B. Reaktionsregeln Fahrerloser Transportsysteme

In diesem Kapitel werden die aus Kapitel 5 bekannten Fahrerlosen Transportsysteme benutzt, um ein Beispiel für Agentenbasierte Simulation darzustellen. Dabei wird ein eher kleines Beispiel in Regeln ausformuliert, um den Rahmen dieser Arbeit nicht zu sprengen. Mögliche Erweiterungen, die das Szenario exakter und damit realistischer beschreiben, werden jeweils natürlichsprachlich angegeben.

Da Nachrichten nicht explizit in den Regeln für den Umgebungssimulator spezifiziert werden müssen, brauchen *TA-Manager* und *Verkehrsregler* bei den U-Regeln nicht beachtet zu werden, so dass die U-Regeln nur die Agententypen *Maschine* und *FTF* betreffen, wobei die Umgebungsereignisse hier bei den Agententypen eingeordnet sind, die sie wahrnehmen.

B.1. U-Regeln Maschinen

Als Zusicherung für das Auftreten der Aktion *aufnehmenTS* kann gegeben werden, dass der Eingangspuffer der Maschine nicht leer war.

Ereignis	**Maschine [i] : aufnehmenTS ( ) @ t**
Zusicherung	**context Maschine [i]** **anz_ein > 0**

Zusicherung Aktion_Maschine_aufnehmenTS

Führt eine Maschine die Aktion *aufnehmenTS* aus, so nimmt sie ein TS aus ihrem Eingangspuffer und beginnt sofort mit der Verarbeitung dieses TS. Die Anzahl der TS im Eingangspuffer verringert sich um 1. Als Folgeereignis wird die Beendigung der Produktion des TS erzeugt.

Ereignis	**Maschine [i] : aufnehmenTS ( ) @ t**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context Maschine [i]** **anz_ein = anz_ein@pre – 1**
Folgeereignisse	**{ ProduktionFertig (i) @ (t + Gleich_ZV (** a_i , b_i **)) }**
Wahrnehmungen	ø

Regel 72: Aktion_Maschine_aufnehmenTS.1

Als Zusicherung für das Auftreten der Aktion *ablegenTS* kann gegeben werden, dass der Eingangspuffer der Maschine nicht voll war.

Ereignis	**Maschine [i] : ablegenTS ( ) @ t**
Zusicherung	**context Maschine [i]** **anz_aus < kap_aus**

Zusicherung Aktion_Maschine_ablegenTS

Führt eine Maschine die Aktion *ablegenTS* aus, so legt sie das TS, das sie gerade fertig bearbeitet hat, in ihren Ausgangspuffer. Die Anzahl der TS im Ausgangspuffer erhöht sich um 1.

Ereignis	`Maschine [ i ] : ablegenTS ( ) @ t`
Ereignisbedingung	-
Zustandsbedingung	-

Zustandseffekt	`context Maschine [ i ]`
	
	`anz_aus = anz_aus@pre + 1`

Folgeereignisse	Ø
Wahrnehmungen	Ø

Regel 73: Aktion_Maschine_ablegenTS.1

Ist die Produktion eines TS bei einer Maschine fertig, so nimmt die Maschine dies wahr.

Ereignis	`ProduktionFertig ( i ) @ ( t )`
Ereignisbedingung	-
Zustandsbedingung	-
Zustandseffekt	-
Folgeereignisse	Ø
Wahrnehmungen	`{ Maschine [ i ] : { Produktion_fertig ( ) } }`

Regel 74: Ereignis_ProduktionFertig.1

B.2. U- Regeln FTF

Als Zusicherung für das Auftreten der Aktion aufnehmenTS kann gegeben werden, dass das FTF unbeladen war. Ferner steht es entweder im Eingangslager oder auf einem Knoten, an dem sich eine Maschine befindet, deren Ausgangspuffer nicht leer war.

Ereignis	`FTF [ i ] : aufnehmenTS ( ) @ t`

Zusicherung	`context`
	`f : FTF [ i ]`
	`m : Maschine [ f.ort.maschine ]`
	
	`f.belZust = unbeladen and f.ort.typ = knoten and f.bewZust = stehend and ( ( f.ort = eingangslager ) or ( f.ort.maschine -> notEmpty() and m.anz_aus > 0 ) )`

Zusicherung Aktion_FTF_aufnehmenTS

Führt ein FTF die Aktion *aufnehmenTS* aus und befindet sich im Eingangslager, so nimmt es ein TS aus dem Eingangslager auf. Das FTF hat nun den Beladungszustand *beladen*.

Ereignis	`FTF [ i ] : aufnehmenTS ( ) @ t`
Ereignisbedingung	–

Zustandsbedingung	`context`
	`f : FTF [ i ]`
	
	`f.ort.maschine -> isEmpty()`

Zustandseffekt	`context`
	`f : FTF [ i ]`
	
	`f.belZust = beladen`

Folgeereignisse	ø
Wahrnehmungen	ø

Regel 75: Aktion_FTF_aufnehmenTS.1

Führt ein FTF die Aktion *aufnehmenTS* aus und befindet es sich an einem Knoten mit einer Maschine, so nimmt es ein TS aus dem Ausgangspuffer der Maschine. Das FTF hat nun den Beladungszustand *beladen*. Die Anzahl der TS im Ausgangspuffer der Maschine verringert sich um 1. Die Maschine nimmt wahr, dass aus ihrem Ausgangspuffer ein TS entnommen wurde.

Ereignis	`FTF [ i ] : aufnehmenTS ( ) @ t`
Ereignisbedingung	–
Zustandsbedingung	–

Zustandseffekt	`context`
	`f : FTF [ i ]`
	`m : Maschine [ f.ort.maschine ]`
	
	`m.anz_aus = m.anz_aus@pre – 1 and f.belZust = beladen`

Folgeereignisse	ø
Wahrnehmungen	`{ Maschine [ FTF [ i ].ort.maschine ] : { TSwirdAbgeholt ( ) } }`

Regel 76: Aktion_FTF_aufnehmenTS.2

Als Zusicherung für das Auftreten der Aktion *ablegenTS* kann gegeben werden, dass das FTF beladen war. Ferner steht es entweder im Ausgangslager oder auf einem Knoten, an dem sich eine Maschine befindet, deren Eingangspuffer nicht voll war.

Ereignis	`FTF [ i ] : ablegenTS ( ) @ t`

Zusicherung	`context`
	`f : FTF [ i ]`
	`m : Maschine [ f.ort.maschine ]`
	
	`f.belZust = beladen and f.ort.typ = knoten and f.bewZust = stehend and ( ( f.ort = ausgangslager ) or ( f.ort.maschine -> notEmpty() and m.anz_ein < m.kap_ein ) )`

Zusicherung Aktion_FTF_ablegenTS

Führt ein FTF die Aktion *ablegenTS* aus und befindet sich im Ausgangslager, so legt es ein TS im Ausgangslager ab. Das FTF hat nun den Beladungszustand *unbeladen*.

Ereignis	`FTF [ i ] : ablegenTS ( ) @ t`
Ereignisbedingung	–

Zustandsbedingung	`context`
	`f : FTF [ i ]`
	
	`f.ort.maschine -> isEmpty()`

Zustandseffekt	`context`
	`f : FTF [ i ]`
	
	`f.belZust = unbeladen`

Folgeereignisse	ø
Wahrnehmungen	ø

Regel 77: Aktion_FTF_ablegenTS.1

Führt ein FTF die Aktion *ablegenTS* aus und befindet es sich an einem Knoten mit einer Maschine, so legt es ein TS in den Eingangspuffer der Maschine. Das FTF hat nun den Beladungszustand *unbeladen*. Die Anzahl der TS im Eingangspuffer der Maschine erhöht sich um 1. Die Maschine nimmt wahr, dass in ihren Eingangspuffer ein TS gelegt wurde.

Ereignis	`FTF [ i ] : ablegenTS ( ) @ t`
Ereignisbedingung	–
Zustandsbedingung	–

Zustandseffekt	`context`
	`f : FTF [ i ]`
	`m : Maschine [ f.ort.maschine ]`
	
	`m.anz_ein = m.anz_ein@pre + 1 and f.belZust = unbeladen`

Folgeereignisse	ø
Wahrnehmungen	`{ Maschine [ FTF [ i ].ort.maschine ] : { TSkommtAn ( ) } }`

Regel 78: Aktion_FTF_ablegenTS.2

Als Zusicherung für das Auftreten der Aktion *abfahrenKnotenUndSegment* kann gegeben werden, dass das FTF am Ende eines Segments steht, das am genannten Knoten endet und dass das genannte Segment am genannten Knoten beginnt.

Ereignis	`FTF [ i ] : abfahrenKnotenUndSegment ( k, s ) @ t`

Zusicherung	`context FTF [ i ]`
	
	`k.typ = knoten and s.typ = segment and ort.typ=segment and`
	`ort.zielknoten = k and s.startknoten = k and bewZust = stehend`

Zusicherung Aktion_FTF_abfahrenKnotenUndSegment

Führt ein FTF die Aktion *abfahrenKnotenUndSegment* aus, so überfährt es den Knoten und anschließend fährt es das Segment komplett bis zum Ende ab. Das FTF ist nun fahrend und befindet sich auf dem Knoten. Als Folgeereignis wird erzeugt, dass das FTF nach c_1 Zeiteinheiten den Knoten passiert hat.

Ereignis	**FTF [i] : abfahrenKnotenUndSegment (k, s) @ t**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context FTF [i]** **ort = k and bewZust = fahrend**
Folgeereignisse	**{ Knoten_passiert (i, k, s) @ (t +** c_1 **) }**
Wahrnehmungen	ø

Regel 79: Aktion_FTF_abfahrenKnotenUndSegment.1

Als Zusicherung für das Auftreten der Aktion *befahrenKnoten* kann gegeben werden, dass das FTF am Ende eines Segments steht, das am genannten Knoten endet.

Ereignis	**FTF [i] : befahrenKnoten (k) @ t**
Zusicherung	**context FTF [i]** **k.typ = knoten and ort.typ=segment and ort.zielknoten = k and bewZust = stehend**

Zusicherung Aktion_FTF_befahrenKnoten

Führt ein FTF die Aktion *befahrenKnoten* aus, so fährt es in einen Knoten ein und bleibt anschließend im Knoten stehen. Das FTF ist nun fahrend und befindet sich auf dem Knoten. Als Folgeereignis wird erzeugt, dass das FTF nach c_2 Zeiteinheiten den Knoten erreicht hat und auf ihm stehengeblieben ist.

Ereignis	**FTF [i] : befahrenKnoten (k) @ t**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context FTF [i]** **ort = k and bewZust = fahrend**
Folgeereignisse	**{ Knoten_erreicht (i, k) @ (t +** c_2 **) }**
Wahrnehmungen	ø

Regel 80: Aktion_FTF_befahrenKnoten.1

Als Zusicherung für das Auftreten der Aktion *abfahrenSegment* kann gegeben werden, dass das FTF auf dem genannten Knoten steht und dass das genannte Segment am genannten Knoten beginnt.

Ereignis	**FTF [i] : abfahrenSegment (k, s) @ t**
Zusicherung	**context FTF [i]** **k.typ = knoten and s.typ = segment and ort = k and bewZust = stehend and s.startknoten = k**

Zusicherung Aktion_FTF_abfahrenSegment

Führt ein FTF die Aktion *abfahrenSegment* aus, so fährt es aus dem Knoten aus, in dem es gerade steht und anschließend fährt es das Segment komplett bis zum Ende ab. Das FTF ist nun fahrend. Als Folgeereignis wird erzeugt, dass das FTF nach c_3 Zeiteinheiten den Knoten verlassen hat.

