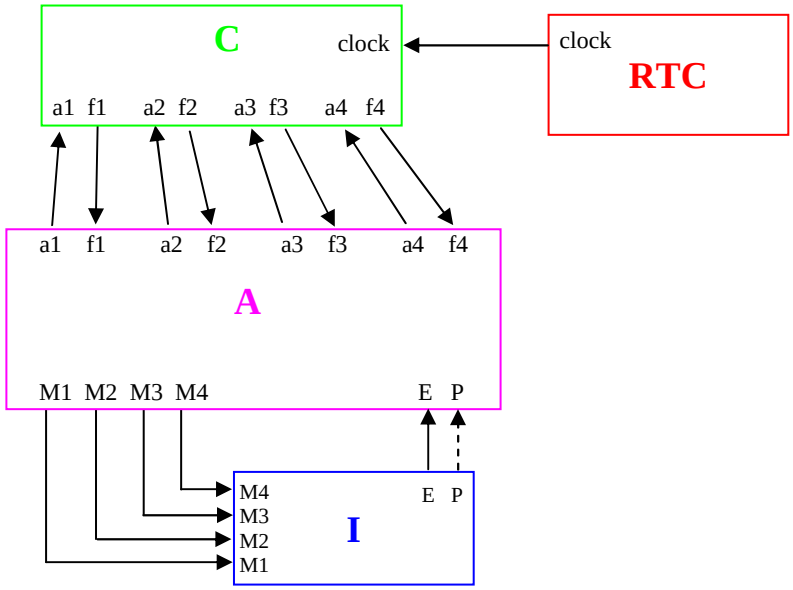


Abb. 47: erweitertes Simulationsmodellbasierte Steuerungssystem

Die Bewegungsdaten werden wieder in der Matrix $traj_{(N, 5)}$ gespeichert, welche dem atomic DEVS Control *C* als Parameter übergeben wird.

traj = [$\mathbf{A_{i,j}}$]$_{(N, 5)}$

$\mathbf{A_{i\,1}}$, ..., $\mathbf{A_{i\,4}}$: Zielposition i der Achsen 1 bis 4

$\mathbf{A_{i,5}}$: Dauer der Ruhephase nach Erreichen der Zielposition i

Die neue Spezifikation des atomic DEVS Control C ist nachfolgend aufgeführt und Änderungen sind rot markiert.

$DEVS_C = \{X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta\}$

IPorts = {“a1”, “a2”, “a3” , “a4”, "clock"}, where $X_{a1, ...}, X_{a4} \in N$ and $X_{clock} \in R_0^+$

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values

OPorts = {“f1”, “f2” , “f3” , “f4”}, where $Y_{f1}, ..., Y_{f4} \in N$

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = \{matrix_{(1,4)} | components \in N\} \times N \times [R_{0,\infty}^+, R_{0,\infty}^+] \times R_0^+ \times R_{0,\infty}^+$

initial state s = ([akt: Pos: der 4 Akt:], index; *ti*, clock, σ) = ([1, 1, 1, 1], 1, [*inf, inf*], ∅, 1)

δ_{ext}([a_inc1, a_inc2, a_inc3, a_inc4], index, *ti*, clock, σ, X, e)

```
if (“a1”, x1) ∈ X
  a_inc1 = x1
elseif (“a2”, x2) ∈ X
  a_inc2 = x2
elseif (“a3”, x3) ∈ X
  a_inc3 = x3
elseif (“a4”, x4) ∈ X
  a_inc4 = x4
elseif (“clock”,x5) ∈ X
  clock = x5
elseif ti(1) ≤ x5 ≤ ti(2)
    σ = 0
    ti = [inf, inf]
//arrival at final point?
elseif [a_inc_1,...,a_inc_4] == traj( index, (1, ...,4) )
  if this not the last point in traj
    σ = 0
      if traj( index, 5) > 0
        ti = [clock+traj(index,5)-ε, clock+traj(index,5)+ε]
        σ = inf
    ++index
  else
    σ = inf
```

=([a_inc1, a_inc2, a_inc3, a_inc4], index, ti, clock, σ)

δ_{int}(a_inc_pos, index, ti, clock, σ) = (a_inc_pos, index, ti, clock, *inf*)

δ_{con}(s, x, e) = δ_{ext}(s, x, e)

λ([a_inc1, a_inc2, a_inc3, a_inc4], index, ti, clock, σ)

```
    f1 = traj(index, 1)
    f2 = traj(index, 2)
    f3 = traj(index, 3)
    f4 = traj(index, 4)
```

=((“f1“, *f1*), (“f2“, *f2*), (“f3“, *f3*), (“f4“, *f4*))

ta(a_inc_pos, index, ti, clock, σ) = σ

Außerdem muss das atomic DEVS RTC spezifiziert werden.

$DEVS_{RTC} = \{X, Y, S, \delta_{int}, \lambda, ta\}$

$X = \varnothing$

OPorts = {"clock"}, where $Y_{clock} \in R_0^+$

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values

$S = R_0^+$

initial state $s = (\sigma) = (0)$

$\delta_{int}(\sigma) = (1)$

$\lambda(\sigma) =$ ("clock", *current real time)*

$ta(\sigma) = \sigma$

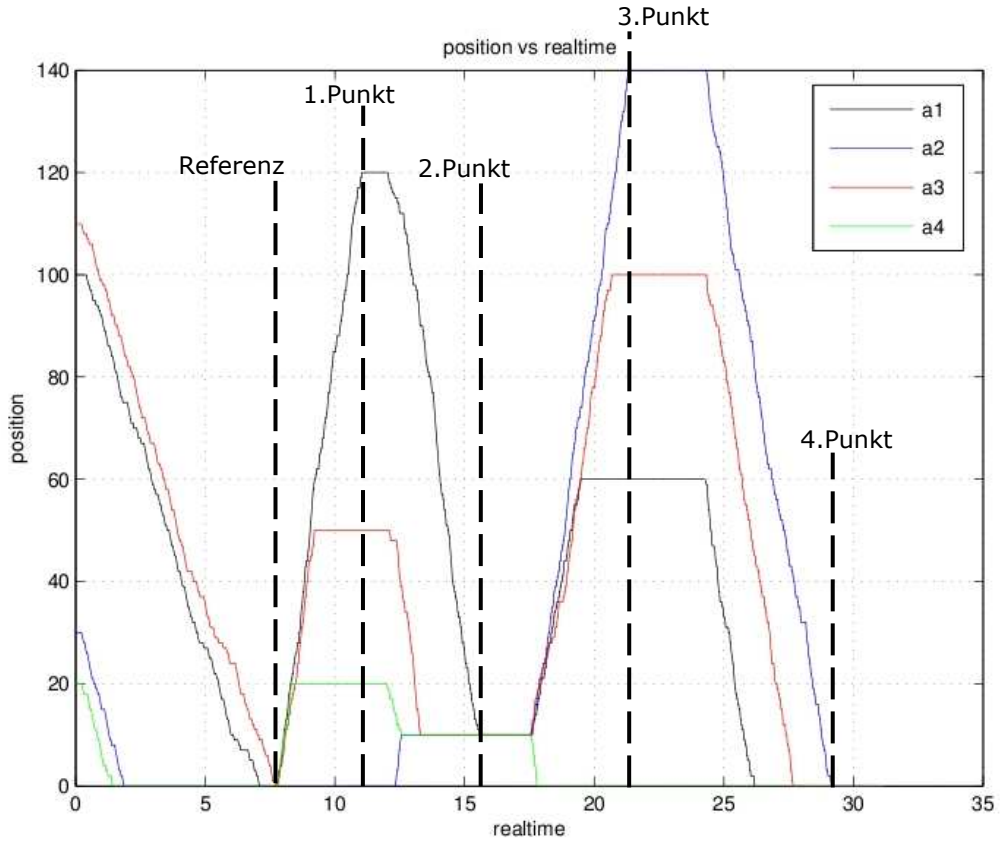
Für die Steuerung muss das im Anhang A3 hinterlegte Initialisierungsskript *init_c.m* des Echtzeitsteuerungssystems ausgeführt werden. Dieses Skript erzeugt und parametriert alle notwendigen Komponenten und speichert die vollständig initialisierte Steuerung auf der Variablen *model_robot.*
Anschließend muss mittels der Funktion *devs_root_coordinator* die Steuerung gestartet und der Zustand der Steuerung am Ende auf einer Variablen rückgespeichert werden. Die folgenden Codeausschnitte zeigen den Aufruf, die Durchführung und die Auswertung der Echtzeitrobotersteuerung in Matlab.

```
init_c;
tstart = 0;
tend = 2500;
erg_model =devs_root_coordinator(model_robot,tstart,tend);
```

Der Roboter fährt nacheinander alle Punkte gemäß der Matrix *traj* an und bleibt in der letzten Position stehen.
Außerdem befindet sich auf dem beiliegenden Datenträger ein Planungsmodell der Echtzeitrobotersteuerung. Dieses Planungsmodell simuliert den vollständigen Echtzeitsteuerprozess auf dem Rechner. Dabei ist im Interface *I* das Ein-/Ausgangsverhalten des Roboters modelliert.

