Transitionsspezifikation und Konsistenznachweis eines Entity-Relationship-Modells für Datenbankapplikationen

Studienarbeit
vorgelegt von Roya Madani

August 1995

Zusammenfassung

In dieser Arbeit werden ein Entity-Relationship-Modell und Transitionen eines Beispielsystems mit Hilfe von Transitionsspezifikationen spezifiziert. Im Beispiel handelt es sich um ein System zur Unterstützung der Tourplanung innerhalb einer Speditionsfirma. Ein Entity-Relationship-Diagramm repräsentiert das Datenmodell. Zur formalen Spezifikation von ERD und Transitionen werden Transitionsspezifikationstechniken vorgestellt und verwendet. Im Anschluß daran wird der Konsistenznachweis für zwei Methoden durchgeführt.

Die Transitionsspezifikation wird zum Schluß im Hinblick auf ihre Transparenz und Komplexität kritisch betrachtet.

Inhaltsverzeichnis

Einleitung

Algebraische Spezifikationen sind syntaktische Konstruktionen, die zur Beschreibung von Datentypen benutzt werden. Eine algebraische Spezifikation legt zunächst die Anzahl der Mengen, Konstanten und Operationen mit ihrer Stelligkeit fest. Im nächsten Schritt werden Gleichungen angegeben, die die Operationen des intendierten Datentyps spezifizieren. Damit umfaßt der Begriff algebraische Spezifikation auf der Semantikebene nur durch Gleichungen charakterisierbare Algebren. Die Ausgabe einer Operation ist deterministisch durch ihre Eingabe bestimmt und kann nicht durch die Umgebung beeinflußt werden. Transitionsspezifikationen erweitern algebraische Spezifikationen um dynamische Operationen. Es werden Zustände und Zustandsübergänge eingeführt. Zustände sind abhängig von Operationen und werden durch Zustandsübergänge (Methoden) geändert. Dies ermöglicht den formalen Vergleich zwischen den Spezifikationen und imperativen Algorithmen.

Entity-Relationship-Diagramme beschreiben Datenmodelle eines Softwaresystems. Diese Formalismen stellen eine Syntax zur Verfügung, um Objekte aus der realen Welt zu beschreiben. Ziel dieser Arbeit ist mit Hilfe von Transitionsspezifikationen eine formale Semantik für Objekte und ihre dynamischen Beziehungen in einem Entity-Relationship-Diagramm zu geben.

Zunächst werden in Abschnitt1 die grundlegenden Begriffe, die zum Verständnis der darauffolgenden Spezifikationen und Beweise notwendig sind, erläutert.
In Abschnitt2 wird das Beispiel und das dazugehörige Modell vorgestellt. Zur formalen Spezifikation dieses Modells wird mit einem statischen Datentyp angefangen. Dies ist eine partielle Algebra, die im nächsten Schritt durch Referenzen erweitert wird. Zu jeder Spezifikation wird ihr Basismodell definiert.
In Abschnitt3 werden die angegebenen Spezifikationen auf ihre Korrektheit untersucht. Die Persistenz der vorgestellten dynamischen Methoden wird nachgewiesen.

Im Anschluß an diese einleitenden Worte möchte ich Prof. Dr. Hartmut Ehrig für sein entgegengebrachtes Interesse an dieser Arbeit danken. Rosi Bardohl danke ich für ihre Unterstützung und mein ganz besonderer Dank gilt meinem Betreuer Martin Große-Rhode, der durch seine konstruktive und mühevolle Betreuung zum Gelingen dieser Arbeit ganz wesentlich beigetragen hat.

1 Transitionsspezifikation

Grundlegend für das Verständnis dieser Spezifikation sind Begriffe wie Partielle Algebren und ihre Spezifikation, Zustände, Referenzen und Zustandsübergänge. Die Hauptidee der Transitionsspezifikation ist, Zustände als partielle Funktionen und Zustandsübergänge als Transformationen zwischen solchen Funktionen zu betrachten.

Zu einer gegebenen partiellen Gleichungsspezifikation wird eine partielle Algebra definiert. Diese Algebra beinhaltet Referenzen (=Elemente der Menge zur Referenzsorte) und Inhaltsfunktionen, die einer Referenz einen Wert zuordnen (total undefinierte Funktion).
In einem Zustand wird dieser Wert neu definiert, d.h. ein Zustand ist eine Erweiterung des Datentyps um ein "environment". Die aktuelle Definition der Inhaltsfunktion, die den aktuellen Zustand charakterisiert, kann wie andere Operationen des Datentyps durch bedingte Existenzgleichungen spezifiziert werden. Wenn A eine partielle Algebra, $d \in A$ eine Referenz und $a \in A$ ein Element der Trägermenge ist, definiert die Gleichung "$inhalt(d) = a$" einen neuen Wert a für eine Referenz.

Zustandsübergänge sind entweder Transformations-Methoden oder Create- bzw. Delete-Methoden. Transformations-Methoden weisen einer Liste von Referenzen neue Inhalte zu. Create-Methoden initialisieren Referenzen mit Werten und Delete-Methoden löschen die Werte. Es ist zu beachten, daß dadurch keine Referenzen erzeugt bzw. gelöscht werden sondern, daß der Definitionsbereich von Inhaltsfunktion geändert wird.