Ereignis	**FTF [i] : abfahrenSegment (k, s) @ t**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context FTF [i]** **bewZust = fahrend**
Folgeereignisse	**{ KnotenPassiert (i, k, s) @ (t +** c_3 **) }**
Wahrnehmungen	ø

Regel 81: Aktion_FTF_abfahrenSegment.1

Als Zusicherung für das Auftreten des Ereignis *KnotenPassiert* kann gegeben werden, dass das FTF bisher auf dem genannten Knoten fährt und das genannte Segment am genannten Knoten beginnt.

Ereignis	**KnotenPassiert (i, k, s) @ (t)**
Zusicherung	**context FTF [i]** **k.typ = knoten and s.typ = segment and ort = k and bewZust = fahrend and s.startknoten = k**

Zusicherung Ereignis_KnotenPassiert

Hat ein FTF einen Knoten passiert, so nimmt es dies wahr. Das FTF befindet sich nun auf dem erreichten Segment. Als Folgeereignis wird erzeugt, dass das FTF das Ende des Segments erreicht. Die Zeit, die das FTF zum Abfahren des Segments benötigt, ergibt sich aus der Segmentlänge geteilt durch die Geschwindigkeit des FTF.[100]

[100] Das ist eine ungenaue Berechnung, da das FTF ja zunächst beschleunigen muss, um seine Geschwindigkeit zu erreichen und am Ende abbremsen, um am Segmentende zum Stand zu kommen. Enthält das Segment mehrere Teilsegmente, die durch Kurven miteinander verbunden sind, so wird die hier vorgenommene Berechnung noch ungenauer, falls zum Durchfahren einer Kurve die Geschwindigkeit des FTF verringert werden muss. Genauere Formeln für die Berechnung der Zeit, die ein FTF für eine Strecke benötigt, die Beschleunigen und Abbremsen berücksichtigen, findet man bei Balko [Bal00].

Ereignis	**KnotenPassiert (i, k, s) @ (t)**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context FTF [i]** **ort = s**
Folgeereignisse	**{ SegmentendeErreicht (i, s) @ (t + s.länge /** v_i **) }**
Wahrnehmungen	**{ FTF [i] : { KnotenPassiert (k, s) } }**

Regel 82: Ereignis_KnotenPassiert.1

Als Zusicherung für das Auftreten des Ereignis *KnotenErreicht* kann gegeben werden, dass das FTF bisher auf dem genannten Knoten fährt.

Ereignis	**KnotenErreicht (i, k) @ (t)**
Zusicherung	**context FTF [i]** **k.typ = knoten and ort = k and bewZust = fahrend**

Zusicherung Ereignis_KnotenErreicht

Hat ein FTF einen Knoten erreicht, so nimmt es dies wahr. Das FTF steht nun.

Ereignis	**KnotenErreicht (i, k) @ (t)**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**context FTF [i]** **bewZust = stehend**
Folgeereignisse	ø
Wahrnehmungen	**{ FTF [i] : { KnotenErreicht (k) } }**

Regel 83: Ereignis_KnotenErreicht.1

Als Zusicherung für das Auftreten des Ereignis *SegmentendeErreicht* kann gegeben werden, dass das FTF bisher auf dem genannten Segment fährt.

Ereignis	**SegmentendeErreicht (i, s) @ (t)**
Zusicherung	**context FTF [i]** **s.typ = segment and ort = s and bewZust = fahrend**

Zusicherung Ereignis_SegmentendeErreicht

Hat ein FTF ein Segmentende erreicht, so nimmt es dies wahr. Das FTF steht nun.

Ereignis	`SegmentendeErreicht ( i, s ) @ ( t )`
Ereignisbedingung	–
Zustandsbedingung	–

Zustandseffekt	`context FTF [ i ]`
	
	`bewZust = stehend`

Folgeereignisse	Ø
Wahrnehmungen	`{ FTF [ i ] : { SegmentendeErreicht ( s ) } }`

Regel 84: Ereignis_SegmentendeErreicht.1

B.3. A-Regeln Maschine

Als Zusicherung für das Auftreten der Wahrnehmung *TSkommtAn* kann gegeben werden, dass der Eingangspuffer der Maschine nicht voll war.

Ereignis	`TSkommtAn ( )`
Zusicherung	`anz_ein < kap_ein`

Zusicherung Wahrnehmung_TSkommtAn

Kommt ein TS an und ist die Maschine im Leerlauf, so verringert sich die Anzahl der bestellten TS, das TS wird sofort aufgenommen und die Maschine beginnt zu produzieren. Die Maschine überprüft, ob es notwendig ist, einen Beschaffungsauftrag zu erteilen.

Ereignis	`TSkommtAn ( )`
Ereignisbedingung	–
Zustandsbedingung	`prodZust = leerlauf`
Zustandseffekt	`bestellteTS = bestellteTS@pre – 1 and prodZust = produzierend`
Folge-Zeitereignisse	`{ ueberpruefenBA ( ) @ c }`
Nachrichten	Ø
Aktionen	`{ aufnehmenTS ( ) }`

Regel 85: Wahrnehmung_TSkommtAn.1

Ansonsten vergrößert sich der Bestand des Eingangspuffers, die Anzahl der bestellten TS verringert sich.

Ereignis	`TSkommtAn ( )`
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	`bestellteTS = bestellteTS@pre – 1 and anz_ein = anz_ein@pre + 1`
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 86: Wahrnehmung_TSkommtAn.2

Als Zusicherung für das Auftreten der Wahrnehmung *TSwirdAbgeholt* kann gegeben werden, dass der Ausgangspuffer der Maschine nicht leer war.

Ereignis	**TSwirdAbgeholt ( )**
Zusicherung	**anz_aus > 0**

Zusicherung Wahrnehmung_TSwirdAbgeholt

Wird ein TS abgeholt, ist die Maschine bisher blockiert und befindet sich mindestens ein TS im Eingangspuffer, so legt die Maschine das fertige TS in den Ausgangspuffer und nimmt sich ein TS aus dem Eingangspuffer. Damit verringert sich die Anzahl der abzuholenden TS und die Anzahl der TS im Eingangspuffer. Das TS wird sofort aufgenommen und die Maschine beginnt zu produzieren. Die Anzahl der TS im Ausgangspuffer bleibt unverändert, da sowohl ein TS durch ein FTF aus dem Ausgangspuffer abgeholt wird als auch das fertige TS in den Ausgangspuffer gelegt wird. Die Maschine überprüft sowohl, ob es notwendig ist, einen Beschaffungsauftrag zu erteilen als auch, ob es notwendig ist, einen Abholauftrag zu erteilen.

Ereignis	**TSwirdAbgeholt ( )**
Ereignisbedingung	**–**
Zustandsbedingung	**prodZust = blockiert and anz_ein > 0**
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre – 1 and anz_ein = anz_ein@pre – 1 and prodZust = produzierend**
Folge-Zeitereignisse	**{ ueberpruefenBA ( ) @ c , ueberpruefenAA ( ) @ c }**
Nachrichten	**Ø**
Aktionen	**{ ablegenTS ( ), aufnehmenTS ( ) }**

Regel 87: Wahrnehmung_TSwirdAbgeholt.1

Wird ein TS abgeholt, ist die Maschine bisher blockiert und befindet sich kein TS im Eingangspuffer, so legt die Maschine das fertige TS in den Ausgangspuffer und befindet sich danach im Leerlauf. Damit verringert sich die Anzahl der abzuholenden TS. Die Maschine überprüft, ob es notwendig ist, einen Abholauftrag zu erteilen.

Ereignis	**TSwirdAbgeholt ( )**
Ereignisbedingung	**–**
Zustandsbedingung	**prodZust = blockiert**
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre – 1 and prodZust = leerlauf**
Folge-Zeitereignisse	**{ ueberpruefenAA ( ) @ c }**
Nachrichten	**Ø**
Aktionen	**{ ablegenTS ( ) }**

Regel 88: Wahrnehmung_TSwirdAbgeholt.2

Ansonsten verringert sich die Anzahl der abzuholenden TS und die Anzahl der TS im Ausgangspuffer.

Ereignis	**TSwirdAbgeholt ( )**
Ereignisbedingung	**–**
Zustandsbedingung	**–**
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre – 1 and anz_aus = anz_aus@pre – 1**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 89: Wahrnehmung_TSwirdAbgeholt.3

Als Zusicherung für das Auftreten der Wahrnehmung *ProduktionFertig* kann gegeben werden, dass die Maschine im Zustand *produzierend* war.

Ereignis	**ProduktionFertig ( )**
Zusicherung	**prodZust = produzierend**

Zusicherung Wahrnehmung_ProduktionFertig

Ist die Produktion eines TS fertig und ist der Ausgangspuffer voll, so ist die Maschine blockiert.

Ereignis	**ProduktionFertig ( )**
Ereignisbedingung	**–**
Zustandsbedingung	**anz_aus = kap_aus**
Zustandseffekt	**prodZust = blockiert**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 90: Wahrnehmung_ProduktionFertig.1

Ist die Produktion eines TS fertig, ist der Ausgangspuffer noch nicht voll, aber ist der Eingangspuffer leer, so legt die Maschine das TS ab und ist danach im Leerlauf. Damit erhöht sich die Anzahl der TS im Ausgangspuffer. Die Maschine überprüft, ob es notwendig ist, einen Abholauftrag zu erteilen.

Ereignis	**ProduktionFertig ( )**
Ereignisbedingung	**–**
Zustandsbedingung	**anz_ein = 0**
Zustandseffekt	**anz_aus = anz_aus@pre + 1 and prodZust = leerlauf**
Folge-Zeitereignisse	**{ ueberpruefenAA ( ) @ c }**
Nachrichten	**Ø**
Aktionen	**{ ablegenTS ( ) }**

Regel 91: Wahrnehmung_ProduktionFertig.2

Ist die Produktion eines TS fertig, ist der Ausgangspuffer noch nicht voll und der Eingangspuffer nicht leer, so legt die Maschine das fertige TS in den Ausgangspuffer und nimmt sich ein TS aus dem

Eingangspuffer. Damit erhöht sich die Anzahl der TS im Ausgangspuffer und es verringert sich die Anzahl der TS im Eingangspuffer. Das TS wird sofort aufgenommen und die Maschine beginnt zu produzieren. Die Maschine überprüft sowohl, ob es notwendig ist, einen Beschaffungsauftrag zu erteilen als auch, ob es notwendig ist, einen Abholauftrag zu erteilen.

Ereignis	**ProduktionFertig ( )**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**anz_aus = anz_aus@pre + 1 and anz_ein = anz_ein@pre – 1**
Folge-Zeitereignisse	**{ ueberpruefenBA ( ) @ c , ueberpruefenAA ( ) @ c }**
Nachrichten	Ø
Aktionen	**{ ablegenTS ( ), aufnehmenTS ( ) }**

Regel 92: Wahrnehmung_ProduktionFertig.3

Bei der Überprüfung, ob ein Beschaffungsauftrag erteilt werden muss, ermittelt die Maschine, ob der Inhalt des Eingangspuffers zuzüglich der bereits bestellten TS die Schwelle C_EIN nicht überschreitet.

Ist das der Fall, so wird ein Beschaffungsauftrag an den TAManager gesendet. Die Anzahl der bestellten TS erhöht sich um 1. Es wird überprüft, ob weitere Beschaffungsaufträge erteilt werden müssen.