Für die Simulation muss das im Anhang A4 hinterlegte Skript *off-line-simulation.m* ausgeführt werden. Dieses Skript erzeugt und parametriert alle notwendigen Komponenten, führt die Simulation in Echtzeit aus und gibt die Trajektorie der Bewegungsachsen über der Realzeit graphisch aus. Das Ergebnis einer Planungssimulation zeigt Abbildung 48.

Zustände des Realmodells



Startposition der vier Achsen

$$[100 \quad 30 \quad 110 \quad 20]$$

Bewegungsbahn *traj*

$$\left[\begin{array}{cccc:c} 0 & 0 & 0 & 0 & 0 \\ 120 & 0 & 50 & 20 & 1 \\ 10 & 10 & 10 & 10 & 2 \\ 60 & 140 & 100 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Abb. 48: Ergebnisse einer Planungssimulation

7 Zusammenfassung und Ausblick

Seit der Einführung des klassischen DEVS Formalismus durch Zeigler 1976, wird diese Theorie und deren Erweiterungen für vielfältige Aufgaben innerhalb der ereignisorientierten Simulation eingesetzt. Die heutigen Einsatzgebiete gehen dabei über die Systemsimulation komplexer Modelle und Zellulärer Automaten, der Parameter- und Strukturoptimierung bis hin zu echtzeitfähigen Steuerungen. Das Rapid Control Prototyping (RCP) Konzept von Abel beschreibt die effiziente Entwicklung einer Automatisierungsfunktion, beginnend bei der Simulation bis hin zum praktischen Einsatz, mittels einer durchgängigen Werkzeugkette. Außerdem ist von ihm der Begriff *Software in the Loop* (SiL) eingeführt worden. Dieser beschreibt die Steuerung eines technischen Prozesses mittels eines simulierten Control-Modells. Ein ähnliches Konzept stellt der Simulationsmodellbasierte Steuerungsansatz der FG CEA dar. Auch hier wird ein technischer Prozess mittels eines simulierten Control-Modells gesteuert. Im Unterschied zu Abel wird neben diesem Control-Modell, stets das um eine Prozessschnittstelle erweiterte Prozessmodell in Form eines Zustandsbeobachters simuliert. Beide Steuerungskonzepte können bislang in den SCEs[1] Matlab/Simulink, Matlab/Stateflow oder Scilab/Scicos umgesetzt werden.

In dieser Arbeit ist eine Robotersteuerung unter Nutzung des DEVS-Formalismus entwickelt worden. Im ersten Teil dieser Arbeit werden der klassische DEVS-Formalismus und der Parallel-DEVS-Formalismus beschrieben. Außerdem werden zwei unterschiedliche Simulationsalgorithmen für Parallel DEVS Modelle nach Chow und Zeigler analysiert. Auf Grundlage dieser Analyse ist ein modifizierter Simulationsalgorithmus entwickelt und in Form eines Matlab-DEVS-Simulatorprototyps implementiert worden, welcher den Anforderungen von Simulation und Steuerung gerecht wird. Im zweiten Teil dieser Arbeit wird die Leistungsfähigkeit des Simulatorprototyps durch die Modellierung und Simulation einer laborativen Montagestraße überprüft. Außerdem wird auf Grundlage des Simulationsmodellbasierten Steuerungsansatzes der FG CEA eine Robotersteuerung modelliert und zur Steuerung eines *Fischertechnik*-Roboters eingesetzt. Anschließend wird im dritten Teil die Steuerungsaufgabe des Roboters um zusätzliche Echtzeitanforderungen erweitert. Es besteht die Problemstellung, die Realzeit in die ereignisorientierte Simulation zu integrieren. Prinzipiell sind Echtzeitsteuerungen mit dem RT-DEVS-Formalismus möglich. In

1 Scientific and Technical Computing and Visualization Environments

dieser Arbeit wird aufgezeigt, dass eine direkte Überführung von Parallel DEVS Modellen in RT-DEVS Modelle ausgeschlossen ist. Deshalb wurde die Integration vom Simulationsmodellbasiertem Steuerungsansatz und RT-DEVS entwickelt. Auf Grundlage dieser Integration kann ein Planungsmodell einer Echtzeitsteuerung modelliert und in Echtzeit simuliert werden. Anschließend wird das Planungsmodell zu einem Steuerungsmodell erweitert und zur praktischen Echtzeitsteuerung des *Fischertechnik*-Roboters genutzt.

Die Master-Thesis zeigt, dass Robotersteuerungen auf Grundlage des DEVS-Formalismus entwickelt werden können. Die praktische Steuerungsentwicklung lässt sich dabei in das RCP-Konzept integrieren. Damit die Leistungsfähigkeit von DEVS innerhalb von RCP umfassend beurteilt werden kann, sind weitere und umfangreichere Anwendungsbeispiele zu modellieren und zu simulieren. Dafür sind insbesondere die sequentiellen Simulationsalgorithmen zu nutzen, welche eine möglichst schnelle Verarbeitung des Steuerungssystems erlauben.

Literaturverzeichnis

[AB07] Abel, D. ; Bollig, A. : *Rapid Control Prototyping, Methoden und Anwendungen*. Berlin, Heidelberg: Springer Verlag, 2007

[Dea03] Deatcu, Chr. : *Development of an Object-Orientated DEVS-Simulator with Matlab***.** Wismar, Hochschule Wismar, Forschungsgruppe CEA, Diplomarbeit, Januar 2003

[Mat04] MathWorks, The: *Matlab® 7 Help*. Natick, MA USA: The Mathworks, Inc. , 2004

[Mel07] Meller, F. : *Untersuchungen zur Transformation von Ereignisgraphenmodellen (EGM) in DEVS-Modelle*. Wismar, Hochschule Wismar, Forschungsgruppe CEA, Master-Thesis, Mai 2007

[PS92] Pichler, F. ; Schwärzler, H. : *CAST Methods in Modelling: Computer Aided Systems Theory for the Design of Intelligent Machines*. Berlin, Heidelberg, New York: Springer-Verlag, 1992

[MPP06] Maletzki, M. ; Pawletta, T. ; Pawletta, S. ; Dünow, P. ; Lampe, B. : *Simulationsmodellbasiertes Rapid Prototyping von komplexen Robotersteuerungen*. atp-Automatisierungstechnische Praxis, Oldenbourg Verlag, München, 50(2008)8, 54-60 (submitted 12/2006)

[SS08] Stegemann, F. ; Schwatinski T. : *Steuerung einer integrierten Fertigungszelle und eines Roboters mit Matlab/Simulink/Stateflow*. Wismar, Hochschule Wismar, Forschungsgruppe CEA, Projektarbeit, März 2008

[Cho96] Chow, A. C.-H. : *Parallel DEVS: A parallel, hierachical, modular modelling formalism and its distributed simulator*. Transactions of The Society for Computer Simulation International: *Volume 13 No 2*, June 1996, 55-67

[UH06] Uhrmacher, A. M. ; Himmelspach, J.: *Sequential processing of PDEVS models*. Proceedings of the 3rd EMSS, Barcelona, Spain, October 2006, 239-244

[ZPK00] Zeigler, B. P. ; Prähofer, H. ; Kim, T. G. : Theory of Modeling ans Simulaltion Second Edition: *Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2. Aufl. San Diego, San Francisco, New York, Boston, London, Sydney, Tokyo: Academic Press 2000