In diesem Abschnitt werden die Basisdefinitionen gegeben. Alle zugrundeliegenden Definitionen stammen aus [GR 95].

1.1 Partielle Algebren

Eine **Signatur** $SIG = (S,\ OP)$ ist gegeben durch eine Menge S von Sortennamen und eine $S^* \times S^*$–indizierte Familie $OP = (OP_{w,v})_{w,v \in S^*}$ von Operationsymbolen, in gleicher Weise wie für totale Algebren. Allerdings gilt für die Operationen $v \in S^*$.
Die Menge der Terme zu einer Signatur $Term = Term(SIG)$ wird entsprechend wie folgt definiert:

- Variablen
  $x_i : s_1 \ldots s_n \to s_i \in Term \quad (s \in S, i = 1, \ldots, n)$

- Tupel
  wenn $\quad t_1 : v \to s_1, \ldots, t_n : v \to s_n \in Term$
  dann $\quad \langle t_1, \ldots, t_n \rangle : v \to s_1 \ldots s_n \in Term$

- Operationen

wenn	$\langle t_1, \ldots, t_n\rangle : v \to w \in Term$	
und	$\sigma : w \to s'_1 \ldots s'_k \in OP$	
dann	$\sigma_i(t_1, \ldots t_n) : v \to s'_i \in Term$	für $i = 1, \ldots k$
bzw.	$\sigma(t_1, \ldots t_n) : v \to \lambda \in Term$	falls $k = 0$

Eine **partielle** *SIG*–**Algebra** $A = ((A_s)_{s\in S}, (\widehat{A_\sigma})_{\sigma\in OP})$ ist gegeben durch

- eine Familie von Mengen $(A_s)_{s\in S}$, die *Trägermengen* von A
- eine Familie von partielle Funktionen $(\widehat{A_\sigma})_{\sigma\in OP}$, die *Operationen* von A

  - $\widehat{A_\sigma} : A_w \rightarrow\!\!\!\!\!\circ\;\, A_v$ falls $\sigma : w \to v \in OP$

  $\widehat{A_\sigma}$ ist gegeben durch

  - eine Teilmenge $dom(A_\sigma) \subseteq A_w$, der *Definitionsbereich* von A_σ
  - eine totale Funktion $A_\sigma : dom(A_\sigma) \to A_v$

(Mit $A_u = A_{s_1} \times \ldots \times A_{s_n}$ für $u = s_1 \ldots s_n$.)
Die **Auswertung** eines Terms $t : w \to v \in Term$ in einer partiellen *SIG*–Algebra A ist die partielle Funktion
$A_t : A_w \rightarrow\!\!\!\!\!\circ\;\, A_v$ definiert wie folgt:

- Variablen
  $dom(A_{x_i}) = A_{s_1} \times \ldots \times A_{s_n}$
  $A_{x_i}(\langle a_1, \ldots, a_n\rangle) = a_i$
- Tupeln
  $dom(A_{\langle t_1,\ldots,t_n\rangle}) = \bigcap_{1\le i\le n} dom(A_{t_i})$

  $A_{\langle t_1,\ldots,t_n\rangle}(a) = \langle A_{t_1}(a), \ldots, A_{t_n}(a)\rangle$
- Operationen
  $dom(A_{\sigma(t_1,\ldots,t_n)}) = \{a \in dom(A_{\langle t_1,\ldots,t_n\rangle}) \mid \langle A_{t_1}(a), \ldots, A_{t_n}(a)\rangle \in dom(A_\sigma)\}$
  $A_{\sigma(t_1,\ldots,t_n)}(a) = A_\sigma(A_{t_1}(a), \ldots, A_{t_n}(a))$

Eine **Existenzgleichung** e zu einer Signatur *SIG* ist gegeben durch ein Paar von Termen zur gleichen Sorte $t, t' : w \to v \in Term$, geschrieben $e = (w : t \stackrel{e}{=} t')$. $w = s_1 \ldots s_n$ repräsentiert die Deklaration $x_1 : s_1, \ldots, x_n : s_n$ von Variablen,die in t oder t' vorkommen können. Eine **bedingte Existenzgleichung** ce bzgl. *SIG* ist gegeben durch zwei Paare von Termen zur gleichen Sorte. $r, r' : w \to v'$, und $t, t' : w \to v \in Term$ mit gemeinsamer Quelle w, geschrieben $ce = (w : r \stackrel{e}{=} r' \to t \stackrel{e}{=} t')$.
Die **Lösungsmenge** A_e einer Existenzgleichung $e = (w : t \stackrel{e}{=} t')$ ist definiert durch

- $A_e =_{def} \{a \in dom(A_t) \cap dom(A_{t'}) \mid A_t(a) = A_{t'}(a)\}$

und A **erfüllt die bedingte Existenzgleichung** $ce = (w : r \stackrel{e}{=} r' \to t \stackrel{e}{=} t')$, geschrieben $A \models ce$, falls