Ereignis	**ueberpruefenBA ( )**
Ereignisbedingung	–
Zustandsbedingung	**anz_ein + bestellteTS <= c_ein**
Zustandseffekt	**bestellteTS = bestellteTS@pre + 1**
Folge-Zeitereignisse	**{ ueberpruefenBA ( ) @ c }**
Nachrichten	**{ erteilenBA ( ) → TAManager [manager] }**
Aktionen	Ø

Regel 93: Zeitereignis_ueberpruefenBA.1

Ansonsten muss nichts unternommen werden.

Ereignis	**ueberpruefenBA ( )**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 94: Zeitereignis_ueberpruefenBA.2

Bei der Überprüfung, ob ein Abholauftrag erteilt werden muss, ermittelt die Maschine, ob der Inhalt des Ausgangspuffers abzüglich der bereits als abzuholend gemeldeten TS die Schwelle C_AUS nicht unterschreitet.

Ist das der Fall, so wird ein Abholauftrag an den TAManager gesendet. Die Anzahl der abzuholenden TS erhöht sich um 1. Es wird überprüft, ob weitere Abholaufträge erteilt werden müssen.

Ereignis	**ueberpruefenAA ( )**
Ereignisbedingung	–
Zustandsbedingung	**anz_aus – abzuholendeTS >= c_aus**
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre + 1**
Folge-Zeitereignisse	**{ ueberpruefenAA ( ) @ c }**
Nachrichten	**{ erteilenAA ( ) → TAManager [manager] }**
Aktionen	Ø

Regel 95: Zeitereignis_ueberpruefenAA.1

Ansonsten muss nichts unternommen werden.

Ereignis	**ueberpruefenAA ( )**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 96: Zeitereignis_ueberpruefenAA.2

Als Zusicherung für den Empfang der Nachricht *anfordenAnnahmeTS* kann gegeben werden, dass der Absender der der Maschine zugeordnete TAManager ist.

Ereignis	**Empfang(anfordernAnnahmeTS (AA) ← TAManager [i])**
Zusicherung	**i = manager**

Zusicherung Nachricht_anfordernAnnahmeTS

Ist der Eingangspuffer zuzüglich der bereits bestellten TS noch nicht voll, so kann die Maschine ein weiteres TS aufnehmen. Sie schickt eine entsprechende Nachricht an den TAManager. Die Anzahl der bestellten TS erhöht sich somit.

Ereignis	**Empfang(anfordernAnnahmeTS (AA) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	**anz_ein + bestellteTS < kap_ein**
Zustandseffekt	**bestellteTS = bestellteTS@pre + 1**
Folge-Zeitereignisse	Ø
Nachrichten	**{ bestaetigenAnnahme (AA) → TAManager [i] }**
Aktionen	Ø

Regel 97: Nachricht_anfordernAnnahmeTS.1

Ansonsten kann sie kein weiteres TS aufnehmen. Auch hier schickt sie eine entsprechende Nachricht an den TAManager.

Ereignis	**Empfang(anfordernAnnahmeTS (AA) ← TAManager [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**–**
Zustandseffekt	**–**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ ablehnenAnnahme (AA) → TAManager [i] }**
Aktionen	**Ø**

Regel 98: Nachricht_anfordernAnnahmeTS.2

Als Zusicherung für den Empfang der Nachricht *anfordernAbgabeTS* kann gegeben werden, dass der Absender der der Maschine zugeordnete TAManager ist.

Ereignis	**Empfang(anfordernAbgabeTS (BA) ← TAManager [i])**
Zusicherung	**i = manager**

Zusicherung Nachricht_anfordernAbgabeTS

Ist der Ausgangspuffer abzüglich der bereits abzuholenden TS noch nicht leer, so kann die Maschine ein weiteres TS abgeben. Sie schickt eine entsprechende Nachricht an den TAManager. Die Anzahl der abzuholenden TS erhöht sich somit.

Ereignis	**Empfang(anfordernAbgabeTS (BA) ← TAManager [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**anz_aus – abzuholendeTS > 0**
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre + 1**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ bestaetigenAbgabe (BA) → TAManager [i] }**
Aktionen	**Ø**

Regel 99: Nachricht_anfordernAbgabeTS.1

Ansonsten kann sie kein weiteres TS abgeben. Auch hier schickt sie eine entsprechende Nachricht an den TAManager.

Ereignis	**Empfang(anfordernAbgabeTS (BA) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	ø
Nachrichten	**{ ablehnenAbgabe (BA) → TAManager [i] }**
Aktionen	ø

Regel 100: Nachricht_anfordernAbgabeTS.2

Als Zusicherung für den Empfang der Nachricht *stornierenAnnahme* kann gegeben werden, dass der Absender der der Maschine zugeordnete TAManager ist und dass vorher der Manager die Anforderung gestellt hat, ein TS aufzunehmen, also muss die Anzahl der bestellten TS größer Null sein.

Ereignis	**Empfang(stornierenAnnahme (AA) ← TAManager [i])**
Zusicherung	**i = manager and bestellteTS > 0**

Zusicherung Nachricht_stornierenAnnahme

Die Maschine verringert die Anzahl der bestellten TS um 1. Da es sein kann, dass die Maschine vorher nur deshalb keinen BA erteilt hat, da sie das zu liefernde TS bereits mit eingeplant hat, muss sie überprüfen, ob es nun notwenig ist, einen BA zu erteilen

Ereignis	**Empfang(stornierenAnnahme (AA) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**bestellteTS = bestellteTS@pre – 1**
Folge-Zeitereignisse	**{ ueberpruefenBA ( ) @ c }**
Nachrichten	ø
Aktionen	ø

Regel 101: Nachricht_stornierenAnnahme.1

Als Zusicherung für den Empfang der Nachricht *stornierenAbgabe* kann gegeben werden, dass der Absender der der Maschine zugeordnete TAManager ist und dass vorher der Manager die Anforderung gestellt hat, ein TS abzugeben, also muss die Anzahl der abzuholenden TS größer Null sein.

Ereignis	**Empfang(stornierenAbgabe (BA) ← TAManager [i])**
Zusicherung	**i = manager and abzuholendeTS > 0**

Zusicherung Nachricht_stornierenAbgabe

Die Maschine verringert die Anzahl der abzuholenden TS um 1. Da es sein kann, dass die Maschine vorher nur deshalb keinen AA erteilt hat, da sie das abzuholende TS bereits mit eingeplant hat, muss sie überprüfen, ob es notwenig ist, nun einen AA zu erteilen

Ereignis	**Empfang(stornierenAbgabe (BA) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**abzuholendeTS = abzuholendeTS@pre – 1**
Folge-Zeitereignisse	**{ ueberpruefenAA ( ) @ c }**
Nachrichten	ø
Aktionen	ø

Regel 102: Nachricht_stornierenAbgabe.1

B.4. Erweiterungen Maschine

Eine denkbare Erweiterung für das Verhalten einer Maschine wäre, dass sie mehrere TA-Manager kennt und bei der Auftragserteilung zwischen ihnen wechseln kann. Dies kann sinnvoll sein, wenn ein TA-Manager ausfällt oder auch, wenn er überlastet ist und TA nicht schnell genug ausschreiben kann oder sich bei ihm nicht genügend FTF bewerben, so dass es sehr lange dauert, bis die TA dieses Managers erfüllt werden. Dadurch kann das Gesamtsystem seine Lastverteilung selbst dynamisch regulieren.

Eine weitere Erweiterung wäre, dass eine Maschine ihre Schwellenwerte C_EIN bzw. C_AUS für die Erteilung eines BA bzw. AA dynamisch verändern kann. Wenn die tatsächliche Anlieferung eines TS oft zu spät erfolgt, so dass sich die Maschine im Leerlauf befindet, so kann sie ihre Schwelle C_EIN erhöhen, um in Zukunft bereits früher mit TS beliefert zu werden. Spiegelbildlich dazu kann sie ihre Schwelle C_AUS senken, wenn die Abholung der TS oft so spät erfolgt, dass die Maschine blockiert ist. Werden hingegen die BA bzw. AA immer sehr schnell erfüllt, kann in Erwägung gezogen werden, die Schwellen C_EIN zu senken bzw. C_AUS zu erhöhen.

B.5. A-Regeln TA-Manager

Erhält der Manager eine Nachricht erteilenAA, so generiert er einen neuen AA, fügt ihn zu den offenen AA hinzu und versucht sofort, ob er diesen AA bearbeiten kann, also einen TA aus ihm generieren kann.

Ereignis	**Empfang(erteilenAA ( ) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**(offeneAA = offeneAA@pre including AA (nextAA@pre, i, Maschine [i].schritt)) and nextAA = nextAA@pre + 1**
Folge-Zeitereignisse	**{ bearbeitenAA (nextAA@pre) @ c }**
Nachrichten	ø
Aktionen	ø

Regel 103: Nachricht_erteilenAA.1

Erhält der Manager eine Nachricht *erteilenBA*, so generiert er einen neuen BA, fügt ihn zu den offenen BA hinzu und versucht sofort, ob er diesen BA bearbeiten kann, also einen TA aus ihm generieren kann.

Ereignis	**Empfang(erteilenBA ( ) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**(offeneBA = offeneBA@pre including BA (nextBA@pre, i, Maschine [i].schritt)) and nextBA = nextBA@pre + 1**
Folge-Zeitereignisse	**{ bearbeitenBA (nextBA@pre) @ c }**
Nachrichten	Ø
Aktionen	Ø

Regel 104: Nachricht_erteilenBA.1

Bearbeitet der Manager einen AA, der sich gar nicht mehr in der Menge der offenen AA befindet, weil inzwischen ein passender BA gefunden wurde, so kann es dieses Ereignis ignorieren.

Ereignis	**bearbeitenAA (AA)**
Ereignisbedingung	–
Zustandsbedingung	**offeneAA -> excludes (AA)**
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 105: Zeitereignis_bearbeitenAA.1

Bearbeitet der Manager einen AA der von einer Maschine mit Arbeitsschritt *n* gestellt wurde, so generiert er einen TA mit Zielort Ausgangslager und informiert alle angemeldeten FTF mittels *cfp*. Der AA ist nun nicht mehr in der Menge der offenen AA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**bearbeitenAA (AA)**
Ereignisbedingung	–
Zustandsbedingung	**AA.schritt = n**
Zustandseffekt	**(offeneAA = offeneAA@pre excluding AA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, AA.maschine.ort, ausgangslager)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, AA.maschine.ort, ausgangslager) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	Ø

Regel 106: Zeitereignis_bearbeitenAA.2

Bearbeitet der Manager einen AA und findet einen zu diesem AA passenden BA in den offenen BA, so generiert er aus den beiden Aufträgen einen TA und informiert alle angemeldeten FTF mittels *cfp*. Der AA und der passende BA sind nun nicht mehr in der Menge der offenen AA bzw. der offenen BA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**bearbeitenAA (AA)**
Ereignisbedingung	–
Zustandsbedingung	**offeneBA -> exists (BA | BA.schritt = AA.schritt + 1)**
Zustandseffekt	**(offeneAA = offeneAA@pre excluding AA) and (offeneBA = offeneBA@pre excluding BA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, AA.maschine.ort, BA.maschine.ort)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, AA.maschine.ort, BA.maschine.ort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	Ø

Regel 107: Zeitereignis_bearbeitenAA.3

Bearbeitet der Manager einen AA und findet einen zu diesem AA keinen passenden BA in den offenen BA, so versucht er, eine Maschine mit Arbeitsschritt *n+1* zu finden, die ein TS aufnehmen kann. An eine dieser Maschinen sendet er eine Anforderung zur Annahme eines TS, alle übrigen werden in der Tabelle der möglichen Partner des AA gespeichert.