Anhang

A Initialisierungsskripte

A1 Initialisierungsskript Montagestraße

```
%init.m               Initialization Montagestrasse
clear all,clc;
global model    %used for modelling

%time elapsed for all components at model
e=0;

%coupled DEVS c1 and c3

    %atomic DEVS m1 and m2
        x_m=struct('p1',[],'p2',[]);
        y_m=struct('p1',[]);
        s_m=struct('phase','passiv','sigma',inf,'q1',0,'q2',0);
        model.am_m1=am_proc_block(x_m,y_m,s_m,e,4);
        model.am_m2=am_proc_block(x_m,y_m,s_m,e,3);

    %coupled DEVS c1 and c3
        x_cm_c1=struct('p1',[],'p2',[],'p3',[]);
        y_cm_c1=y_m;
        Da_cm_c1={'am_m1','am_m2'};
        Dc_cm_c1=[];
        Zid_cm_c1={'parent','p1','am_m1','p1';...
            'parent','p2','am_m1','p2';...
            'parent','p3','am_m2','p2';...
            'am_m1','p1','am_m2','p1';...
            'am_m2','p1','parent','p1'};
        model.cm_c1=coupled(x_cm_c1,y_cm_c1,Da_cm_c1,Dc_cm_c1,Zid_cm_c1);
        model.cm_c3=model.cm_c1;

%coupled DEVS c2

    %coupled DEVS c2
        x_cm_c2=struct('p1',[],'p2',[],'p3',[],'p4',[]);
        y_cm_c2=struct('p1',[],'p2',[]);
        Da_cm_c2={'am_m1','am_m2'};
        Dc_cm_c2=[];
        Zid_cm_c2={'parent','p1','am_m1','p1';...
            'parent','p2','am_m1','p2';...
            'parent','p3','am_m2','p1';...
            'parent','p4','am_m2','p2';...
            'am_m1','p1','parent','p1';...
            'am_m2','p1','parent','p2'};
        model.cm_c2=coupled(x_cm_c2,y_cm_c2,Da_cm_c2,Dc_cm_c2,Zid_cm_c2);

    %change service_time of m1 and m2 with set method
        model.cm_c2=set(model.cm_c2,'service_time',3,{'am_m1'});
        model.cm_c2=set(model.cm_c2,'service_time',5,{'am_m2'});

%coupled DEVS cm_g

    %atomic DEVS am_g1 ... am_g7 with arrival_time
        model.am_g1=am_generator(0,3);
        model.am_g2=am_generator(0,4);
        model.am_g3=am_generator(0,5);
        model.am_g4=am_generator(0,4);
        model.am_g5=am_generator(0,3);
        model.am_g6=am_generator(0,2);
         model.am_g7=am_generator(0,2)
```

```
    %coupled DEVS cm_g
        x_generator=[];
        y_generator=struct('p1',[],'p2',[],'p3',[],'p4',[],'p5',[],'p6',...
                           [],'p7',[]);
        Da_generator={'am_g1','am_g2','am_g3','am_g4','am_g5','am_g6',...
                      'am_g7'};
        Dc_generator=[];
        Zid_generator={'am_g1','p1','parent','p1';...
            'am_g2','p1','parent','p2';...
            'am_g3','p1','parent','p3';...
            'am_g4','p1','parent','p4';...
            'am_g5','p1','parent','p5';...
            'am_g7','p1','parent','p7';...
            'am_g6','p1','parent','p6'};
        model.cm_g=coupled(x_generator,y_generator,Da_generator,...
                           Dc_generator,Zid_generator);

%atomic DEVS am_t

        model.am_t=am_transducer();

%model

        Dc_model={'cm_c1','cm_c2','cm_c3','cm_g'};
        Da_model={'am_t'};
        Zid_model={'cm_c1','p1','cm_c3','p1';...
            'cm_c2','p1','cm_c3','p2';...
            'cm_c2','p2','cm_c3','p3';...
            'cm_g','p1','cm_c1','p1';...
            'cm_g','p2','cm_c1','p2';...
            'cm_g','p3','cm_c1','p3';...
            'cm_g','p4','cm_c2','p1';...
            'cm_g','p5','cm_c2','p2';...
            'cm_g','p6','cm_c2','p3';...
            'cm_g','p7','cm_c2','p4';...
            'cm_c3','p1','am_t','p1'};
        model=coupled([],[],Da_model,Dc_model,Zid_model);
```

A2 Initialisierungsskript Steuerungssystem

```
%init_c.m             Initialization Steuerungssystem
clear all,clc;
global model
addpath([cd,'\RS232']);

%atomic DEVS a1, a2, a3 and a4 - with upper_limit and el_number
model.a1=ml_actor(250,1);
model.a2=ml_actor(145,2);
model.a3=ml_actor(240,3);
model.a4=ml_actor(40,4);

%atomic DEVS i
model.i=ll_cinterface('com1');

%atomic DEVS c
traj=[0,0,0,0;...
    120,0,50,20;...
    10,10,10,10;...
    60,140,100,0;...
    0,0,0,0];
model.c=hl_control(traj);
```

```
%coupled DEVS a
a_x=struct('E',[],'P',[],'f1',[],'f2',[],'f3',[],'f4',[]);
a_y=struct('M1',[],'M2',[],'M3',[],'M4',[],'a1',[],'a2',[],'a3',[],'a4',[]);
a_Da={'a1','a2','a3','a4'};
a_Dc=[];
a_Zid={'parent','E','a1','E';...
    'parent','P','a1','P';...
    'parent','f1','a1','f_inc_pos';...
    'a1','M','parent','M1';...
    'a1','a_inc_pos','parent','a1';...
    'parent','E','a2','E';...
    'parent','P','a2','P';...
    'parent','f2','a2','f_inc_pos';...
    'a2','M','parent','M2';...
    'a2','a_inc_pos','parent','a2';...
    'parent','E','a3','E';...
    'parent','P','a3','P';...
    'parent','f3','a3','f_inc_pos';...
    'a3','M','parent','M3';...
    'a3','a_inc_pos','parent','a3';...
    'parent','E','a4','E';...
    'parent','P','a4','P';...
    'parent','f4','a4','f_inc_pos';...
    'a4','M','parent','M4';...
    'a4','a_inc_pos','parent','a4'};
model.a=coupled(a_x,a_y,a_Da,a_Dc,a_Zid);

%coupled DEVS model_robot
model_x=[];
model_y=[];
model_Da={'i','c'};
model_Dc={'a'};
model_Zid={'i','E','a','E';...
    'i','P','a','P';...
    'a','M1','i','M1';...
    'a','M2','i','M2';...
    'a','M3','i','M3';...
    'a','M4','i','M4';...
    'a','a1','c','a1';...
    'a','a2','c','a2';...
    'a','a3','c','a3';...
    'a','a4','c','a4';...
    'c','f1','a','f1';...
    'c','f2','a','f2';...
    'c','f3','a','f3';...
    'c','f4','a','f4'};
model_robot=coupled(model_x,model_y,model_Da,model_Dc,model_Zid);
```

A3 Initialisierungsskript Echtzeitsteuerungssystem

```
%init_c.m             Initialization Echtzeitsteuerungssystem
clear all,clc;
global model

addpath([cd,'\RS232']);

%atomic DEVS a1, a2, a3 and a4 - with upper_limit and el_number
model.a1=ml_actor(250,1);
model.a2=ml_actor(145,2);
model.a3=ml_actor(240,3);
model.a4=ml_actor(40,4);

%atomic DEVS i
model.i=ll_cinterface('com1');
```

```
%atomic DEVS c
traj=[0,0,0,0,0;...
    140,0,50,20,1;...
    10,10,10,10,2;...
    60,140,100,0,3;...
    0,0,0,0,0];
model.c=hl_control(traj);

%atomic DEVS RTC
model.RTC=RTC();

%coupled DEVS a
a_x=struct('E',[],'P',[],'f1',[],'f2',[],'f3',[],'f4',[]);
a_y=struct('M1',[],'M2',[],'M3',[],'M4',[],'a1',[],'a2',[],'a3',[],'a4',[]);
a_Da={'a1','a2','a3','a4'};
a_Dc=[];
a_Zid={'parent','E','a1','E';...
    'parent','P','a1','P';...
    'parent','f1','a1','f_inc_pos';...
    'a1','M','parent','M1';...
    'a1','a_inc_pos','parent','a1';...
    'parent','E','a2','E';...
    'parent','P','a2','P';...
    'parent','f2','a2','f_inc_pos';...
    'a2','M','parent','M2';...
    'a2','a_inc_pos','parent','a2';...
    'parent','E','a3','E';...
    'parent','P','a3','P';...
    'parent','f3','a3','f_inc_pos';...
    'a3','M','parent','M3';...
    'a3','a_inc_pos','parent','a3';...
    'parent','E','a4','E';...
    'parent','P','a4','P';...
    'parent','f4','a4','f_inc_pos';...
    'a4','M','parent','M4';...
    'a4','a_inc_pos','parent','a4'};
model.a=coupled(a_x,a_y,a_Da,a_Dc,a_Zid);

%coupled DEVS model_robot
model_x=[];
model_y=[];
model_Da={'RTC','i','c'};
model_Dc={'a'};
model_Zid={'i','E','a','E';...
    'i','P','a','P';...
    'a','M1','i','M1';...
    'a','M2','i','M2';...
    'a','M3','i','M3';...
    'a','M4','i','M4';...
    'a','a1','c','a1';...
    'a','a2','c','a2';...
    'a','a3','c','a3';...
    'a','a4','c','a4';...
    'c','f1','a','f1';...
    'c','f2','a','f2';...
    'c','f3','a','f3';...
    'c','f4','a','f4';...
    'RTC','clock','c','clock'};
model_robot=coupled(model_x,model_y,model_Da,model_Dc,model_Zid);
```