- $A_{(w:r \stackrel{e}{=} r')} \subseteq A_{(w:t \stackrel{e}{=} t')}$

Jede Existenzgleichung $e = (w : t \stackrel{e}{=} t')$ kann als eine bedingte Existenzgleichung $ce = (w : \langle\rangle \stackrel{e}{=} \langle\rangle \to t \stackrel{e}{=} t')$ betrachtet werden. Da $A_{(w:\langle\rangle \stackrel{e}{=} \langle\rangle)} = A_w$ ist, gilt $A \models ce$, falls $A_w \subseteq A_{(w:t \stackrel{e}{=} t')}$, d.h. jedes Element $a \in A_w$ ist eine Lösung von e. In diesem Fall wird $A \models e$ als kurze Notation für $A \models ce$ benutzt. Nach der Definition der Lösungsmenge und der obigen Konvention gilt $A \models (w : t \stackrel{e}{=} t)$, falls es sich bei $A_t : A_w \multimap A_v$ um eine totale Funktion handelt. Insbesonders gilt $A \models (\lambda : t \stackrel{e}{=} t)$ für einen Grundterm $t : \lambda \to v$, falls $A_t \in A_v$ definiert ist. Im folgenden wird statt $(w : t \stackrel{e}{=} t)$, $(w : t \uparrow)$ geschrieben.
Eine Partielle Gleichungsspezifikation $SPEC = (S,\ OP,\ CE)$ ist gegeben durch

- eine Signatur $SIG = (S,\ OP)$ und
- eine Menge CE von bedingten Existenzgleichungen bzgl. SIG.

Eine partielle SIG–Algebra A ist eine **partielle** $SPEC$**-Algebra**, falls A alle bedingten Existenzgleichungen in CE erfüllt. Ein $SPEC$–**Homomorphismus** $h : A \to B$ ist das gleiche wie ein SIG–Homomorphismus, falls A und B partielle $SPEC$–Algebren sind.

1.2 Basisspezifikationen, Zustände, Zustandsübergänge

Definition 1.1 (Basisspezifikation) Eine **Basisspezifikation** $BaseSPEC = (S, OP, CE, ref)$ ist gegeben durch eine partielle Gleichungsspezifikation $SPEC = (S, OP, CE)$ und eine partielle Funktion $ref : S \multimap S$, so, daß die folgenden Bedingungen erfüllt sind.

1. Es gibt Operationsymbole $!_s : ref(s) \to s \in OP$ für alle $s \in dom(ref)$
2. Jedes Operationssymbol $\sigma : w \to v \in OP$ ist entweder eine Konstante zu einer Referenzsorte d.h. $\sigma : \lambda \to ref(s)$, oder $v = s_1 \dots s_n$ und $v \in (S - ref(S))^*$.
3. Jede Referenz ist definiert, d.h. $(\lambda : \delta \stackrel{e}{=} \delta) \in CE$ für alle $\delta : \lambda \to ref(s) \in OP$
4. Es gibt keine Gleichungen zwischen Referenzen, d.h.
   $target(t_1) \in (S - ref(S))^*$ für alle $(w : r_1 \stackrel{e}{=} r_2 \to t_1 \stackrel{e}{=} t_2) \in CE$

Definition 1.2 (Zustände) Sei $BaseSPEC = (SPEC, ref)$ eine Basisspezifikation, A ein $BaseSPEC$ –Model.

1. Für jedes Paar bestehend aus einer Liste von Referenzen $d_w = \langle d_1, \dots, d_n \rangle \in A_{ref(s_1)} \times \dots \times A_{ref(s_n)}$ mit $d_i \neq d_j$ für $i \neq j$, und einer Liste von Elementen $a_w = \langle a_1, \dots, a_n \rangle \in A_{s_1} \times \dots \times A_{s_n}$, ist der **Zustand** $A^{[d_w := a_w]}$ über A die freie Erweiterung von A durch die A–Gleichung $(\lambda :\ !d_w \stackrel{e}{=} a_w) = (\lambda :\ !d_1 \stackrel{e}{=} a_1 \wedge \dots \wedge\ !d_n \stackrel{e}{=} a_n)$, d.h.

   - $A^{[d_w := a_w]} =_{def} A[(\lambda :\ !d_w \stackrel{e}{=} a_w)]$

2. Der universelle Morphismus $\eta^{st} : A \to A^{st}$ (mit $st = [d_w := a_w]$) erzeugt durch die freie Erweiterung wird als **Zustandsmorphismus** bezeichnet.

3. Zwei Zustände A^{st} and $A^{st'}$ sind **äquivalent**, falls ein Isomorphismus $i : A^{st} \to A^{st'}$ mit $i \circ \eta^{st} = \eta^{st'}$existiert.

Die Menge $State(A)$ ist definiert als Menge der Äquivalenzklassen von Zuständen über A, d.h.

- $State(A) = \{[A^{st}] \mid A^{st} \text{ ist ein Zustand über } A\}$

Definition 1.3 (Methodensignatur) Eine **Methodensignatur** $MetSIG = (M,C)$ bzgl. einer gegebenen Basisspezifikation $BaseSPEC = (S,OP,CE,ref)$ ist gegeben durch

- eine Familie von Mengen $M = (M_{w,p})_{w,p \in S^*}$, die *Menge der Methoden*,
- eine Familie von Teilmengen $C = (C_{w,p})_{w,p \in S^*}$, mit $C_{w,p} \subseteq M_{w,p}$ für alle $w, p \in S^*$, die *Menge von Create-Methoden.*

Die Methoden $m \in M_{w,p}$ werden geschrieben $m : ref(w); p$, mit $ref(w) =_{def} ref(s_1) \ldots ref(s_n)$. Falls $m \in M_{w,p} - C_{w,p}$,dann wird m als *Transformations-Methode* bezeichnet.