Ereignis	**bearbeitenAA (AA)**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**let m = (maschinen -> select (i | Maschine [i].schritt = AA.schritt + 1))** **AA.moeglPart = (m -> excluding j) and j = m.extract_min( )**
Folge-Zeitereignisse	Ø
Nachrichten	**{ anfordernAnnahmeTS (AA) → Maschine [j] }**
Aktionen	Ø

Regel 108: Zeitereignis_bearbeitenAA.4

Bearbeitet der Manager einen BA, der sich gar nicht mehr in der Menge der offenen BA befindet, weil inzwischen ein passender AA gefunden wurde, so kann es dieses Ereignis ignorieren.

Ereignis	**bearbeitenBA (BA)**
Ereignisbedingung	–
Zustandsbedingung	**offeneBA -> excludes (BA)**
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 109: Zeitereignis_bearbeitenBA.1

Bearbeitet der Manager einen BA der von einer Maschine mit Arbeitsschritt 1 gestellt wurde, so generiert er einen TA mit Startort Eingangslager und informiert alle angemeldeten FTF mittels *cfp*.

Der BA ist nun nicht mehr in der Menge der offenen BA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**bearbeitenBA (BA)**
Ereignisbedingung	–
Zustandsbedingung	**BA.schritt = 1**
Zustandseffekt	**(offeneBA = offeneBA@pre excluding BA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, eingangslager, BA.maschine.ort)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, eingangslager, BA.maschine.ort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	**Ø**

Regel 110: Zeitereignis_bearbeitenBA.2

Bearbeitet der Manager einen BA und findet einen zu diesem BA passenden AA in den offenen AA, so generiert er aus den beiden Aufträgen einen TA und informiert alle angemeldeten FTF mittels *cfp*. Der BA und der passende AA sind nun nicht mehr in der Menge der offenen BA bzw. der offenen AA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**bearbeitenBA (BA)**
Ereignisbedingung	–
Zustandsbedingung	**offeneAA -> exists (AA | AA.schritt = BA.schritt – 1)**
Zustandseffekt	**(offeneBA = offeneBA@pre excluding BA) and (offeneAA = offeneAA@pre excluding AA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, AA.maschine.ort, BA.maschine.ort)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, AA.maschine.ort, BA.maschine.ort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	**Ø**

Regel 111: Zeitereignis_bearbeitenBA.3

Bearbeitet der Manager einen BA und findet einen zu diesem BA keinen passenden AA in den offenen AA, so versucht er, eine Maschine mit Arbeitsschritt *n-1* zu finden, die ein TS abgeben kann. An eine dieser Maschinen sendet er eine Anforderung zur Abgabe eines TS, alle übrigen werden in der Tabelle der möglichen Partner des BA gespeichert.

Ereignis	**bearbeitenBA (BA)**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**let m = (maschinen -> select (i | Maschine [i].schritt = BA.schritt – 1))** **BA.moeglPart = (m -> excluding j) and j = m.extract_min( )**
Folge-Zeitereignisse	Ø
Nachrichten	**{ anfordernAbgabeTS (BA) → Maschine [j] }**
Aktionen	Ø

Regel 112: Zeitereignis_bearbeitenBA.4

Erhält der Manager eine Bewerbung für einen TA, der noch ausgeschrieben ist, so fügt er diese seiner internen Bewerbungstabelle hinzu.

Ereignis	**Empfang(propose (TA, t) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneTA -> includes (TA) and TA.zust = ausgeschrieben**
Zustandseffekt	**TA.bewerbTab -> includes (i , t)**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 113: Nachricht_propose.1

Erhält der Manager eine Bewerbung für einen TA, der nicht mehr ausgeschrieben ist, weil die Bewerbungsfrist schon abgelaufen ist, so lehnt er die Bewerbung ab.

Ereignis	**Empfang(propose (TA, t) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	**{ reject_proposal (TA.id) → FTF [i] }**
Aktionen	Ø

Regel 114: Nachricht_propose.2

Bei einem Timeout zu einem TA, zu dem mindestens eine Bewerbung eingegangen ist, ermittelt der Manager das beste FTF und erteilt ihm den Auftrag.

Ereignis	**timeoutTA (TA)**
Ereignisbedingung	–
Zustandsbedingung	**TA.bewerbTab -> notEmpty()**
Zustandseffekt	**TA.zust = zuschlagErteilt and (TA.bewerbTab -> excludes j) and j = TA.bewerbTab@pre.extract_min( )**
Folge-Zeitereignisse	Ø
Nachrichten	**{ accept_proposal (TA.id, TA.startort, TA.zielort) → FTF [j] }**
Aktionen	Ø

Regel 115: Zeitereignis_timeoutTA.1

Bei einem Timeout zu einem TA, zu dem keine Bewerbung eingegangen ist, schreibt der TA-Manager den TA erneut aus.

Ereignis	**timeoutTA (TA)**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	**{ timeoutTA (TA) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (TA.id, TA.startort, TA.zielort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	Ø

Regel 116: Zeitereignis_timeoutTA.2

Als Zusicherung für den Empfang der Nachricht *accept_order* kann gegeben werden, dass der TA sich in den offenen TA befindet und den Zustand *zuschlagErteilt* hat.[101]

Ereignis	**Empfang(accept_order (TA) ← FTF [i])**
Zusicherung	**offeneTA -> includes (TA) and TA.zust = zuschlagErteilt**

Zusicherung Nachricht_accept_order

Erhält der Manager eine Auftragsannahme, so sendet allen anderen FTF, die sich beworben haben, eine Ablehnung. Der TA kann nun aus der Liste der offenen TA entfernt werden.

[101] Als weitere Zusicherung kann gegeben werden, dass der Sender der Nachricht das FTF ist, das den Zuschlag erhalten hat. Da sich der TA-Manager hier aus Gründen der Einfachheit nicht merkt, welchem FTF er den Zuschlag erteilt hat, kann diese Zusicherung nicht als Bedingung des Zustands des TA-Managers formuliert werden.

Ereignis	**Empfang(accept_order (TA) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**offeneTA -> excludes (TA)**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ reject_proposal (TA.id) → FTF [j] | j ∈ TA.bewerbTab }**
Aktionen	**Ø**

Regel 117: Nachricht_accept_order.1

Als Zusicherung für den Empfang der Nachricht *reject_order* kann gegeben werden, dass der TA sich in den offenen TA befindet und den Zustand *zuschlagErteilt* hat.[102]

Ereignis	**Empfang(reject_order (TA) ← FTF [i])**
Zusicherung	**offeneTA -> includes (TA) and TA.zust = zuschlagErteilt**

Zusicherung Nachricht_reject_order

Erhält der Manager eine Auftragsablehnung und enthält die Bewerbungstabelle noch mindestens eine Bewerbung, so ermittelt er das nächstbeste FTF und erteilt ihm den Auftrag.

Ereignis	**Empfang(reject_order (TA) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	**TA.bewerbTab -> notEmpty()**
Zustandseffekt	**(TA.bewerbTab -> excludes j) and j = TA.bewerbTab@pre.extract_min( )**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ accept_proposal (TA.id, TA.startort, TA.zielort) → FTF [j] }**
Aktionen	**Ø**

Regel 118: Nachricht_reject_order.1

Erhält der Manager eine Auftragsablehnung und enthält die Bewerbungstabelle keine Bewerbung mehr, so schreibt der TA-Manager den TA erneut aus. Der TA ist nun wieder ausgeschrieben.

[102] Als weitere Zusicherung kann auch hier gegeben werden, dass der Sender der Nachricht das FTF ist, das den Zuschlag erhalten hat. Da sich der TA-Manager aus Gründen der Einfachheit nicht merkt, welchem FTF er den Zuschlag erteilt hat, kann auch hier diese Zusicherung nicht als Bedingung des Zustands des TA-Managers formuliert werden.

Ereignis	**Empfang(reject_order (TA) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**TA.zust = ausgeschrieben**
Folge-Zeitereignisse	**{ timeoutTA (TA) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (TA.id, TA.startort, TA.zielort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	**Ø**

Regel 119: Nachricht_reject_order.2

Erhält der Manager eine Bestätigung für eine Annahmeanforderung und ist der entsprechende AA noch offen, so generiert er einen TA mit der sendenden Maschine als Zielort und informiert alle angemeldeten FTF mittels *cfp*. Der AA ist nun nicht mehr in der Menge der offenen AA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**Empfang(bestaetigenAnnahme (AA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneAA -> includes AA**
Zustandseffekt	**(offeneAA = offeneAA@pre excluding AA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, AA.maschine.ort, Maschine[i].ort)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, AA.maschine.ort, Maschine[i].ort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	**Ø**

Regel 120: Nachricht_bestaetigenAnnahme.1

Erhält der Manager eine Bestätigung für eine Annahmeanforderung und ist der entsprechende AA nicht mehr offen, weil inzwischen ein passender BA gefunden wurde, so storniert er die Annahmeanforderung und sendet der Maschine eine entsprechende Nachricht.

Ereignis	**Empfang(bestaetigenAnnahme (AA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ stornierenAnnahme (AA) → Maschine [i] }**
Aktionen	**Ø**

Regel 121: Nachricht_bestaetigenAnnahme.2

Erhält der Manager eine Ablehnung für eine Annahmeanforderung, ist der entsprechende AA noch offen und gibt es noch einen weiteren möglichen Partner, so sendet er an diesen eine Anforderung zur Annahme eines TS.

Ereignis	**Empfang(ablehnenAnnahme (AA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneAA -> includes AA and AA.moeglPart -> notEmpty()**
Zustandseffekt	**(AA.moeglPart -> excludes j) and j = AA.moeglPart@pre.extract_min( )**
Folge-Zeitereignisse	Ø
Nachrichten	**{ anfordernAnnahmeTS (AA) → Maschine [j] }**
Aktionen	Ø

Regel 122: Nachricht_ablehnenAnnahme.1

Erhält der Manager eine Ablehnung für eine Annahmeanforderung, ist der entsprechende AA noch offen und gibt es keinen weiteren möglichen Partner, so versucht er, den AA erneut zu bearbeiten.

Ereignis	**Empfang(ablehnenAnnahme (AA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneAA -> includes AA**
Zustandseffekt	–
Folge-Zeitereignisse	**{ bearbeitenAA (AA) @ c +** c_{aa_neu} **}**
Nachrichten	Ø
Aktionen	Ø

Regel 123: Nachricht_ablehnenAnnahme.2

Erhält der Manager eine Ablehnung für eine Annahmeanforderung und ist der entsprechende AA nicht mehr offen, weil inzwischen ein passender BA gefunden wurde, so braucht er keine weiteren Anforderungen stellen, da ja der AA nicht mehr offen ist. Er tut in diesem Fall nichts.

Ereignis	**Empfang(ablehnenAnnahme (AA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 124: Nachricht_ablehnenAnnahme.3

Erhält der Manager eine Bestätigung für eine Abgabeanforderung und ist der entsprechende BA noch offen, so generiert er einen TA mit der sendenden Maschine als Startort und informiert alle angemeldeten FTF mittels *cfp*. Der BA ist nun nicht mehr in der Menge der offenen BA. Folge-Zeitereignis ist das Timeout für den generierten TA.

Ereignis	**Empfang(bestaetigenAbgabe (BA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneBA -> includes BA**
Zustandseffekt	**(offeneBA = offeneBA@pre excluding BA) and (offeneTA = offeneTA@pre including TA (nextTA@pre, Maschine[i].ort, BA.maschine.ort)) and nextTA = nextTA@pre + 1**
Folge-Zeitereignisse	**{ timeoutTA (nextTA@pre) @ c +** c_{bewerb} **}**
Nachrichten	**{ cfp (nextTA@pre, Maschine[i].ort, BA.maschine.ort) → FTF [i] | i ∈ angemeldeteFTF }**
Aktionen	**Ø**

Regel 125: Nachricht_bestaetigenAbgabe.1

Erhält der Manager eine Bestätigung für eine Abgabeanforderung und ist der entsprechende BA nicht mehr offen, weil inzwischen ein passender AA gefunden wurde, so storniert er die Abgabeanforderung und sendet der Maschine eine entsprechende Nachricht.