A4 Initialisierungsskript Planungsmodell Echtzeitsteuerung

```
%off_line_simulation.m   Initialization Planungsmodell_Echtzeitsteuerungssystem
clear all,close all,clear classes,clc;
global model

%atomic DEVS i  - Prozessmodell
P_process=[1,0,0,1];
E_process=[1,1,1,1];
position=[100,30,110,20];
model.i=ll_interface(P_process,E_process,position);

%atomic DEVS a1, a2, a3 and a4 - with upper_limit and el_number
model.a1=ml_actor(200,1);
model.a2=ml_actor(145,2);
model.a3=ml_actor(210,3);
model.a4=ml_actor(40,4);

%atomic DEVS c
traj=[0,0,0,0,0;...
    120,0,50,20,1;...
    10,10,10,10,2;...
    60,140,100,0,3;...
    0,0,0,0,0];
model.c=hl_control(traj);

%atomic DEVS RTC
model.RTC=RTC();

%coupled DEVS a
a_x=struct('E',[],'P',[],'f1',[],'f2',[],'f3',[],'f4',[]);
a_y=struct('M1',[],'M2',[],'M3',[],'M4',[],'a1',[],'a2',[],'a3',[],'a4',[]);
a_Da={'a1','a2','a3','a4'};
a_Dc=[];
a_Zid={'parent','E','a1','E';...
    'parent','P','a1','P';...
    'parent','f1','a1','f_inc_pos';...
    'a1','M','parent','M1';...
    'a1','a_inc_pos','parent','a1';...
    'parent','E','a2','E';...
    'parent','P','a2','P';...
    'parent','f2','a2','f_inc_pos';...
    'a2','M','parent','M2';...
    'a2','a_inc_pos','parent','a2';...
    'parent','E','a3','E';...
    'parent','P','a3','P';...
    'parent','f3','a3','f_inc_pos';...
    'a3','M','parent','M3';...
    'a3','a_inc_pos','parent','a3';...
    'parent','E','a4','E';...
    'parent','P','a4','P';...
    'parent','f4','a4','f_inc_pos';...
    'a4','M','parent','M4';...
    'a4','a_inc_pos','parent','a4'};
model.a=coupled(a_x,a_y,a_Da,a_Dc,a_Zid);

%coupled DEVS model_robot
model_x=[];
model_y=[];
model_Da={'RTC','i','c'};
model_Dc={'a'};
model_Zid={'i','E','a','E';...
    'i','P','a','P';...
    'a','M1','i','M1';...
    'a','M2','i','M2';...
    'a','M3','i','M3';...
    'a','M4','i','M4';...
    'a','a1','c','a1';...
    'a','a2','c','a2';...
```

```
    'a','a3','c','a3';...
    'a','a4','c','a4';...
    'c','f1','a','f1';...
    'c','f2','a','f2';...
    'c','f3','a','f3';...
    'c','f4','a','f4';...
    'RTC','clock','c','clock'};
model_robot=coupled(model_x,model_y,model_Da,model_Dc,model_Zid);

%report
clc;
disp(['Planungsmodell wird in Echtzeit gerechnet - Bitte haben Sie Geduld']);
erg_model=devs_root_coordinator(model_robot,0,1700);
Daten=get(get(erg_model,'comps','i'),'Daten');
figure(1);
plot(Daten(:,5),Daten(:,1),'k');
hold on;
plot(Daten(:,5),Daten(:,2),'b');
plot(Daten(:,5),Daten(:,3),'r');
plot(Daten(:,5),Daten(:,4),'g');
legend('a1','a2','a3','a4');
ylabel('position');
xlabel('realtime');
title('position vs realtime');
grid on;
```

B Methoden der Klasse atomic

B1 atomic()

```
function a=atomic(x,y,s,e)
%
%constructor method for class atomic
%a=atomic(x,y,s,e)
%variables:
%   a       //type: object
%   x       //type: structure    //set of input values
%   y       //type: structure    //set of output values
%   s       //type: structure    //set of states
%   e       //type: float        //time elapsed since last
%                                  transition (only for initialization)

if nargin==4
        a=struct('e',e,'tl',[],'tn',inf,'x',x,'y',y,'s',s);
        a=class(a,'atomic');
else
    disp('mistake at constructor method for class atomic');
end
```

B2 atomic_simulator()

```
function m=atomic_simulator(m_parent,m,flag,gt)
%
%simulator for basic parallel DEVS
%m=atomic_simulator(m_parent,m,flag,gt)
%variables:
%   m           //type: object
%   m_parent    //type: object   //parent object
%   flag        //type: string   //i_msg|s_msg|y_msg|x_msg
%   gt          //type: float    //current simulation time
%variables used in simulator
%   tn      //type: float            //next transition time
%   tl      //type: float            //last transition time

%when receive i-message at time tg
if strcmp(flag,'i_msg')
    tl=gt-m_parent.e;
    tn=tl+tafun(m);
    m=set(m,'tl',tl);
    m=set(m,'tn',tn);

%when receive *-message
elseif strcmp(flag,'S_MSG')
    if m_parent.tn~=gt
        disp('error: bad synchronization at atomic_simulator');
    end
    m=deltaintfun(m);
    tn=gt+tafun(m);
    m=set(m,'tl',gt);
    m=set(m,'tn',tn);

%when receive y-message
elseif strcmp(flag,'y_msg')
    if m_parent.tn~=gt
        disp('error: bad synchronization at atomic_simulator');
    end
    m=lambdafun(m);

%when receive x-message
elseif strcmp(flag,'x_msg')
    if m_parent.tl>gt | gt>m_parent.tn
        disp('error: bad synchronization at atomic_simulator');
    end
    m=deltaextfun(m,gt);
    tn=gt+tafun(m);
    m=set(m,'tl',gt);
    m=set(m,'tn',tn);
end
```

B3 get()

```
function value=get(a,prop)
%
%get method for class atomic
%value=get(a,prop)
%variables:
%   value       //type: depends on argument  //value from argument
%   a           //type: object
%   prop        //type: string
%variables used at simulator
%   e       //type: float           //elapsed time since last
%                                     transition (only for initialization)
%   tl      //type: float           //last transition time
%   tn      //type: float           //next transition time
%   x       //type: structure       //set of input values
%   y       //type: structure       //set of output values
%   s       //type: structure       //set of states

if nargin == 2
    switch prop
        case 'e'
            value=a.e;
        case 'tl'
            value=a.tl;
        case 'tn'
            value=a.tn;
        case 'x'
            value=a.x;
        case 'y'
            value=a.y;
        case 's'
            value=a.s;
        otherwise
            disp('mistake at get method for class atomic');
    end
end
```

B4 set()

```
function a=set(a,prop,val)
%
%set method for class atomic
%a=set(a,prop,val)
%variables:
%   a       //type: object
%   prop    //type: string           //property which should be changed
%   val     //type: depends on props //new value for property
%variables used at simulator
%   e       //type: float            //elapsed time since last
%                                      transition (only for initialization)
%   tl      //type: float            //last transition time
%   tn      //type: float            //next transition time
%   x       //type: structure        //set of input values
%   y       //type: structure        //set of output values
%   s       //type: structure        //set of states

if nargin==3
    switch prop
        case 'e'
            a.e=val;
        case 'tl'
            a.tl=val;
        case 'tn'
            a.tn=val;
        case 'x'
            a.x=val;
        case 'y'
            a.y=val;
        case 's'
            a.s=val;
        otherwise
            disp('mistake at set method for class atomic');
    end
end
```