Definition 1.4 (Methodenausdrücke) Gegeben sei eine Basisspezifikation $BaseSPEC$, eine entsprechende Methodsignatur $MetSIG = (M,C)$, und ein $BaseSPEC$ –Modell A. Die Menge der **Methodenausdrücke** $MetExp(A)$ ist definiert durch die folgenden Regeln:

- *Basis-Methoden :*

  falls $m \in M_{w,p}$
  $d_w \in A_{ref(w)}$
  mit $d_w = \langle d_1, \ldots, d_n \rangle$ und $d_i \neq d_j$ (for $i \neq j$)
  $a_p \in A_p$
  dann $m(d_w; a_p) \in MetExp(A)$

- *Delete-Methoden :*

  falls $d_w \in A_{ref(w)}$
  $d_w = \langle d_1, \ldots, d_n \rangle$ und $d_i \neq d_j$ (für $i \neq j$)
  dann $del(d_w) \in MetExp(A)$

- *Sequentielle-Methoden :*
  - $skip \in MetExp(A)$
  - $met_1 \in MetExp(A), met_2 \in MetExp(A) \Rightarrow met_1; met_2 \in MetExp(A)$
  - $met; skip = skip; met = met$ für alle $met \in MetExp(A)$
  - $met_1; (met_2; met_3) = (met_1; met_2); met_3$ für alle $met_1, met_2, met_3 \in MetExp(A)$

Definition 1.5 (Methoddefinition) Gegeben sei eine Basisspezifikation $BaseSPEC$ und eine entsprechende Methodsignatur $MetSIG = (M,C)$. Eine **Methoddefinition** def ist gegeben durch

- $def = (v : r_1 \stackrel{e}{=} r_2 \to m(\Delta; t) := r)$

oder

- $def = (v : r_1 \stackrel{e}{=} r_2 \to del(\Delta))$

wobei

- $v \in S^*$
- $r_1, r_2 : v \to v' \in Term(BaseSPEC)$
- $m \in M_{w,p}$
- $\Delta : v \to ref(w) \in Term(BaseSPEC)$
  mit $w = s_1 \dots s_n$, $\Delta = \langle \Delta_1, \dots, \Delta_n \rangle$ und $\Delta_i \neq \Delta_j$ für $i \neq j$
- $t : v \to p \in Term(BaseSPEC)$
- $r : v \to w \in Term(BaseSPEC)$

Bemerkung: Δ_i ist entweder eine Referenzkonstante ($\Delta_i = \delta :\to ref(s_i)$) oder eine Variable ($\Delta_i = x_i$).

Definition 1.6 (Transitionsspezifikation, Methodspezifikation) eine **Transitionsspezifikation** $TS = (BaseSPEC, MetSPEC)$ ist gegeben durch eine Basisspezifikation $BaseSPEC$ und eine **Methodspezifikation** $MetSPEC = (MetSIG, DEF)$, wobei

- $MetSIG$ eine Methodsignatur bzgl. $BaseSPEC$ und
- DEF eine Menge von Methoddefinitionen bzgl. $MetSIG$ und $BaseSPEC$ ist.

Definition 1.7 (Auswertung von Methodenausdrücke) Sei $TS = (BaseSPEC, MetSPEC)$ eine Transitionsspezifikation mit $MetSPEC = (M, C, DEF)$, A ein $BaseSPEC$-Modell. Jeder Methodenausdruck $met \in MetExp(A)$ definiert eine Relation $A^{met} \subseteq State(A) \times State(A)$, geschrieben $A^{st_1} \xrightarrow{met} A^{st_2}$ wie folgt:

1. **Transformations-Methoden** : $met = m(d_w, a_p)$, $\quad m \in M_{w,p} - C_{w,p}$
Gegeben sei

- ein Zustand $A^{st} = A^{[\langle d_w, d_u \rangle := \langle a_w, a_u \rangle]}$
- eine Methoddefinition $(v : r_1 \stackrel{e}{=} r_2 \to m(\Delta; t) := r) \in DEF$
- ein Element $a_v \in A_v$ (d.h. eine Auswertung $x_v \mapsto a_v^{st} \in A^{st}$) und ein Element $b_w \in A_w$ so, daß
  - Δ und t bzgl. der Auswertung $x_v \mapsto a_v^{st}$ zu d_w^{st} und a_p^{st} in A^{st} ausgewertet werden:

    $A_\Delta^{st}(a_v^{st}) = d_w^{st}$ und $A_t^{st}(a_v^{st}) = a_p^{st}$

– A^{st} erfüllt die Bedingungen von def:

$A^{st}_{r_1}(a^{st}_v) = A^{st}_{r_2}(a^{st}_v)$

– r bzgl. der Auswertung $x_v \mapsto a^{st}_v$ zu b^{st}_w in A^{st} ausgewertet wird:

$A^{st}_r(a^{st}_v) = b^{st}_w$

dann ist

- $A^{[\langle d_w,d_u\rangle:=\langle a_w,a_u\rangle]} \xrightarrow{m(d_w;a_p)} A^{[\langle d_w,d_u\rangle:=\langle b_w,a_u\rangle]}$

eine Transition.