Ereignis	**Empfang(bestaetigenAbgabe (BA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ stornierenAbgabe (BA) → Maschine [i] }**
Aktionen	**Ø**

Regel 126: Nachricht_bestaetigenAbgabe.2

Erhält der Manager eine Ablehnung für eine Abgabeanforderung, ist der entsprechende BA noch offen und gibt es noch einen weiteren möglichen Partner, so sendet er an diesen eine Anforderung zur Abgabe eines TS.

Ereignis	**Empfang(ablehnenAbgabe (BA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneBA -> includes BA and BA.moeglPart -> notEmpty()**
Zustandseffekt	**(BA.moeglPart -> excludes j) and j = BA.moeglPart@pre.extract_min( )**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ anfordernAbgabeTS (BA) → Maschine [j] }**
Aktionen	**Ø**

Regel 127: Nachricht_ablehnenAbgabe.1

Erhält der Manager eine Ablehnung für eine Abgabeanforderung, ist der entsprechende BA noch offen und gibt es keinen weiteren möglichen Partner, so versucht er, den BA erneut zu bearbeiten.

Ereignis	**Empfang(ablehnenAbgabe (BA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	**offeneBA -> includes BA**
Zustandseffekt	–
Folge-Zeitereignisse	**{ bearbeitenBA (BA) @ c +** c_{ba_neu} **}**
Nachrichten	Ø
Aktionen	Ø

Regel 128: Nachricht_ablehnenAbgabe.2

Erhält der Manager eine Ablehnung für eine Abgabeanforderung und ist der entsprechende BA nicht mehr offen, weil inzwischen ein passender AA gefunden wurde, so braucht er keine weiteren Anforderungen stellen, da ja der BA nicht mehr offen ist. Er tut in diesem Fall nichts.

Ereignis	**Empfang(ablehnenAbgabe (BA) ← Maschine [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 129: Nachricht_ablehnenAbgabe.3

B.6. Erweiterungen TA-Manager

Eine mögliche Erweiterung des Verhaltens des TA-Managers wäre eine Fehlerkontrolle. So könnte sich der TA-Manager in einer Tabelle speichern, welche TA im Zustand *unerledigt* sind. Ein TA wäre *unerledigt*, wenn der Zuschlag für ihn angenommen worden ist, aber er noch nicht ausgeführt worden ist. Nachdem das TS am Zielort abgelegt worden ist, müsste das FTF dies dem TA-Manager mittels einer Nachricht *meldenTAerledigt* melden, woraufhin der TA dann in den Zustand *erledigt* übergehen würde. Sollte für einen TA keine Erledigungsmeldung eingehen, so kann der TA-Manager beim FTF nachfragen und, falls dieses liegengeblieben ist oder sich nicht mehr meldet eine Fehlerbehandlung einleiten, wie z.B. ein Abschleppfahrzeug beauftragen.

Zur Fehlerkontrolle gehört auch, dass der Manager unzulässige Nachrichten der FTF überwacht bzw. abfängt, z.B., wenn ein FTF ein *accept_order* oder *reject_order* sendet, ohne vorher für den genannten TA ein *accept_proposal* erhalten zu haben.

Ebenfalls zur Fehlerkontrolle gehört, dass ein TA-Manager einem FTF den Zuschlag wieder entziehen kann, wenn dieses nicht innerhalb einer angemessenen Frist auf ein *accept_proposal* antwortet.

Dynamisches An- und Abmelden von FTF könnte als denkbare Erweiterung mit aufgenommen werden.

Ein weitere Erweiterung wäre, dass der TA-Manager den FTF beim *cfp* mitteilt, wann denn die Bewerbungsfrist für den TA endet. Dies würde allerdings bedingen, dass es eine globale Zeit gibt, was zwar in der Simulation vorhanden ist, aber im zu simulierenden Realsystem nicht ohne weiteres vorausgesetzt werden darf. Dann würde es sich auch anbieten, dass der TA-Manager dem FTF bei einem *reject-order* mitteilt, ob die Bewerbung abgelehnt wurde, weil ein anderes FTF eine bessere Bewerbung abgegeben hat oder, weil die Bewerbung erst nach Ablauf der Bewerbungsfrist beim TA-Manager eingetroffen ist.

Im Fall einer verspätet eingetroffenen Bewerbung wäre es auch denkbar, dass der TA-Manager diese Bewerbung nicht ablehnt, falls innerhalb der Bewerbungsfrist keine Bewerbung eingetroffen ist. In diesem Fall würde der TA-Manager dem FTF mit der verspäteten Bewerbung den Zuschlag mittels *accept-proposal* erteilen.

Im Fall, dass ein TA nicht vergeben werden konnte, weil sich kein FTF für den TA beworben hat oder, weil alle FTF, die sich beworben haben, mittels *reject-order* den Zuschlag abgelehnt haben, schreibt der TA-Manager den TA erneut aus. Als Erweiterung wäre denkbar, dass der TA bei einer erneuten Ausschreibung eine erhöhte Priorität bekommt, so dass der Anreiz für die FTF, sich für diesen TA zu bewerben größer wird. Eine erhöhte Priorität hat hingegen bei AA und BA, die erneut bearbeitet werden müssen, keinen Sinn, da ja die Maschinen bei jedem freien Platz im Eingangspuffer bzw. bei jedem vorhandenen TS im Ausgangspuffer die Anforderung erfüllen und sie nur dann ablehnen, wenn sie nicht erfüllbar ist, also kein Platz im Eingangspuffer mehr frei ist bzw. kein TS sich im Ausgangspuffer befindet. Dann können sie aber auch bei erhöhter Priorität die Anforderung nicht erfüllen.

Die Maschinen, die als möglicher Partner für einen AA bzw. BA in Frage kommen, sind in der Prioritätswarteschlange *moeglPart* eingeordnet. Anstatt aus Gründen der Einfachheit für alle Maschinen die gleiche Priorität zu verwenden, käme als Priorität der Abstand der Maschine zum Startort (im Falle eines AA) bzw. Zielort (im Falle eines BA) in Frage. Weiter denkbar wäre, den aktuellen Stand des Eingangs- bzw. Ausgangspuffers zu berücksichtigen. Dazu müssten die Maschinen allerdings dem TA-Manager diesen Stand melden.

Damit Maschinen eine größere Planungssicherheit haben, könnte der TA-Manager, der eine Aufnahme- bzw. Abgabeanforderung an Maschinen mit der Nachricht *stornierenAufnahme* bzw. *stornierenAbgabe* stornieren kann, in dem Moment, in dem feststeht, dass die Aufnahme bzw. Abgabe nicht mehr storniert wird, eine entsprechende Bestätigungsnachricht senden.

Sollten sich im System mehrere Eingangs- und Ausgangslager befinden, so kann der TA-Manager aus den BA einer Maschine mit Arbeitsschritt 1 sowie den AA einer Maschine mit Arbeitsschritt n jeweils einen TA mit dem am besten geeigneten Eingangslager als Startort bzw. Ausgangslager als Zielort generieren.

B.7. A-Regeln FTF

Das FTF bewirbt sich um jeden ausgeschriebenen TA, wenn es gerade frei ist.

Ereignis	**Empfang(cfp (TA, startort, zielort) ← TAManager [i])**
Ereignisbedingung	**-**
Zustandsbedingung	**aktuellerTA -> isEmpty()**
Zustandseffekt	**-**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ propose (TA, c + Zeitberechnung (ort, startort)) → TAManager [i] }**
Aktionen	**Ø**

Regel 130: Nachricht_cfp.1

Wenn es jedoch gerade nicht frei ist, bewirbt es sich nicht.

Ereignis	**Empfang(cfp (TA, startort, zielort) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	∅
Nachrichten	∅
Aktionen	∅

Regel 131: Nachricht_cfp.2

Empfängt das FTF eine Nachricht *accept_proposal*,[103] so nimmt es den TA an, wenn es gerade noch frei ist. Es übernimmt den angenommenen TA als aktuellen TA, berechnet den Weg zum Startort und beginnt, ihn abzufahren. An den TA-Manager geht die Nachricht *accept_order*, um auszudrücken, dass der TA angenommen wurde.

Ereignis	**Empfang(accept_proposal (TA, startort, zielort) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	**aktuellerTA -> isEmpty()**
Zustandseffekt	**aktuellerTA = TA (TA, startort, zielort) and weg = Wegberechnung (ort, startort)**
Folge-Zeitereignisse	**{ abfahrenWeg ( ) @ c }**
Nachrichten	**{ accept_order (TA) → TAManager [i] }**
Aktionen	∅

Regel 132: Nachricht_accept_proposal.1

Empfängt das FTF eine Nachricht *accept_proposal* und ist es gerade nicht frei, weil es bereits einen anderen TA angenommen hat, so lehnt es den Zuschlag durch die Nachricht *reject_order* an den TA-Manager ab.

Ereignis	**Empfang(accept_proposal (TA, startort, zielort) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	∅
Nachrichten	**{ reject_order (TA) → TAManager [i] }**
Aktionen	∅

Regel 133: Nachricht_accept_proposal.2

[103] Als Zusicherung kann gegeben werden, dass sich das FTF vorher mittels *propose* beim TA-Manager für diesen TA beworben hat. Da sich das FTF die gesendeten Nachrichten nicht speichert, kann diese Zusicherung allerdings nicht in Form einer Zustandsbedingung ausgedrückt werden.

Empfängt das FTF eine Nachricht *reject_proposal*,[104] so braucht es hier gar nicht darauf zu reagieren.

Ereignis	**Empfang(reject_proposal (TA) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 134: Nachricht_reject_proposal.1

Als Zusicherung für das Auftreten des internen Zeitereignisses *abfahrenWeg* kann gegeben werden, dass das FTF zu diesem Zeitpunkt steht und einen TA übernommen hat. Außerdem ist der Weg nur dann leer, wenn das FTF sich am Startort (wenn der TA im Zustand *startortAnvisiert* ist), am Ende eines Segments, das am Startort endet (ebenfalls wenn der TA im Zustand *startortAnvisiert* ist) oder am Ende eines Segments, das am Zielort endet (wenn der TA im Zustand *zielortAnvisiert* ist) befindet.[105] Ist der Weg nicht leer, so steht das FTF entweder auf dem Knoten, an dem das erste Segment des Weges beginnt oder am Ende eines Segments, das an diesem Knoten endet.

Ereignis	**abfahrenWeg ( )**
Zusicherung	**bewZust = stehend and** **aktuellerTA -> notEmpty() and** **(weg.length = 0 implies** **(aktuellerTA.zust = startortAnvisiert and** **((ort.typ = knoten and aktuellerTA.startort = ort) or** **(ort.typ = segment and aktuellerTA.startort = ort.zielknoten)))** **or (aktuellerTA.zust = zielortAnvisiert and** **(ort.typ = segment and aktuellerTA.zielort = ort.zielknoten))** **and** **(weg.length > 0 implies** **((ort.typ = knoten and weg.head.startknoten = ort) or** **(ort.typ = segment and weg.head.startknoten = ort.zielknoten)))**

Zusicherung Zeitereignis_abfahrenWeg

Möchte das FTF einen nichtleeren Weg abfahren und befindet sich gerade auf einem Knoten, dann fährt es das erste Segment des Weges ab, führt also die Aktion *abfahrenSegment* mit dem ersten Element des Weges als Argument aus. Der Weg wird jedoch noch nicht um dieses Element verkürzt, da dies bei der Ankunft am Ende des Segments geschieht.