C Methoden der Klasse coupled

C1 coupled()

```
function c=coupled(x,y,Da,Dc,Zid)
%
%constructor method for class coupled
%c=coupled(x,y,Da,Dc,Zid)
%variables:
%   c       //type: object
%   x       //type: structure                //set of input values
%   y       //type: structure                //set of output values
%   Da      //type: cell-array with strings  //atomic objects at c
%   Dc      //type: cell-array with strings  //coupled objects at c
%   Zid     //type: cell-array with strings  //"i-to-d-matrix"
%        (if Da, Dc or Zid are empty: please type [] )
%variables used at simulator
%   model   //type: structure                //basis for all DEVS-objects
%   comps   //type: structure                //field for saving DEVS-objects
%   i       //type: int                      //only counter-variable

global model

if nargin==5
    i=0;
    comps=struct();
    if ~isempty(Da)
        for i=1:length(Da)
            comps=setfield(comps,char(Da(i)),getfield(model,char(Da(i))));
        end
    end
    if ~isempty(Dc)
        for i=1:length(Dc)
            comps=setfield(comps,char(Dc(i)),getfield(model,char(Dc(i))));
        end
    end
    c=struct('tl',[],'tn',inf,'x',x,'y',y,'Da',{Da},'Dc',{Dc},'Zid',...
             {Zid},'eventlist',[],'comps',comps);
    c=class(c,'coupled');
else
    disp('mistake at constructor method for class coupled');
end
```