2. **Create-Methoden** : $met = c(d_w, a_p), \quad c \in C_{w,p}$
Gegeben seien

- ein Zustand $A^{st} = A^{[d_u:=a_u]}$
- so, daß $\{d'_1, \ldots, d'_m\} \cap \{d_1, \ldots, d_n\} = \emptyset$
für $d_u = \langle d'_1, \ldots, d'_m\rangle$ und $d_w = \langle d_1, \ldots, d_n\rangle$
- eine Methoddefinition $(v : r_1 \stackrel{e}{=} r_2 \to c(\Delta; t) := r) \in DEF$
- Elemente $a_v \in A_v$ und $b_w \in A_w$ so, daß
  – Δ und t bzgl. der Auswertung $x_v \mapsto a^{st}_v$ zu d^{st}_w und a^{st}_p in A^{st} ausgewertet werden:

    $A^{st}_\Delta(a^{st}_v) = d^{st}_w$ und $A^{st}_t(a^{st}_v) = a^{st}_p$

  – A^{st} erfüllt die Bedingungen von def:

    $A^{st}_{r_1}(a^{st}_v) = A^{st}_{r_2}(a^{st}_v)$

  – r bzgl. der Auswertung $x_v \mapsto a^{st}_v$ zu b_w in A^{st} ausgewertet wird:

    $A^{st}_r(a^{st}_v) = b^{st}_w$

dann ist

- $A^{[d_u:=a_u]} \xrightarrow{c(d_w;a_p)} A^{[\langle d_w,d_u\rangle:=\langle b_w,a_u\rangle]}$

eine Transition.

3. **Delete-Methoden** : $met = del(d_w)$
Gegeben sei

- ein Zustand $A^{st} = A^{[\langle d_w,d_u\rangle:=\langle a_w,a_u\rangle]}$
- eine Methoddefinition $def = (v : r_1 \stackrel{e}{=} r_2 \to del(\Delta)) \in DEF$
- ein Element $a_v \in A_v$ so, daß

- Δ zu d_w^{st} in A^{st} ausgewertet wird (für $x_v \mapsto a_v$):

  $A_\Delta^{st}(a_v^{st}) = d_w^{st}$

- A^{st}erfüllt die Bedingungen von def:

  $A_{r_1}^{st}(a_v^{st}) = A_{r_2}^{st}(a_v^{st})$

dann ist

- $A^{[\langle d_w, d_u\rangle := \langle a_w, a_u\rangle]} \xrightarrow{del(d_w)} A^{[d_u := a_u]}$

eine Transition.

4. **Sequentielle-Methoden** : $met = skip, met = met_1; met_2$

- $A^{st} \xrightarrow{skip} A^{st}$ ist eine Transition für jeden Zustand A^{st}
- falls $A^{st_1} \xrightarrow{met_1} A^{st'_1}$ und $A^{st_2} \xrightarrow{met_2} A^{st'_2}$ Transitionen sind und $A^{st'_1} \equiv A^{st_2}$, dann ist $A^{st_1} \xrightarrow{met_1;met_2} A^{st'_2}$ eine Transition.

2 Das Beispiel: Ein Softwaresystem für eine Speditionsfirma

Die Methoden der Transitionsspezifikation werden in diesem Abschnitt anhand eines zentralen Beispiels veranschaulicht. Im Beispiel handelt es sich um einen Ausschnitt des Projekts *Betriebs-Ablauf-System für Transportabwicklung,* das in Zusammenarbeit der Spedition *Rasch, Hamm* mit den Firmen *MSB, Berlin* und *MST, Hamm* durchgeführt wird.
Es soll ein System zur Unterstützung der Tourplanung innerhalb einer Speditionsfirma entwickelt werden. Zunächst wird ein kurzer Überblick über einen Teil der Aufgaben im Rahmen der Tourplanung gegeben. Eine Modellierung mit ERD wird danach vorgestellt.

2.1 Die Aufgaben im Rahmen der Tourplanung

Bestimmte Daten müssen für die Tourplanung ständig in aktualisierter Form verfügbar sein. Dazu zählen:

- Auftraggeber mit Artikel- und Empfängerliste
- Fahrer
- Fuhrpark

Die Aufträge müssen erfaßt und der Tourplanung zur Verfügung gestellt werden. Aufträge werden zu Sendungen zusammengefaßt. Eine Sendung ist ein Paket, welches die Aufträge enthält, die den selben Lieferort und das selbe Lieferdatum haben. Die Sendungen wiederum werden zu Touren zusammengefaßt.
Folgende Aufgaben sollen von dem System unterstützt werden:

- Erfassen und Verwalten der Auftraggeber zusammen mit ihren Artikeln und Empfängern.
- Verwaltung der für die Tourplanung erforderlichen Fahrzeugdaten.
- Verwaltung der Fahrer.
- Erfassen und Bearbeiten von Aufträgen.
- Zusammenstellung von Aufträgen zu Sendungen.
- Zusammenstellung von Sendungen zu Touren.