[104] Als Zusicherung kann auch hier gegeben werden, dass sich das FTF vorher mittels *propose* beim TA-Manager für diesen TA beworben hat. Da sich das FTF die gesendeten Nachrichten nicht speichert, kann diese Zusicherung ebenfalls nicht in Form einer Zustandsbedingung ausgedrückt werden.

[105] Das FTF kann sich nur dann direkt am Startort befinden, wenn es von vorneherein einen leeren Weg geplant hat, weil es sich zum Zeitpunkt der Auftragserteilung bereits am Startort des TA befunden hat. Direkt am Zielort kann es sich nicht befinden, weil Startort und Zielort eines TA immer verschieden sind.

Ereignis	**abfahrenWeg ( )**
Ereignisbedingung	–
Zustandsbedingung	**weg.length > 0 and ort.typ = knoten**
Zustandseffekt	**bewZust = fahrend**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	**{ abfahrenSegment (ort, weg@pre.head) }**

Regel 135: Zeitereignis_abfahrenWeg.1

Möchte das FTF einen leeren Weg abfahren und befindet sich gerade auf einem Knoten, dann ist nach Zusicherung sichergestellt, dass es sich am Startort des aktuellen TA befindet. Es nimmt ein TS auf, wodurch es im Beladungszustand *beladen* ist. Ferner plant es den Weg zum Zielort und beginnt, den geplanten Weg abzufahren. Der aktuelle TA ist nun im Zustand *zielortAnvisiert*.

Ereignis	**abfahrenWeg ( )**
Ereignisbedingung	–
Zustandsbedingung	**ort.typ = knoten**
Zustandseffekt	**belZust = beladen and aktuellerTA.zust = zielortAnvisiert and weg = Wegberechnung (ort, aktuellerTA.zielort)**
Folge-Zeitereignisse	**{ abfahrenWeg ( ) @ c }**
Nachrichten	Ø
Aktionen	**{ aufnehmenTS ( ) }**

Regel 136: Zeitereignis_abfahrenWeg.2

Möchte das FTF einen Weg abfahren und befindet sich gerade am Ende eines Segments, dann muss es eine Erlaubnisanfrage für das Einfahren in den Knoten stellen.

Ereignis	**abfahrenWeg ( )**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	**{ Erlaubnisanfrage ( ) → Verkehrsregler [ort.zielknoten.verkehrsregler] }**
Aktionen	Ø

Regel 137: Zeitereignis_abfahrenWeg.3

Als Zusicherung für den Empfang der Nachricht *Erlaubnis* kann gegeben werden, dass das FTF am Ende eines Segments steht, dessen Zielkonten den Sender der Nachricht als Verkehrsregler hat. Ist der Weg leer, so befindet sich das FTF am Ende eines Segments, das am Startort endet (wenn der TA im Zustand *startortAnvisiert* ist) oder am Ende eines Segments, das am Zielort endet (wenn der TA im Zustand *zielortAnvisiert* ist). Ist der Weg nicht leer, so steht das FTF am Ende eines Segments, das an dem Knoten endet, an dem das erste Segment des Weges beginnt.

Ereignis	**Empfang(Erlaubnis ( ) ← Verkehrsregler [i])**
Zusicherung	**bewZust = stehend and ort.typ=segment and ort.zielknoten.verkehrsregler = i and (weg.length = 0 implies (aktuellerTA.zust = startortAnvisiert and aktuellerTA.startort = ort.zielknoten) or (aktuellerTA.zust = zielortAnvisiert and aktuellerTA.zielort = ort.zielknoten)) and (weg.length > 0 implies weg.head.startknoten = ort.zielknoten)**

Zusicherung Nachricht_Erlaubnis

Erhält das FTF eine Erlaubnis zur Einfahrt in einen Knoten und möchte einen nichtleeren Weg abfahren, dann führt es die Aktion *abfahrenKnotenUndSegment* mit dem Knoten, an dem es sich befindet und dem ersten Element des Weges aus.

Ereignis	**Empfang(Erlaubnis ( ) ← Verkehrsregler [i])**
Ereignisbedingung	–
Zustandsbedingung	**weg.length > 0**
Zustandseffekt	**bewZust = fahrend and ort = ort@pre.zielknoten**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	**{ abfahrenKnotenUndSegment (ort, weg.head) }**

Regel 138: Nachricht_Erlaubnis.1

Erhält das FTF eine Erlaubnis zur Einfahrt in einen Knoten und möchte einen leeren Weg abfahren, dann führt es die Aktion *befahrenKnoten* mit dem Knoten, an dem es sich befindet aus.

Ereignis	**Empfang(Erlaubnis ( ) ← Verkehrsregler [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**bewZust = fahrend and ort = ort@pre.zielknoten**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	**{ befahrenKnoten (ort) }**

Regel 139: Nachricht_Erlaubnis.2

Als Zusicherung für die Wahrnehmung *KnotenPassiert* kann gegeben werden, dass das FTF sich fahrend auf dem Knoten befunden hat, den es nun passiert hat.

Ereignis	**KnotenPassiert (k, s)**
Zusicherung	**ort = k and bewZust = fahrend**

Zusicherung Wahrnehmung_KnotenPassiert

Nimmt das FTF wahr, dass es einen Knoten passiert hat, so aktualisiert es seinen Ort und gibt den Knoten durch eine Nachricht an den Verkehrsregler wieder frei.

Ereignis	`KnotenPassiert ( k, s )`
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	`ort = s`
Folge-Zeitereignisse	∅
Nachrichten	`{ Knotenfreigabe ( ) → Verkehrsregler [ k.verkehrsregler ] }`
Aktionen	∅

Regel 140: Wahrnehmung_KnotenPassiert.1

Als Zusicherung für die Wahrnehmung *KnotenErreicht* kann gegeben werden, dass das FTF sich fahrend auf dem Knoten befunden hat, den es nun erreicht hat. Der Weg hat die Länge 0 und das FTF befindet sich am Startort des aktuellen TA (wenn der TA im Zustand *startortAnvisiert* ist) oder am Zielort des aktuellen TA (wenn der TA im Zustand *zielortAnvisiert* ist).

Ereignis	`KnotenErreicht ( k )`
Zusicherung	`ort = k and bewZust = fahrend and weg.length = 0 and ( ( aktuellerTA.zust = startortAnvisiert and aktuellerTA.startort = ort ) or ( aktuellerTA.zust = zielortAnvisiert and aktuellerTA.zielort = ort ) )`

Zusicherung Wahrnehmung_KnotenErreicht

Hat das FTF einen Knoten erreicht und befindet sich am Startort des aktuellen TA, so nimmt es ein TS auf, wodurch es im Beladungszustand *beladen* ist. Da das FTF nun zunächst steht, ist es im Bewegungszustand *stehend*. Ferner plant es den Weg zum Zielort und beginnt, den geplanten Weg abzufahren. Der aktuelle TA ist nun im Zustand *zielortAnvisiert*.

Ereignis	`KnotenErreicht ( k )`
Ereignisbedingung	–
Zustandsbedingung	`aktuellerTA.zust = startortAnvisiert`
Zustandseffekt	`bewZust = stehend and belZust = beladen and aktuellerTA.zust = zielortAnvisiert and weg = Wegberechnung ( ort, aktuellerTA.zielort )`
Folge-Zeitereignisse	`{ abfahrenWeg ( ) @ c }`
Nachrichten	∅
Aktionen	`{ aufnehmenTS ( ) }`

Regel 141: Wahrnehmung_KnotenErreicht.1

Hat das FTF einen Knoten erreicht und befindet sich am Zielort des aktuellen TA, so legt es das TS ab, wodurch es im Beladungszustand *unbeladen* ist. Da das FTF nun steht, ist es im Bewegungszustand *stehend*. Der aktuelle TA ist somit erfüllt, so dass das FTF nun keinen aktuellen TA mehr zu bearbeiten hat und wieder frei ist. Einen geplanten Weg zum Abfahren hat das FTF nun auch nicht mehr.

Ereignis	`KnotenErreicht ( k )`
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	`bewZust = stehend and belZust = unbeladen and weg -> isEmpty() and aktuellerTA -> isEmpty()`
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	`{ ablegenTS ( ) }`

Regel 142: Wahrnehmung_KnotenErreicht.2

Als Zusicherung für die Wahrnehmung *SegmentendeErreicht* kann gegeben werden, dass das FTF sich fahrend auf dem Segment befunden hat, dessen Ende es nun erreicht hat. Das erste Element des Weges ist gerade dieses Segment.

Ereignis	`SegmentendeErreicht ( s )`
Zusicherung	`ort = s and bewZust = fahrend and weg.head = s`

Zusicherung Wahrnehmung_SegmentendeErreicht

Nimmt das FTF wahr, dass es ein Segmentende erreicht hat, so verkürzt es den Weg um gerade dieses Segment. Den so verkürzten Weg beginnt es, abzufahren. Da das FTF nun zunächst steht, ist es im Bewegungszustand *stehend*.

Ereignis	`SegmentendeErreicht ( s )`
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	`bewZust = stehend and weg = weg@pre.tail`
Folge-Zeitereignisse	`{ abfahrenWeg ( ) @ c }`
Nachrichten	Ø
Aktionen	Ø

Regel 143: Wahrnehmung_SegmentendeErreicht.1

B.8. Erweiterungen FTF

Eine mögliche Erweiterung des Verhaltens des FTF wäre, dass es das aktuelle Verkehrsgeschehen (z.B. Staus) sowohl bei der Wegeplanung als auch bei der Zeitberechnung berücksichtigt. Als Grundlage für eine Bewerbung könnte dann nicht nur die Zeit gewählt werden, sondern eine beliebige Gewichtung von Faktoren wie geringe Fahrzeit (schnelle Auftragserledigung), kleine Strecke (geringe Fahrzeugabnutzung) und wenig Energieverbrauch (Treibstoffkosten bzw. Reichweite bei Batteriebetrieb). So kann je nach Gewichtung eine schnelle, lange Strecke einer langsamen, kurzen vorgezogen werden und umgekehrt. Wenn das aktuelle Verkehrsgeschehen berücksichtigt wird, ist es sinnvoll, nicht mehr zwangsläufig den am Anfang geplanten Weg abzufahren. Ein FTF könnte nämlich dann dynamisch den geplanten Weg ändern, um einen Stau zu umfahren. Ein FTF wäre dann damit auch in der Lage, auf dynamische Änderungen des Fahrkurses, wie z.B. den Ausfall eines Segments oder eines Knotens zu reagieren.

Ein FTF könnte sich auch bewerben, wenn es gerade nicht frei ist, z.B. wenn es nicht mehr lange braucht, um den aktuellen TA zu erledigen und sich danach in der Nähe des Startorts des ausgeschriebenen TA befindet. Umgekehrt muss sich ein FTF nicht immer, wenn es gerade frei ist, für einen TA bewerben. Sich nicht zu bewerben käme z.B. in Frage, wenn ein FTF sich weit entfernt vom

Startort des ausgeschriebenen TA befindet. Ein FTF müsste außerdem einen erhalteneren Zuschlag nicht unbedingt mittels *reject-order* ablehnen, wenn es gerade nicht frei ist, sondern könnte ihn auch annehmen, wenn es bald frei wird und die Zeit, die es bei der Bewerbung angegeben hat, trotzdem noch einhalten kann.