C2 coupled_simulator()

```
function parent=coupled_simulator(parent,flag,gt)
%
%simulator for parallel DEVS coupled model
%m=coupled_simulator(parent,flag,gt)
%variables:
%   parent      //type: object   //parent object
%   flag        //type: string   //i_msg|s_msg|y_msg|x_msg
%   gt          //type: float    //current simulation time
%variables used in simulator
%   i,j,k,l     //type: int              //index-counter for simulator
%   comps       //type: structure        //field for saving DEVS-objects
%   a           //type: object           //atomic object
%   c           //type: object           //coupled object
%   imminents   //type: matrix           //matrix for imminents
%   parenty     //type: matrix           //output-ports from parent
%   parentx     //type: structure        //old input-ports from parent
%   new_parentx //type: structure        //new input-ports from parent
%   x           //type: structure        //input ports from DEVS
%   y           //type: structure        //output ports from DEVS
%   names       //type: cell-array with strings  //fieldnames from ports
%   deltaconf   //type: inf              // true or false if element is conf

%######################when receive i-message##############################
if strcmp(flag,'i_msg')

%   WHAT HAPPENS?
%   - set times tl and tn for all objects
%   - set matrix eventlist
%                eventlist=[a,b,c;...]
%                   a - time für next transition
%                   b - {1,0} | 1 if atomic DEVS, 2 if coupled DEVS
%                   c - index for eventlist (position in Da or Dc)

    comps=parent.comps;

    %atomic-models
    if ~isempty(parent.Da)
        for i=1:length(parent.Da)
            a=getfield(comps,char(parent.Da(i)));
            a=atomic_simulator(get(a,'parent'),a,'i_msg',gt);
            comps=setfield(comps,char(parent.Da(i)),a);
            if i==1
                parent.tl=get(a,'tl');
                parent.tn=get(a,'tn');
                parent.eventlist(i,:)=[parent.tn,1,i];
            else
                if parent.tl<get(a,'tl')
                    parent.tl=get(a,'tl');
                end
                parent.eventlist(i,:)=[get(a,'tn'),1,i];
                if parent.tn>parent.eventlist(i,1)
                    parent.tn=parent.eventlist(i,1);
                end
            end
        end
    end

    %coupled models
    if ~isempty(parent.Dc)
      for j=1:length(parent.Dc)
        c=coupled_simulator(getfield(comps,char(parent.Dc(j))),'i_msg',gt);
        comps=setfield(comps,char(parent.Dc(j)),c);
        if j==1 && isempty(parent.eventlist)
             parent.tl=get(c,'tl');
             parent.tn=get(c,'tn');
             parent.eventlist(j,:)=[parent.tn,2,j];
             i=0;
```

```
        else
             if parent.tl<get(c,'tl')
                 parent.tl=get(c,'tl');
             end
             parent.eventlist(i+j,:)=[get(c,'tn'),2,j];
             if parent.tn>parent.eventlist(i+j,1)
                 parent.tn=parent.eventlist(i+j,1);
             end
        end
      end
    end
    parent.comps=comps;

%############################when receive s-message############################
elseif (strcmp(flag,'s_msg')|strcmp(flag,'S_MSG'))
%
%   WHAT HAPPENS?
%   - copy all elements from matrix eventlist which are imminent to
%     matrix imminents (same structure like eventlist)
%   - delete all copied rows (look at step before) in eventlist
%   - carry y-message (only for imminents) out -> set all relevant
%     output ports to input ports
%   - transmit relevant input ports from parent to all childs -> delete
%     all input ports from parent
%   - carry x-message out from relevant elements in matrix eventlist and
%     delete the input ports
%   - carry s-message or confluent-function out for relevant elements in
%     matrix imminents and delete the input ports
%   - set times tl an tn at parent and build a new matrix eventlist

    comps=parent.comps;
    if parent.tn~=gt
        disp('error: bad synchronization at coupled_simulator');
    end

 %set matrix imminents
 imminents=parent.eventlist(find(parent.eventlist(:,1)==gt),:);
 parent.eventlist=parent.eventlist(find(parent.eventlist(:,1)~=gt),:);

  if strcmp(flag,'s_msg') %only when *-msg from root-coordiantor
    %carry y-message out, set and delete output/input ports
    for i=1:size(imminents,1)
        %atomic models
        if imminents(i,2)==1
            a=getfield(comps,char(parent.Da(imminents(i,3))));
            y=get(atomic_simulator(get(a,'parent'),a,'y_msg',gt),'y');
            for l=1:size(parent.Zid,1)
                if strcmp(char(parent.Zid(l,1)),...
                          char(parent.Da(imminents(i,3))))
                    if ~strcmp(char(parent.Zid(l,3)),'parent')
                        x=get(getfield(comps,char(parent.Zid(l,3))),'x');
                        x=setfield(x,char(parent.Zid(l,4)),...
                                   getfield(y,char(parent.Zid(l,2))));
                        comps=setfield(comps,char(parent.Zid(l,3)),...
                                       set(getfield(comps,char(...
                                       parent.Zid(l,3))),'x',x));
                    end
                end
            end
        %coupled models
        else
            c=coupled_simulator(getfield(comps,char(parent.Dc(...
                                imminents(i,3)))),'y_msg',gt);
            y=get(c,'y');
            for l=1:size(parent.Zid,1)
                if strcmp(char(parent.Zid(l,1)),...
                          char(parent.Dc(imminents(i,3))))
                    if ~strcmp(char(parent.Zid(l,3)),'parent')
                        x=get(getfield(comps,char(parent.Zid(l,3))),'x');
                        x=setfield(x,char(parent.Zid(l,4)),...
```

```
                                getfield(y,char(parent.Zid(l,2))));
                        comps=setfield(comps,char(parent.Zid(l,3)),...
                                  set(getfield(comps,char(...
                                  parent.Zid(l,3))),'x',x));
                    end
                end
            end
            names=fieldnames(y);%delete outputs and save couplded DEVS
            for k=1:length(names)
                y=setfield(y,char(names(k)),[]);
            end
            comps=setfield(comps,char(parent.Dc(imminents(i,3))),...
                set(c,'y',y));
        end
    end
end

    %transmit relevant input ports from parent to its childs -> delete all
    %input ports from parent
    parentx=parent.x;
    new_parentx=parent.x;
    for i=1:size(parent.Zid,1)
        if strcmp(char(parent.Zid(i,1)),'parent')
            x=get(getfield(comps,char(parent.Zid(i,3))),'x');
            x=setfield(x,char(parent.Zid(i,4)),getfield(parentx,...
                      char(parent.Zid(i,2))));
            new_parentx=setfield(new_parentx,char(parent.Zid(i,2)),[]);
            comps=setfield(comps,char(parent.Zid(i,3)),set(...
                          getfield(comps,char(parent.Zid(i,3))),'x',x));
        end
    end
    parent.x=new_parentx;

    %carry x-message out from relevant elements in matrix eventlist and
    %delete the input ports
    for i=1:size(parent.eventlist,1)
      %atomic DEVS
      if parent.eventlist(i,2)==1
         x=get(getfield(comps,char(parent.Da(parent.eventlist(i,3)))),'x');
         if ~isempty(x)
             names=fieldnames(x);
             for l=1:length(names)
                if ~isempty(getfield(x,char(names(l))))
                  a=getfield(comps,char(parent.Da(parent.eventlist(i,3))));
                  a=atomic_simulator(get(a,'parent'),a,'x_msg',gt);
                  for k=l:length(names)
                       x=setfield(x,char(names(k)),[]);
                  end
                  comps=setfield(comps,char(parent.Da(...
                            parent.eventlist(i,3))),set(a,'x',x));
                  break;
                end
             end
         end
      %coupled DEVS
      else
         x=get(getfield(comps,char(parent.Dc(parent.eventlist(i,3)))),'x');
         if ~isempty(x)
             names=fieldnames(x);
             for l=1:length(names)
                if ~isempty(getfield(x,char(names(l))))
                  c=coupled_simulator(getfield(comps,char(parent.Dc(...
                        parent.eventlist(i,3)))),'x_msg',gt);
                  for k=l:length(names)
                       x=setfield(x,char(names(k)),[]);
                  end
                  comps=setfield(comps,char(parent.Dc(...
                            parent.eventlist(i,3))),set(c,'x',x));
                  break;
                end
```

```
         end
      end
   end
end

%carry s-message or confluent-function out for relevant elements in
%matrix imminents and delete the input ports
for i=1:size(imminents,1)
    %atomic DEVS
    if imminents(i,2)==1
        deltaconf=0;
        x=get(getfield(comps,char(parent.Da(imminents(i,3)))),'x');
        if ~isempty(x)
            names=fieldnames(x);
            for k=1:length(names)
                if ~isempty(getfield(x,char(names(k))))
                    deltaconf=1;
                    break;
                end
            end
        end
        switch deltaconf
            %carry s-message out
            case 0
               a=getfield(comps,char(parent.Da(imminents(i,3))));
               a=atomic_simulator(get(a,'parent'),a,'S_MSG',gt);
               comps=setfield(comps,char(parent.Da(imminents(i,3))),a);
            %carry confluent-function out
            case 1
               a=deltaconffun(getfield(comps,char(...
                              parent.Da(imminents(i,3)))),gt);
               for l=k:length(names)
                   x=setfield(x,char(names(l)),[]);
               end
               comps=setfield(comps,char(parent.Da(imminents(i,3))),...
                              set(a,'x',x));
        end
    %coupled DEVS
    else
        c=coupled_simulator(getfield(comps,char(parent.Dc(...
                            imminents(i,3)))),'S_MSG',gt);
        comps=setfield(comps,char(parent.Dc(imminents(i,3))),c);
    end
end

%set times tl and tn of parent and build a new matrix eventlist
parent.eventlist=[];

%atomic-models
if ~isempty(parent.Da)
    for i=1:length(parent.Da)
        a=getfield(comps,char(parent.Da(i)));
        if i==1
            parent.tl=get(a,'tl');
            parent.tn=get(a,'tn');
            parent.eventlist(i,:)=[parent.tn,1,i];
        else
            if parent.tl<get(a,'tl')
                parent.tl=get(a,'tl');
            end
            parent.eventlist(i,:)=[get(a,'tn'),1,i];
            if parent.tn>parent.eventlist(i,1)
                parent.tn=parent.eventlist(i,1);
            end
        end
    end
end

%coupled models
if ~isempty(parent.Dc)
```

```
        for j=1:length(parent.Dc)
            c=getfield(comps,char(parent.Dc(j)));
            if j==1 && isempty(parent.eventlist)
                parent.tl=get(c,'tl');
                parent.tn=get(c,'tn');
                parent.eventlist(j,:)=[parent.tn,2,j];
                i=0;
            else
                if parent.tl<get(c,'tl')
                    parent.tl=get(c,'tl');
                end
                parent.eventlist(i+j,:)=[get(c,'tn'),2,j];
                if parent.tn>parent.eventlist(i+j,1)
                    parent.tn=parent.eventlist(i+j,1);
                end
            end
        end
    end
    parent.comps=comps;

%############################when receive y-message#############################
elseif strcmp(flag,'y_msg')

%   WHAT HAPPENS?
%   - copy all elements from matrix eventlist which are imminent to
%     matrix imminents (same structure like eventlist)
%   - carry y-message out for all elements at imminents
%       -> set relevant input/output ports

 comps=parent.comps;
 %set matrix imminents
 imminents=parent.eventlist(find(parent.eventlist(:,1)==gt),:);

 %Lambdafunktion ausführen
 parenty=parent.y;
 for i=1:size(imminents,1)
        %atomic DEVS
        if imminents(i,2)==1
            a=getfield(comps,char(parent.Da(imminents(i,3))));
            y=get(atomic_simulator(get(a,'parent'),a,'y_msg',gt),'y');
            for l=1:size(parent.Zid,1)
                if strcmp(char(parent.Zid(l,1)),...
                          char(parent.Da(imminents(i,3))))
                    if strcmp(char(parent.Zid(l,3)),'parent')
                        parenty=setfield(parenty,char(parent.Zid(l,4)),...
                                         getfield(y,char(...
                                         parent.Zid(l,2))));
                    else
                        x=get(getfield(comps,char(parent.Zid(l,3))),'x');
                        x=setfield(x,char(parent.Zid(l,4)),...
                                   getfield(y,char(parent.Zid(l,2))));
                        comps=setfield(comps,char(parent.Zid(l,3)),...
                                       set(getfield(comps,char(...
                                       parent.Zid(l,3))),'x',x));
                    end
                end
            end
        %coupled DEVS
        else
            c=coupled_simulator(getfield(comps,char(parent.Dc(...
                                imminents(i,3)))),'y_msg',gt);
            y=get(c,'y');
            for l=1:size(parent.Zid,1)
                if strcmp(char(parent.Zid(l,1)),...
                          char(parent.Dc(imminents(i,3))))
                    if strcmp(char(parent.Zid(l,3)),'parent')
                        parenty=setfield(parenty,char(parent.Zid(l,4)),...
                                         getfield(y,char(...
                                         parent.Zid(l,2))));
```

```
                    else
                        x=get(getfield(comps,char(parent.Zid(l,3))),'x');
                        x=setfield(x,char(parent.Zid(l,4)),...
                                   getfield(y,char(parent.Zid(l,2))));
                        comps=setfield(comps,char(parent.Zid(l,3)),...
                                  set(getfield(comps,char(...
                                  parent.Zid(l,3))),'x',x));
                    end
                end
            end
            names=fieldnames(y);%delete outputs and save couplded DEVS
            for k=1:length(names)
                y=setfield(y,char(names(k)),[]);
            end
            comps=setfield(comps,char(parent.Dc(imminents(i,3))),...
                set(c,'y',y));
        end
 end
 parent.y=parenty;
 parent.comps=comps;

%############################when receive x-message#############################
elseif strcmp(flag,'x_msg')

%   WHAT HAPPENS?
%   - transmit relevant input ports from parent to all childs -> delete
%     all input ports from parent
%   - carry x-message out from relevant elements in matrix eventlist and
%     delete the input ports
%   - set times tl and tn of parent and build a new matrix eventlist

    comps=parent.comps;

    %transmit relevant input ports from parent to its childs -> delete all
    %input ports from parent
    parentx=parent.x;
    new_parentx=parent.x;
    for i=1:size(parent.Zid,1)
        if strcmp(char(parent.Zid(i,1)),'parent')
            x=get(getfield(comps,char(parent.Zid(i,3))),'x');
            x=setfield(x,char(parent.Zid(i,4)),getfield(parentx,...
                       char(parent.Zid(i,2))));
            new_parentx=setfield(new_parentx,char(parent.Zid(i,2)),[]);
            comps=setfield(comps,char(parent.Zid(i,3)),set(...
                          getfield(comps,char(parent.Zid(i,3))),'x',x));
        end
    end
    parent.x=new_parentx;

    %carry x-message out
    for i=1:size(parent.eventlist,1)  %externe Transition aus eventlist
      %atomic DEVS
      if parent.eventlist(i,2)==1
        x=get(getfield(comps,char(parent.Da(parent.eventlist(i,3)))),'x');
        if ~isempty(x)
            names=fieldnames(x);
            for l=1:length(names)
                if ~isempty(getfield(x,char(names(l))))
                  a=getfield(comps,char(parent.Da(parent.eventlist(i,3))));
                  a=atomic_simulator(get(a,'parent'),a,'x_msg',gt);
                  for k=l:length(names)
                      x=setfield(x,char(names(k)),[]);
                  end
                  comps=setfield(comps,char(parent.Da(...
                                parent.eventlist(i,3))),set(a,'x',x));
                  break;
                end
            end
        end
```

```
      %coupled DEVS
      else
        x=get(getfield(comps,char(parent.Dc(parent.eventlist(i,3)))),'x');
        if ~isempty(x)
            names=fieldnames(x);
            for l=1:length(names)
                if ~isempty(getfield(x,char(names(l))))
                  c=coupled_simulator(getfield(comps,char(parent.Dc(...
                                    parent.eventlist(i,3)))),'x_msg',gt);
                  for k=l:length(names)
                      x=setfield(x,char(names(k)),[]);
                  end
                  comps=setfield(comps,char(parent.Dc(...
                                parent.eventlist(i,3))),set(c,'x',x));
                  break;
                end
            end
        end
      end
    end

    %set times tl and tn of parent and build a new matrix eventlist
    parent.eventlist=[];

    %atomic models
    if ~isempty(parent.Da)
        for i=1:length(parent.Da)
            a=getfield(comps,char(parent.Da(i)));
            if i==1
                parent.tl=get(a,'tl');
                parent.tn=get(a,'tn');
                parent.eventlist(i,:)=[parent.tn,1,i];
            else
                if parent.tl<get(a,'tl')
                    parent.tl=get(a,'tl');
                end
                parent.eventlist(i,:)=[get(a,'tn'),1,i];
                if parent.tn>parent.eventlist(i,1)
                    parent.tn=parent.eventlist(i,1);
                end
            end
        end
    end

    %coupled models
    if ~isempty(parent.Dc)
        for j=1:length(parent.Dc)
            c=getfield(comps,char(parent.Dc(j)));
            if j==1 && isempty(parent.eventlist)
                parent.tl=get(c,'tl');
                parent.tn=get(c,'tn');
                parent.eventlist(j,:)=[parent.tn,2,j];
                i=0;
            else
                if parent.tl<get(c,'tl')
                    parent.tl=get(c,'tl');
                end
                parent.eventlist(i+j,:)=[get(c,'tn'),2,j];
                if parent.tn>parent.eventlist(i+j,1)
                    parent.tn=parent.eventlist(i+j,1);
                end
            end
        end
    end
    parent.comps=comps;
end
```