2.2 Modellierung mit Entity-Relationship-Diagramm

Zur Beschreibung des Datenmodells werden die Objekte und die Beziehungen zwischen diesen Objekten mit ERDen repräsentiert.

Die Objekte haben folgende Attributierung:

Artikel:	Gewicht, Volumen
Auftraggeber:	Name, Anschrift
Empfänger:	Name, Anschrift
Auftrag:	Abholdatum, Lieferdatum, Status
Sendung:	Status
Fahrzeug:	Nutzlast, Nutzvolumen
Fahrer:	Name, Anschrift
Tour:	Tourdatum, Status

Für die Objekte ergibt sich das in Abbildung 2.1 dargestellte Entity-Relationship-Diagramm.

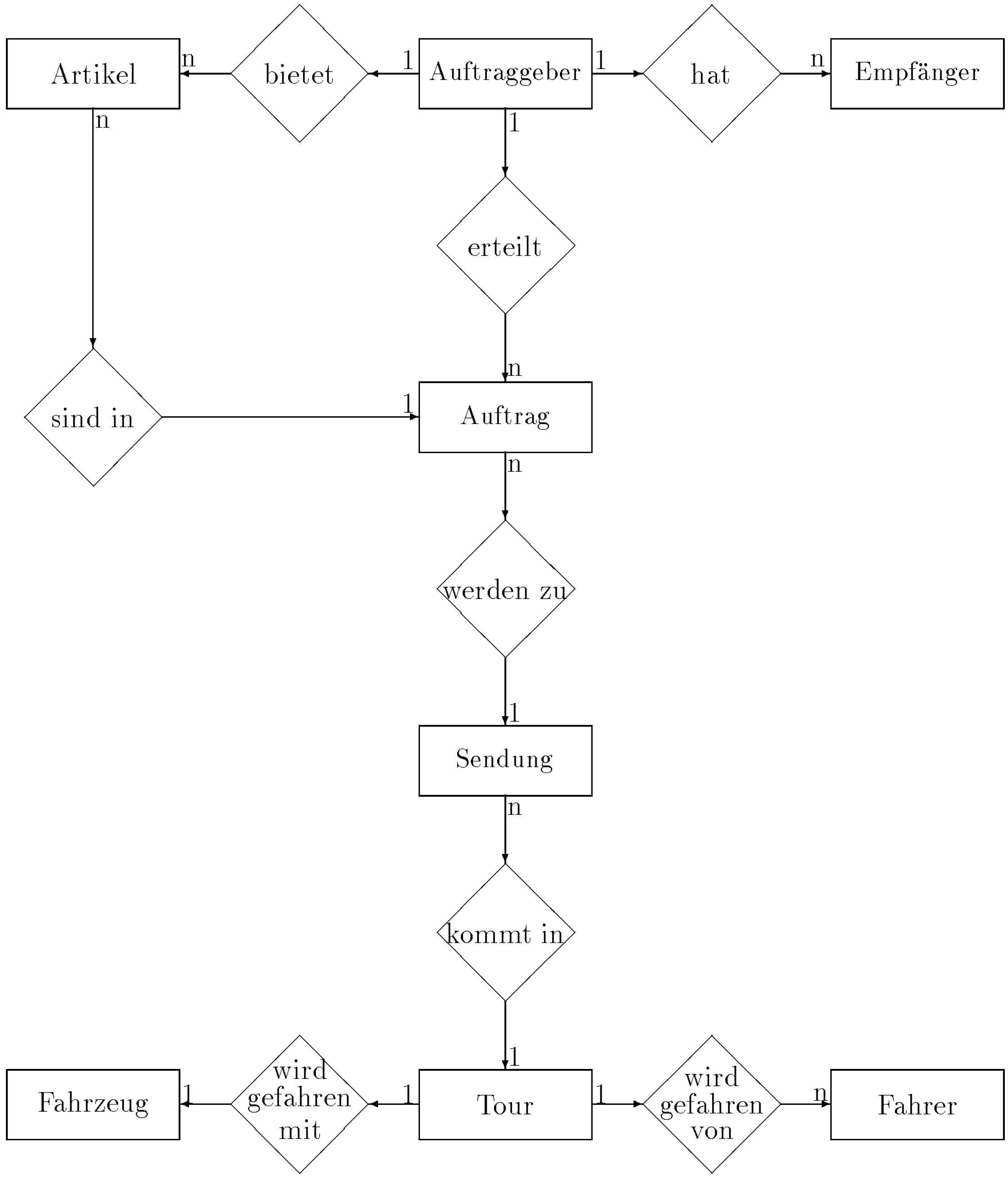

Abbildung 2.1: ERD für das Speditionsbeispiel

2.3 Die Transitionspezifikation des Entity-Relationship-Modells

Zu jedem Objekt und jeder Beziehung zwischen zwei Objekten in ERD wird eine Basisspezifikation und darauf aufbauend eine Transitionsspezifikation angegeben.