Wenn man im Szenario nicht nur Single-Load-Carrier, sondern auch Multiple-Load-Carrier zulässt, ergeben sich zahlreiche weitere Optionen hinsichtlich der Bewerbung für TA und der Wegeplanung. So wäre es nötig, einen optimalen Weg zu ermitteln, der alle Startorte und alle Zielorte aller übernommenen TA beinhaltet, wobei jeweils der Startort vor dem Zielort angefahren wird und bei dem zu keinem Zeitpunkt mehr TS transportiert werden müssten als die Kapazität des FTF ermöglicht.

Um eine schnellere Auftragsvergabe zu ermöglichen, könnte ein FTF an den TA-Manager eine Nachricht *refuse* schicken, wenn es sich entschließt, sich nicht für einen TA zu bewerben. Der Manager müsste dann nicht in allen Fällen den Timeout des TA abwarten, sondern könnte den Auftrag bereits vergeben, nachdem von allen bei ihm angemeldeten FTF entweder ein *propose* oder ein *refuse* angekommen ist.

Ist ein FTF frei, dann muss es nicht notwendig am Zielort des zuletzt ausgeführten TA stehen bleiben, sondern kann eine strategisch günstige Position einnehmen, von der aus es schneller zum Startort von zukünftigen TA, die es möglicherweise erhalten wird, kommt. Insbesondere kann es, wenn es sich für genau einen TA beworben hat, zum Startort dieses TA fahren. Dies gilt vor allem, wenn erfahrungsgemäß die Aussichten, diesen TA zu erhalten groß sind oder sich der Startort des TA in einer zentralen Lage befindet. Dies setzt voraus, dass ein FTF sich merkt, für welche FTF es sich beworben hat. In diesem Fall müsste es dann auch die Nachricht *reject_proposal* beachten.

Realistischer wäre die Modellierung der FTF, wenn man nicht davon abstrahiert, dass das FTF auf seinem Weg vor Kurven abbremsen, Kurven fahren, nach Kurven wieder beschleunigen und ab Ende des Segments anhalten muss, sondern hierfür Aktionen wie *abbremsen*, *fahrenKurve* oder *beschleunigen* modelliert.

Eine weitere Erweiterung wäre, dass sich FTF dynamisch bei TA-Managern an- und abmelden können.

B.9. A-Regeln Verkehrsregler

Erhält ein Verkehrsregler die Erlaubnisanfrage für einen Knoten und ist der Knoten frei, so erteilt er die Erlaubnis. Der Knoten ist nun nicht mehr frei.

Ereignis	**Empfang(Erlaubnisanfrage ( ) ← FTF [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**zust = frei**
Zustandseffekt	**zust = belegt**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ Erlaubnis ( ) → FTF [i] }**
Aktionen	**Ø**

Regel 144: Nachricht_Erlaubnisanfrage.1

Erhält ein Verkehrsregler die Erlaubnisanfrage für einen Knoten und ist der Knoten belegt, so wird das FTF in die Menge der auf die Freigabe dieses Knotens wartenden FTF eingereiht.

Ereignis	**Empfang(Erlaubnisanfrage ( ) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**wartendeFTF = wartendeFTF@pre.enqueue (i)**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 145: Nachricht_Erlaubnisanfrage.2

Als Zusicherung für den Empfang der Nachricht *Knotenfreigabe* kann gegeben werden, dass der Knoten zu diesem Zeitpunkt belegt war.

Ereignis	**Empfang(Knotenfreigabe ( ) ← FTF [i])**
Zusicherung	**zust = belegt**

Zusicherung Nachricht_Knotenfreigabe

Erhält ein Verkehrsregler die Nachricht, dass ein FTF den Knoten passiert hat, so wird einem wartenden FTF die Freigabe erteilt, falls ein solches vorhanden ist.

Ereignis	**Empfang(Knotenfreigabe ( ) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	**wartendeFTF.length > 0**
Zustandseffekt	**j = wartendeFTF@pre.dequeue ( )**
Folge-Zeitereignisse	Ø
Nachrichten	**{ Erlaubnis ( ) → FTF [j] }**
Aktionen	Ø

Regel 146: Nachricht_Knotenfreigabe.1

Erhält ein Verkehrsregler die Nachricht, dass ein FTF den Knoten passiert hat und ist kein wartendes FTF vorhanden, so wird der Knoten wieder als frei markiert.

Ereignis	**Empfang(Knotenfreigabe ( ) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**zust = frei**
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 147: Nachricht_Knotenfreigabe.2

B.10. Erweiterungen Verkehrsregler

Als Erweiterung wäre denkbar, dass ein Verkehrsregler mehrere Knoten verwaltet. In diesem Fall müssten alle Nachrichten noch den Knoten, den sie betreffen als Parameter mitführen und der

Verkehrsregler müsste für jeden Knoten einen eigenen Zustand und eine eigene Warteschlange besitzen.

Eine weitere Erweiterung des Verhaltens des Verkehrsregler wäre, FTF, die auf einem Knoten stehen geblieben sind, weil sie ihren TA erfüllt zu haben, „wegzuschicken“, also aufzufordern, den Knoten zu verlassen, wenn sie andere FTF an der Einfahrt in den Knoten hindern. Dies könnte durch ein Timeout realisiert werden, nach dem ein FTF einen Knoten verlassen haben muss. Das Timeout sollte allerdings nur dann ausgelöst werden, wenn tatsächlich andere FTF in den blockierten Knoten einfahren wollen.

Verkehrsregler könnten auch bidirektionale Segmente verwalten, wenn diese im Szenario zugelassen sind. Die Steuerung wird dadurch allerdings kompliziert, da ein FTF sowohl die Freigabe für einen Knoten als auch für das darauf folgende bidirektionale Segment braucht. Wenn allerdings immer erst die Freigabe für den Knoten und erst dann für das Segment angefordert wird, kann es zumindest zu keiner Verklemmung kommen.

B.11. Variante Austausch von Transportaufträgen

In Abschnitt 8.8.3 wurde eine Variante vorgestellt, in der FTF untereinander Transportaufträge tauschen können. Dieser Abschnitt enthält die Änderungen des Grundszenarios, die für diese Variante erforderlich sind.

In dieser Variante fragt jedes FTF, das einen Auftrag erteilt bekommen hat, bei den anderen FTF nach, ob sie ihrerseits gerade einen Transportauftrag ausführen und, wenn ja, ob sich die Gesamtbearbeitungszeit beider Aufträge verbessern würde, wenn die beiden FTF ihre Aufträge tauschen würden. Ist dies der Fall, so tauscht das FTF seinen Auftrag mit dem FTF, bei dem die größte Ersparnis erzielbar ist.

Es gibt zwischen FTF vier verschiedene Nachrichtentypen, die allesamt den Austausch von Transportaufträgen betreffen. Der Nachrichtentyp *optimierenTA* stellt eine Aufforderung zum Optimieren der Gesamtbearbeitungszeit dar, indem ein FTF den anderen seinen aktuellen Standort und den Startort des TA, für den es gerade den Zuschlag erhalten hat, mitteilt. Der Nachrichtentyp *tauschvorschlagTA* stellt einen konkreten Vorschlag für einen Tausch von TA dar, wobei das vorschlagende FTF dem anderen neben seinem aktuellen TA mitteilt, um wie viel Zeiteinheiten die Gesamtbearbeitungszeit beider Aufträge gesenkt werden kann. Mittels *annehmenTauschTA* bzw. *ablehnenTauschTA* wird dieser Vorschlag angenommen bzw. abgelehnt, wobei im Falle der Annahme das annehmende FTF dem anderen ebenfalls seinen aktuellen TA mitteilt.

Sender	Empfänger
FTF	**FTF**
Nachricht	Nachrichtenkanal
optimierenTA (standort, startort)	**elektronische Nachricht**
tauschvorschlagTA (ersparnis, TA, startort, zielort)	**elektronische Nachricht**
annehmenTauschTA (TA, startort, zielort)	**elektronische Nachricht**
ablehnenTauschTA ( )	**elektronische Nachricht**

Zum internen Zustand eines FTF gehören in dieser Variante zusätzlich das Attribut *tauschVorgeschlagen*, in dem das FTF sich merkt, ob es gerade einen Tausch vorgeschlagen hat. Dies kann durch eine Nachricht *optimierenTA* an alle anderen FTF oder durch eine Nachricht *tauschvorschlagTA* an ein FTF geschehen sein. Des Weiteren beinhaltet der interne Zustand des FTF eine Tabelle *tauschTab*, in der es sich erhaltene Tauschvorschläge von anderen FTF merkt.

Es gibt für FTF einen neuen Typ von internen Zeitereignissen: *timeoutTauschTA*. Bei ihrem Auftreten ermittelt das FTF den bestmöglichen Tausch von TA und führt ihn durch. Der Timeout erfolgt c_{tausch} Zeiteinheiten nach dem Tauschvorschlag.

Als Zusicherung für den Empfang der Nachricht *optimierenTA* kann gegeben werden, dass, wenn das FTF auf einem Knoten steht und einen aktuellen TA hat, es einen Tauschvorschlag gemacht hat.[106]

Ereignis	`Empfang( optimierenTA ( standort, startort ) ← FTF [ i ] )`
Zusicherung	`( ort.typ = knoten and bewZust = stehend and aktuellerTA -> notEmpty()) implies tauschVorgeschlagen`

Zusicherung Nachricht_optimierenTA

Erhält ein FTF die Nachricht *optimierenTA* und hat es bereits einen Tauschvorschlag gemacht, führt es gerade keinen TA aus oder ist der aktuelle TA bereits im Zustand *zielortAnvisiert*, so kann es dem Sender der Nachricht keinen Tauschvorschlag unterbreiten.

Ereignis	`Empfang( optimierenTA ( standort, startort ) ← FTF [ i ] )`
Ereignisbedingung	–
Zustandsbedingung	`tauschVorgeschlagen or aktuellerTA -> isEmpty() or aktuellerTA.zust = zielortAnvisiert`
Zustandseffekt	–
Folge-Zeitereignisse	Ø
Nachrichten	Ø
Aktionen	Ø

Regel 148: Nachricht_optimierenTA.1

Erhält ein FTF die Nachricht *optimierenTA*, befindet es sich gerade auf einem Segment und lässt sich eine Ersparnis erzielen, so macht das FTF einen Tauschvorschlag.

Ereignis	`Empfang( optimierenTA ( standort, startort ) ← FTF [ i ] )`
Ereignisbedingung	–

Zustandsbedingung	`let ersparnis = Zeitberechnung ( ort.zielknoten , aktuellerTA.startort ) + Zeitberechnung ( standort , startort ) - Zeitberechnung ( ort.zielknoten , startort ) - Zeitberechnung ( standort, aktuellerTA.startort )`
	
	`ort.typ = segment and ersparnis > 0`

Zustandseffekt	`tauschVorgeschlagen`
Folge-Zeitereignisse	Ø
Nachrichten	`{ tauschvorschlagTA ( ersparnis, aktuellerTA.id, aktuellerTA.startort, aktuellerTA.zielort ) → FTF [ i ] }`
Aktionen	Ø

Regel 149: Nachricht_optimierenTA.2

[106] Als weitere Zusicherung kann gegeben werden, dass dieser Tauschvorschlag in Form einer Nachricht *optimierenTA* an alle anderen FTF (und nicht in Form einer Nachricht *tauschvorschlagTA* an ein anderes FTF) erfolgt ist. Da sich das FTF hier aus Gründen der Einfachheit nicht merkt, welche Nachrichten es verschickt hat, kann diese Zusicherung nicht als Bedingung des Zustands des FTF formuliert werden.

Erhält ein FTF die Nachricht *optimierenTA*, befindet es sich gerade fahrend auf einem Knoten, der nicht bereits der Startknoten des aktuellen TA ist und lässt sich eine Ersparnis erzielen, so macht das FTF ebenfalls einen Tauschvorschlag.