C3 get()

```
function value=get(c,prop,component)
%
%get method for class coupled
%value=get(c,prop,component)
%variables:
%   value       //type: depends on prop and val
%   c           //type: object
%   prop        //type: string
%   component         //type: string
%variables used at simulator
%   tl      //type: float            //last transition time
%   tn      //type: float            //next transition time
%   x       //type: structure        //set of input values
%   y       //type: structure        //set of output values
%   Da      //type: cell-array with strings  //atomic objects at c
%   Dc      //type: cell-array with strings  //coupled objects at c
%   Zid     //type: cell-array with strings  //"i-to-d-matrix"
%       (if Da, Dc or Zid are empty: please type [] )
%   eventlist //type: 2-D matrix with stringss   //[a,b,c;...]
%   comps     //type: structure      //field for saving DEVS-objects

if nargin == 2
    switch prop
        case 'tl'
            value=c.tl;
        case 'tn'
            value=c.tn;
        case 'x'
            value=c.x;
        case 'y'
            value=c.y;
        case 'Da'
            value=c.Da;
        case 'Dc'
            value=c.Dc;
        case 'Zid'
            value=c.Zid;
        case 'eventlist'
            value=c.eventlist;
        case 'comps'
            value=c.comps;
        otherwise
            disp('mistake at get method for class coupled');
    end
%used for getting an object in comps
elseif nargin ==3
    if strcmp(prop,'comps')
        value=getfield(c.comps,component);
    end
else
    disp('mistake at get method for class coupled');
end
```

C4 set()

```
function c=set(c,prop,val,component)
%set method for class coupled
%value=get(c,prop,val,component)
%variables:
%   c           //type: object
%   prop        //type: string
%   val         //type: depends on prop and component
%   component   //type: cell-array with strings
%variables used at simulator
%   tl      //type: float            //last transition time
%   tn      //type: float            //next transition time
%   x       //type: structure        //set of input values
%   y       //type: structure        //set of output values
%   Da      //type: cell-array with strings  //atomic objects at c
%   Dc      //type: cell-array with strings  //coupled objects at c
%   Zid     //type: cell-array with strings  //"i-to-d-matrix"
%                                            //{'from.','port','to.','port'}
%       (if Da, Dc or Zid are empty: please type [] )
%   eventlist  //type: 2-D matrix with strings //[a,b,c;...]
%   comps      //type: structure     //field for saving DEVS-objects
if nargin==3
    switch prop
        case 'tl'
            c.tl=val;
        case 'tn'
            c.tn=val;
        case 'x'
            c.x=val;
        case 'y'
            c.y=val;
        case 'Da'
            c.Da=val;
        case 'Dc'
            c.Dc=val;
        case 'Zid'
            c.Zid=val;
        case 'eventlist'
            c.eventlist=val;
        case 'comps'
            c.comps=val;
        otherwise
            disp('mistake at set method for class coupled');
    end
elseif nargin==4
    if length(component)>=2
        data(1)=c.comps.(char(component(1)));
        for i=2:length(component)-1
            data(i)=get(data(i-1),'comps',char(component(i)));
        end
        m=get(data(end),'comps',char(component(end)));
        m=set(m,prop,val);
        comps=get(data(end),'comps');
        comps.(char(component(end)))=m;
        data(end)=set(data(end),'comps',comps);
        for i=length(data)-1:-1:1
            comps=get(data(i),'comps');
            comps.(char(component(i)))=m;
            data(i)=set(data(i),'comps',comps);
        end
        c.comps.(char(component(1)))=data(1);
    else
        m=set(c.comps.(char(component(1))),prop,val);
        c.comps.(char(component(1)))=m;
    end
else
   disp('mistake at set method for class coupled');
end
```

D Methoden der Klasse processing_block

D1 am_proc_block()

```
function m=am_proc_block(x,y,s,e,service_time)
%
%constructor method for class am_proc_block
%m=am_generator(x,y,s,e,service_time)
%variables:
%   m           //type: object
%   x           //type: structure    //set of input values
%   y           //type: structure    //set of output values
%   s           //type: structure    //set of states
%   e           //type: float        //time elapsed since last
%                                     transition (only for initialization)
%   proctime    //type: float        //processing time

if nargin==5
        a=atomic(x,y,s,e);
        m.service_time=service_time;
        m=class(m,'am_proc_block',a);
else
    disp('mistake at constructor method for class am_proc_block');
end
```

D2 deltaextfun()

```
function m=deltaextfun(m,gt)
%
%method external transition function
%m=deltaextfun(m,gt)
%varibales:
%   m   //type: object
%   gt  //type: float    //current simulation time

s=get(m.atomic,'s');
x=get(m.atomic,'x');
e=gt-get(m.atomic,'tl');

%############################external transition function##################
%Die external-transition-function ermittelt aus:
%   - s - Zuständen (z.B. s.phase=4),
%   - x - Eingängen (z.B. x.p1='active') und
%   - e - vergangene Zeitspanne seit letztem Ereignis (z.B. e=2)
%die neuen Zustände s
%#############################################################################

        if strcmp(s.phase,'passiv') && ~isempty(x.p1) && isempty(x.p2)...
                  && s.q2==0
            s.phase='passiv';
            s.q1=s.q1+x.p1;
        elseif strcmp(s.phase,'passiv') && isempty(x.p1) &&...
                      ~isempty(x.p2) && s.q1==0
            s.phase='passiv';
            s.q2=s.q2+x.p2;
        elseif strcmp(s.phase,'passiv') && ~isempty(x.p1) &&...
                      isempty(x.p2) && s.q2~=0
            s.phase='activ';
            s.sigma=m.service_time;
            s.q1=s.q1+x.p1-1;
            s.q2=s.q2-1;
```

```
        elseif strcmp(s.phase,'passiv') && isempty(x.p1) &&...
                    ~isempty(x.p2) && s.q1~=0
            s.phase='activ';
            s.sigma=m.service_time;
            s.q1=s.q1-1;
            s.q2=s.q2+x.p2-1;
        elseif strcmp(s.phase,'passiv') && ~isempty(x.p1) &&...
                    ~isempty(x.p2)
            s.phase='activ';
            s.sigma=m.service_time;
            s.q1=s.q1+x.p1-1;
            s.q2=s.q2+x.p2-1;
        elseif strcmp(s.phase,'activ')
            s.phase='activ';
            s.sigma=s.sigma-e;
            if ~isempty(x.p1) s.q1=s.q1+x.p1;end
            if ~isempty(x.p2) s.q2=s.q2+x.p2;end
        end

%#########################################################################

m.atomic=set(m.atomic,'s',s);
```