`AUFG` ist ein statischer Datentyp von Auftraggebern und ihren Referenzen. Die Operationen sind Konstruktoren (cons) und Funktionen (funs). Ein Auftraggeber besteht aus Namen und Adresse. Dazu wird ein Konstruktor *cons* benutzt. Um auf die Namen und Adressen zugreifen zu können, werden zwei Funktionen (Projektionen) benutzt. Aufg-Name und Aufg-Adr projezieren einen Auftraggeber. Eine Adresse und ein Name wird mit Hilfe von zwei weiteren Funktionen geändert. Die Inhaltsfunktion !:`ref(Aufg)`→`Aufg` wird, wie üblich, als ein Default betrachtet.

```
type AUFG = NAME + ADRESSE
  sorts  Aufg, ref(Aufg)

  cons   cons :  Name, Adresse → Aufg

  funs   !  :  ref(Aufg) → Aufg

         Aufg-Name :  Aufg → Name
         for all na:Name, ad:Adresse :
         Auf-Name(cons(na,ad)) = na

         Aufg-Adr :  Aufg → Adresse
         for all na:Name, ad:Adresse :
         Auf-Adr(cons(na,ad)) = ad

         Change-Adr :  Aufg, Adresse → Aufg
         for all ad,ad':Adresse, na:Name :
         Change-Adr(cons(na,ad), ad') = cons(na,ad')

         Change-Name :  Aufg, Name → Aufg
         for all ad:Adresse, na', na:Name :
         Change-Name(cons(na,ad), na') = cons(na',ad)


  axms   for all a:Aufg
         a = cons (Aufg-Name(a), Aufg-Adr(a))

end type
```

Ein AUFG-Basismodell A zur AUFG-Basisspezifikation wird wie folgt gegeben.

$$\begin{aligned} A_{Aufg} &= \{a,\ldots,z\}^* \times (\{a,\ldots,z\} \cup \mathrm{IN})^* \\ A_{ref(Aufg)} &= \mathrm{IN} \end{aligned}$$

Mit zwei Projektionen, zwei Funktionen zur Veränderung der Komponenten und

$$\begin{aligned} A_! &= \emptyset : A_{ref(Aufg)} \twoheadrightarrow A_{Aufg} \\ &= \text{die total undefinierte Funktion} \end{aligned}$$

Die Basisspezifikation von AUFG wird nun um eine Create-Methode erweitert . Mit dieser Methode soll das dynamische Erzeugen eines Auftraggebers spezifiziert werden.

```
dyntype  CREATE-AUFG = AUFG +
  met    Create-Aufg :  ref(Aufg); Name, Adresse
         for all p:ref(Aufg), na:Name, ad:Adresse
         Create-Aufg(p; na,ad) := cons(na,ad)
end dyntype
```

Die Spezifikation von Empfängerobjekten kann analog angegeben werden:

```
type EMPF = NAME + ADRESSE
  sorts Empf, ref(Empf)

  cons   cons :  Name, Adresse → Empf

  funs   !  :  ref(Empf) → Empf

         Empf-Name :  Empf → Name
         for all na:Name, ad:Adresse :
         Empf-Name(cons(na,ad)) = na

         Empf-Adr :  Empf → Adresse
         for all na:Name, ad:Adresse :
         Empf-Adr(cons(na,ad)) = ad

         Change-Adr :  Empf,Adresse  → Empf
         for all ad,ad':Adresse, na:Name :
         Change-Adr(cons(na,ad), ad') = cons(na,ad')

         Change-Name :  Empf, Name → Empf
         for all ad:Adresse, na', na:Name :
         Change-Name(cons(na,ad), na') = cons(na',ad)


  axms   for all e:Empf
```

```
        e = cons (Empf-Name(e), Empf-Adr(e))

end type
```

Die entsprechende Create-Methode wird auch analog definiert

```
dyntype  CREATE-EMPF = EMPF +
  met    Create-Empf :  ref(Empf); Name, Adresse
         for all p:ref(Empf), na:Name, ad:Adresse
         Creat-Empf(p;na,ad) := cons(na,ad)
end dyntype
```

Als EMPF-Basismodell werden die gleichen Trägermengen und Funktionen gewählt, also

$$\begin{array}{lcl} A_{Empf} & = & \{a,\ldots,z\}^* \times (\{a,\ldots,z\} \cup \mathrm{IN})^* \\ A_{ref(Empf)} & = & \mathrm{IN} \end{array}$$

$$\begin{array}{lcl} A_! & = & \emptyset : A_{ref(Empf)} \twoheadrightarrow A_{Empf} \\ & = & \text{die total undefinierte Funktion} \end{array}$$

Die Empfängerobjekte werden auch dynamisch erzeugt so, daß jeder neue Tupel von Namen und Adresse eines Empfängers als Wert für eine Referenz bestimmt wird. Dabei darf dieser Referenz vorher kein Wert zugeordnet sein.