Ereignis	**Empfang(optimierenTA (standort, startort) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	**let ersparnis =** **Zeitberechnung (weg.head.zielknoten , aktuellerTA.startort)** **+ Zeitberechnung (standort , startort)** **– Zeitberechnung (weg.head.zielknoten , startort)** **– Zeitberechnung (standort, aktuellerTA.startort)** **ort.typ = knoten and bewZust = fahrend and** **ort <> aktuellerTA.startort and ersparnis > 0**
Zustandseffekt	**tauschVorgeschlagen**
Folge-Zeitereignisse	**Ø**
Nachrichten	**{ tauschvorschlagTA (ersparnis, aktuellerTA.id,** **aktuellerTA.startort, aktuellerTA.zielort) → FTF [i] }**
Aktionen	**Ø**

Regel 150: Nachricht_optimierenTA.3

Ansonsten kann das FTF dem Sender der Nachricht keinen Tauschvorschlag unterbreiten.

Ereignis	**Empfang(optimierenTA (standort, startort) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	–
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 151: Nachricht_optimierenTA.4

Als Zusicherung für den Empfang der Nachricht *tauschvorschlagTA* kann gegeben werden, dass das FTF auf einem Knoten steht, einen aktuellen TA hat und einen Tauschvorschlag gemacht hat.[107]

Ereignis	**Empfang(tauschvorschlagTA (ersparnis, TA, startort, zielort)** **← FTF [i])**
Zusicherung	**ort.typ = knoten and bewZust = stehend and** **aktuellerTA -> notEmpty() and tauschVorgeschlagen**

Zusicherung Nachricht_tauschvorschlagTA

Erhält ein FTF die Nachricht *tauschvorschlagTA*, so fügt es den Tauschvorschlag seiner Tauschtabelle hinzu.

[107] Als weitere Zusicherung kann auch hier gegeben werden, dass dieser Tauschvorschlag in Form einer Nachricht *optimierenTA* an alle anderen FTF (und nicht in Form einer Nachricht *tauschvorschlagTA* an ein anderes FTF) erfolgt ist. Da sich das FTF hier aus Gründen der Einfachheit nicht merkt, welche Nachrichten es verschickt hat, kann auch hier diese Zusicherung nicht als Bedingung des Zustands des FTF formuliert werden.

Ereignis	**Empfang(tauschvorschlagTA (ersparnis, TA, startort, zielort) ← FTF [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**–**
Zustandseffekt	**tauschTab -> includes (i, ersparnis, TA, startort, zielort)**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 152: Nachricht_ tauschvorschlagTA.1

Als Zusicherung für den Empfang der Nachricht *annehmenTauschTA* kann gegeben werden, dass das FTF einen aktuellen TA hat und einen Tauschvorschlag gemacht hat.[108]

Ereignis	**Empfang(annehmenTauschTA (TA, startort, zielort) ← FTF [i])**
Zusicherung	**aktuellerTA -> notEmpty() and tauschVorgeschlagen**

Zusicherung Nachricht_annehmenTauschTA

Erhält ein FTF die Nachricht *annehmenTauschTA*, so aktualisiert es in allen Fällen seinen aktuellen TA und ändert seinen Zustand dahingehend, dass es keinen Tauschvorschlag mehr gemacht hat. Steht das FTF gerade am Ende eines Segments, so plant es seinen Weg ab dem Zielknoten des Segments, auf dem es steht, neu.

Ereignis	**Empfang(annehmenTauschTA (TA, startort, zielort) ← FTF [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**ort.typ = segment and bewZust = stehend**
Zustandseffekt	**aktuellerTA = TA (TA, startort, zielort) and (not tauschVorgeschlagen) and weg = Wegberechnung (ort.zielknoten, startort)**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 153: Nachricht_ annehmenTauschTA.1

Fährt das FTF gerade auf einem Segment, so plant es seinen Weg ebenfalls ab dem Zielknoten des Segments, auf dem es steht, neu. Jedoch verbleibt in diesem Fall das aktuelle Segment noch am Anfang des geplanten Weges.

[108] Als weitere Zusicherung kann gegeben werden, dass dieser Tauschvorschlag in Form einer Nachricht *tauschvorschlagTA* an ein anderes FTF (und nicht in Form einer Nachricht *optimierenTA* an alle anderen FTF) erfolgt ist. Da sich das FTF aus Gründen der Einfachheit nicht merkt, welche Nachrichten es verschickt hat, kann auch diese Zusicherung nicht als Bedingung des Zustands des FTF formuliert werden.

Ereignis	**Empfang(annehmenTauschTA (TA, startort, zielort) ← FTF [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**ort.typ = segment**
Zustandseffekt	**aktuellerTA = TA (TA, startort, zielort) and (not tauschVorgeschlagen) and weg.tail = Wegberechnung (ort.zielknoten, startort)**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 154: Nachricht_ annehmenTauschTA.2

Befindet sich das FTF gerade auf einem Knoten, so plant es seinen Weg ab dem Zielknoten des ersten Segments seines geplanten Weges neu.

Ereignis	**Empfang(annehmenTauschTA (TA, startort, zielort) ← FTF [i])**
Ereignisbedingung	**–**
Zustandsbedingung	**–**
Zustandseffekt	**aktuellerTA = TA (TA, startort, zielort) and (not tauschVorgeschlagen) and weg.tail = Wegberechnung (weg.head.zielknoten, startort)**
Folge-Zeitereignisse	**Ø**
Nachrichten	**Ø**
Aktionen	**Ø**

Regel 155: Nachricht_ annehmenTauschTA.3

Als Zusicherung für den Empfang der Nachricht *ablehnenTauschTA* kann gegeben werden, dass das FTF einen aktuellen TA hat und einen Tauschvorschlag gemacht hat.[109]

Ereignis	**Empfang(ablehnenTauschTA ( ) ← FTF [i])**
Zusicherung	**and aktuellerTA -> notEmpty() and tauschVorgeschlagen**

Zusicherung Nachricht_ablehnenTauschTA

Erhält ein FTF die Nachricht *ablehnenTauschTA*, so ändert es seinen Zustand dahingehend, dass es keinen Tauschvorschlag mehr gemacht hat.

[109] Als weitere Zusicherung kann auch hier gegeben werden, dass dieser Tauschvorschlag in Form einer Nachricht *tauschvorschlagTA* an ein anderes FTF (und nicht in Form einer Nachricht *optimierenTA* an alle anderen FTF) erfolgt ist. Da sich das FTF aus Gründen der Einfachheit nicht merkt, welche Nachrichten es verschickt hat, kann auch hier diese Zusicherung nicht als Bedingung des Zustands des FTF formuliert werden.

Ereignis	**Empfang(ablehnenTauschTA ( ) ← FTF [i])**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**not tauschVorgeschlagen**
Folge-Zeitereignisse	ø
Nachrichten	ø
Aktionen	ø

Regel 156: Nachricht_ ablehnenTauschTA.1

Als Zusicherung für das Auftreten des Zeitereignisses *timeoutTauschTA* kann gegeben werden, dass das FTF auf einem Knoten steht, einen aktuellen TA hat und einen Tauschvorschlag gemacht hat.[110]

Ereignis	**timeoutTauschTA ( )**
Zusicherung	**ort.typ = knoten and bewZust = stehend and aktuellerTA -> notEmpty() and tauschVorgeschlagen**

Zusicherung Zeitereignis_timeoutTauschTA

Tritt das Timeout für den Tausch auf und sind keine Tauschvorschläge eingegangen, so findet kein Tausch statt. Das FTF fährt den bereits geplanten Weg ab und ändert seinen Zustand dahingehend, dass es keinen Tauschvorschlag mehr gemacht hat.

Ereignis	**timeoutTauschTA ( )**
Ereignisbedingung	–
Zustandsbedingung	**tauschTab -> isEmpty()**
Zustandseffekt	**(not tauschVorgeschlagen)**
Folge-Zeitereignisse	**{ abfahrenWeg ( ) @ c }**
Nachrichten	ø
Aktionen	ø

Regel 157: Zeitereignis_timeoutTauschTA.1

Tritt das Timeout für den Tausch auf und ist mindestens ein Tauschvorschlag eingegangen, so wird der Tauschvorschlag mit der größten Ersparnis berücksichtigt. Das FTF aktualisiert den aktuellen TA, plant den Weg zum Startort seines neuen TA und beginnt den neu geplanten Weg abzufahren. Es ändert seinen Zustand dahingehend, dass es keinen Tauschvorschlag mehr gemacht hat. Die Tauschtabelle ist nun leer. Das FTF, das den berücksichtigten Tauschvorschlag gemacht hat, bekommt die Nachricht *annehmenTauschTA*. Sollte es weitere Tauschvorschläge gegeben haben, so erhalten die entsprechenden FTF die Nachricht *ablehnenTauschTA*.

[110] Als weitere Zusicherung kann auch hier gegeben werden, dass dieser Tauschvorschlag in Form einer Nachricht *optimierenTA* an alle anderen FTF (und nicht in Form einer Nachricht *tauschvorschlagTA* an ein anderes FTF) erfolgt ist. Da sich das FTF aus Gründen der Einfachheit nicht merkt, welche Nachrichten es verschickt hat, kann auch hier diese Zusicherung nicht als Bedingung des Zustands des FTF formuliert werden.

Ereignis	**timeoutTauschTA ( )**
Ereignisbedingung	–
Zustandsbedingung	–
Zustandseffekt	**let t = tauschTab.extractMax()** **(not tauschVorgeschlagen)** **and aktuellerTA = TA (t.TA, t.startort, t.zielort)** **and weg = Wegberechnung (ort, t.startort)** **and tauschTab -> isEmpty()**
Folge-Zeitereignisse	**{ abfahrenWeg ( ) @ c }**
Nachrichten	**{ annehmenTauschTA (aktuellerTA@pre.id, aktuellerTA@pre.startort, aktuellerTA@pre.zielort) → FTF [t.ftf] } ∪ { ablehnenTauschTA ( ) → FTF [i.ftf] | i ∈ tauschTab@pre, i ≠ t }**
Aktionen	**∅**

Regel 158: Zeitereignis_timeoutTauschTA.2

Die Regel *Nachricht_accept_proposal.1* (Regel 132) muss in dieser Variante angepasst werden. Empfängt das FTF eine Nachricht *accept_proposal*, so nimmt es den TA an, wenn es gerade noch frei ist. Es übernimmt wie in der ursprünglichen Regel den angenommenen TA als aktuellen TA und berechnet den Weg zum Startort. Es merkt sich außerdem, dass es einen Tausch vorgeschlagen hat. Das FTF beginnt nicht sofort, den geplanten Weg abzufahren. Stattdessen sendet es an jedes andere FTF die Nachricht *optimierenTA*. An den TA-Manager geht die Nachricht *accept_order*, um auszudrücken, dass der TA angenommen wurde.

Ereignis	**Empfang(accept_proposal (TA, startort, zielort) ← TAManager [i])**
Ereignisbedingung	–
Zustandsbedingung	**aktuellerTA -> isEmpty()**
Zustandseffekt	**aktuellerTA = TA (TA, startort, zielort) and weg = Wegberechnung (ort, startort) and tauschVorgeschlagen**
Folge-Zeitereignisse	**{ timeoutTauschTA ( ) @ c +** c_{tausch} **}**
Nachrichten	**{ accept_order (TA) → TAManager [i] } ∪ { optimierenTA (ort, startort) → FTF [i] | i ∈ FTF, i ≠ ID }**
Aktionen	**∅**

Regel 159: Nachricht_accept_proposal.1 in Variante Austausch Transportaufträge