D3 deltaintfun()

```
function m=deltaintfun(m)
%
%method internal transition function
%m=deltaintfun(m)
%variables:
%   m   //type: object

s=get(m.atomic,'s');

%##############################internal transition function##################
%Die internal-transition-function ermittelt aus:
%   - s - Zuständen (z.B. s.phase=4),
%die neuen Zustände s
%#########################################################################

        if s.q1>=1 && s.q2>=1
            s.sigma=m.service_time;
            s.q1=s.q1-1;
            s.q2=s.q2-1;
        else
            s.phase='passiv';
            s.sigma=inf;
        end

%#########################################################################

m.atomic=set(m.atomic,'s',s);
```

D4 deltaconffun()

```
function m=deltaconffun(m,gt)
%
%method confluent transition function
%m=deltaconffun(m,gt)
%variables:
%   m   //type: object
%   gt  //type: float    //current simulation time

s=get(m.atomic,'s');
x=get(m.atomic,'x');
e=gt-get(m.atomic,'tl');

%############################confluent function#############################
%Die confluent-function ermittelt aus:
%   - s - Zuständen (z.B. s.state1=4),
%   - x - Eingängen (z.B. x.p1='active') und
%   - e - vergangene Zeitspanne seit letztem Ereignis (z.B. e=2)
%die neuen Zustände s
%
%ES MÜSSEN IN JEDEM FALL OBJEKTE ÜBERGEBEN WERDEN!
%VERÄNDERUNGEN AN DEN DATEN MÜSSEN ALSO WIEDER AUF M GESPEICHERT WERDEN.
%
%Ausschnitt aus Datenstruktur von m zum Zweck der Rückspeicherung:
%   m.atomic.s.state1 .... stateX
%   m.atomic.x.p1 .... pX
%   m.atomic.y.p1 .... pX
%############################################################################

        m=deltaintfun(m);
        m.atomic=set(m.atomic,'tl',gt); %//führt in deltaextfun zu e=0
        m=deltaextfun(m,gt);

%############################################################################

tn=gt+tafun(m);
m.atomic=set(m.atomic,'tl',gt);
m.atomic=set(m.atomic,'tn',tn);
```

D5 lambdafun()

```
function m=lambdafun(m)
%
%method lambda-function
%m=lambdafun(m)
%variables:
%   m   //type: object

s=get(m.atomic,'s');

%############################Lamdafunktion#################################
%Die lambda-function ermittelt aus:
%   - s - Zuständen (z.B. s.state1=4),
%die Belegung der Ausgänge y (z.B. y.p1=1)
%############################################################################

y=struct('p1',1);

%############################################################################

m.atomic=set(m.atomic,'y',y);
```

D6 tafun()

```
function ta=tafun(m)
%
%method time advance function
%ta=tafun(m)
%variables:
%   m   //type: object
%   ta  //type: float

s=get(m.atomic,'s');

%############################ta-function#####################################
%Die time-function ermittelt aus:
%   - s - Zuständen (z.B. s.state1=4),
%die Zeit bis zum nächsten Ereignis tn (z.B. ta=2)
%############################################################################

ta=s.sigma;

%############################################################################
```

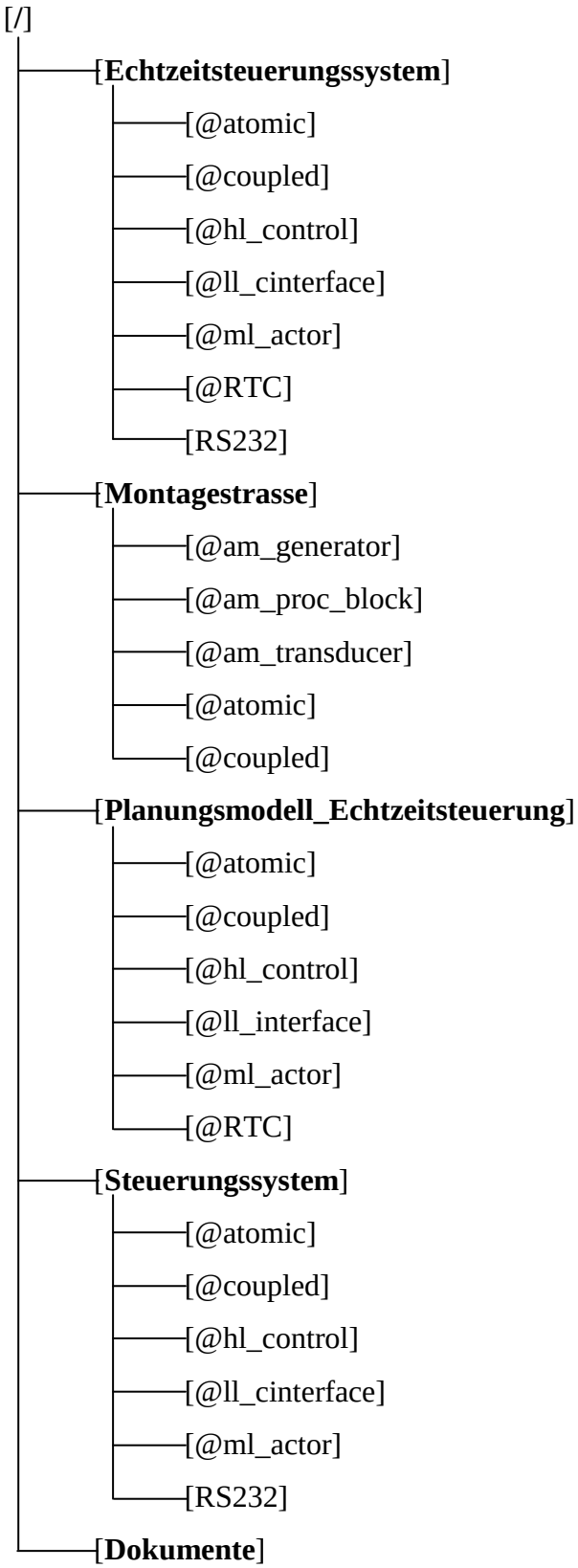
D7 get()

```
function value=get(m,prop)
%
%get method
%value=get(m,prop)
%variables:
%   value   //type: depends on prop
%   m       //type: object
%   prop    //type: string

if nargin == 2
    switch prop
        case 'e'
            value=get(m.atomic,'e');
        case 'tl'
            value=get(m.atomic,'tl');
        case 'tn'
            value=get(m.atomic,'tn');
        case 'x'
            value=get(m.atomic,'x');
        case 'y'
            value=get(m.atomic,'y');
        case 's'
            value=get(m.atomic,'s');
        case 'parent'
            value=m.atomic;
        case 'service_time'
            value=m.service_time;
        otherwise
            disp('mistake at get method for class am_proc_block');
    end
end
```

D8 set()

```
function m=set(m,prob,val)
%
%set method
%m=set(m,prop,val)
%variables:
%   m       //type: object
%   prop    //type: string
%   val     //type: depends on prop

if nargin==3
    switch prob
        case 'e'
            m.atomic=set(m.atomic,'e',val);
        case 'tl'
            m.atomic=set(m.atomic,'tl',val);
        case 'tn'
            m.atomic=set(m.atomic,'tn',val);
        case 'x'
            m.atomic=set(m.atomic,'x',val);
        case 'y'
            m.atomic=set(m.atomic,'y',val);
        case 's'
            m.atomic=set(m.atomic,'s',val);
        case 'service_time'
            m.service_time=val;
        otherwise
            disp('mistake at set method for class am_proc_block');
    end
end
```

Datenträger



Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Wismar, den 25.01.2010

Tobias Schwatinski