Um die Beziehung (Relationship) "hat" zu spezifizieren, werden die Spezifikationen AUFG und EMPF vorausgesetzt. Es wird eine neue Sorte Listh = $List(ref(Aufg) \times ref(Empf))$ definiert, die die möglichen "hat"-Beziehungen kodiert. Die Listen werden durch einen Konstruktor _::_ gebildet. Der wichtigste Punkt ist, diese Relation so zu spezifizieren, daß die 1-n Beziehung nicht verletzt wird. D.h. ein neues Paar darf nur dann in die Liste aufgenommen werden, wenn ein Auftraggeber mit einem oder mehreren Empfängern in Beziehung steht und nicht umgekehrt. Die Referenzkonstante "Hat" zeigt auf eine Liste von Paaren, die aus Referenzen bestehen. "Hat" wird mit einer Liste initialisiert, die die Konsistenzbedingung (1-n Beziehung) der Liste erfüllt. Die Konsistenz der Liste wird mit Hilfe zwei Prädikaten "Not-in" und "Kons" überprüft. Eine leere Liste ist konsistent. Für ein gegebenes Paar wird in "Not-in" überprüft, ob das dereferenzierte Paar in der Liste von Auftraggebern und Empfängern nicht vorhanden oder sein Empfänger von jedem schon vorhandenen Empfänger verschieden ist. In beiden Fällen liefert "Not-in" True. "Deref" ist eine Funktion, die aus einer Liste von Referenzpaaren eine Liste von Paaren bestehend aus Auftraggebern und Empfängern liefert. Die Prädikate benutzen eine Reihe von Zugriffsfunktionen wie Hd (liefert das erste Paar einer Liste), Tl (liefert den Rest der Liste), Right bzw. Left greifen auf das rechte bzw. linke Element eines Paares zu.

```
type HAT = AUFG + EMPF
  sorts Listh
        ref(Listh)
        List(Aufg × Empf)

  cons  [] :  → Listh
        _::_ :  ref(Aufg) × ref(Empf), Listh → Listh
        _::_ :  Listh, ref(Aufg) × ref(Empf) → Listh
        Hat :  → ref(Listh)

  funs  !  :  ref(Listh) → Listh

        Empty:  Listh
        Empty([])


        Not-Empty:  List(Aufg × Empf)
        for all a:Aufg, e:Empf, l:List(Aufg × Empf)
        Not-Empty( (a,e)::l)


        Hd :  List(Aufg × Empf) →  Aufg × Empf
        for all a:Aufg, e:Empf, l:List(Aufg × Empf)
        Hd((a,e) ::l) = (a,e)


        Tl :  List(Aufg × Empf) → List(Aufg × Empf)
        for all a:Aufg, e:Empf, l:List(Aufg × Empf)
        Tl((a,e)::l) = l


        Right :  Aufg × Empf → Empf
        for all a:Aufg, e:Empf
        Right(a,e) = e


        Left :  Aufg × Empf → Aufg
        for all a:Aufg, e:Empf
        Left(a,e) = a

        Deref :  Listh → List(Aufg × Empf)
        for all a:ref(Aufg), e:ref(Empf), l:Listh
        Deref([]) = []
```

```
!a↑ ∧ !e↑ ∧ Deref(l)↑ → Deref((a,e)::l) = (!a,!e)::Deref(l)


Not-in:  Aufg × Empf, List(Aufg × Empf)
for all a:Aufg, e:Empf, l:List(Aufg × Empf)
Not-in((a,e),[])

Right((a,e) ) ≠ Right ( Hd(l)) ∧ Not-in( (a,e) , Tl(l)) →
Not-in((a,e) , l)


Kons:  List(Aufg × Empf)
for all a:Aufg, e:Empf, l:List(Aufg × Empf)
Kons([])
Kons(l) ∧ Not-in((a,e), l) →
Kons((a,e)::l)


  axms   Kons(Deref(!Hat))

end type
```

Das Axiom Kons(Deref(!Hat)) ist die Invariante, die ausdrückt, daß die Konsistenzbedingung (1-n Beziehung) in jedem Zustand erfüllt ist. Der Konsistenznachweis für die Methoden in Abschnitt3 besteht im wesentlichen in dem Beweis, daß die Methoden dieses Axiom bewahren.

Der Datentyp "HAT" wird zuerst mit zwei Methoden zur Initialisierung (Createh) und zum Hinzufügen (Add) von Paaren unter Berücksichtigung der Konsistenzbedingung erweitert.

```
dyntype HAT1 = HAT +
  met    Createh :  ref(Listh); Listh → Listh
  defs   for all l:Listh
         Kons(Deref(l)) → Createh(Hat; l) := l

  met    Add :  ref(Listh); ref(Aufg) × ref(Empf) → Listh
  defs   for all a:ref(Aufg), e:ref(Empf)
         Not-in((!a,!e ), Deref(!Hat)) → Add(Hat;(a,e)) := (a,e)::!Hat

end dyntype
```

"HAT2" ist eine Erweiterung des Datentyps HAT um drei weiteren Methoden. Mit diesen Methoden wird die Entfernung eines Paares aus der Hat-Liste realisiert. In Remove wird eine leere oder eine einelementige Liste behandelt. Wenn die Liste mehr als ein Element hat, werden in Init die Referenzkonstanten Hat und Hilf initialisiert. In Remove' wird rekursiv in der Liste nach dem gegebenen Paar (a,e) gesucht. Wenn das Paar in der Liste vorhanden ist, wird es gelöscht. Dabei wird mit Hilfe der Hilf-Referenz die Liste ohne (a,e) wieder konstruiert. Damit sich beim Durchsuchen der Liste die Reihenfolge der Elemente nicht ändert, wird eine Listenfunktion _::_ : Listh, ref(Aufg) × ref(Empf) → Listh benutzt, die ein Paar rechts an die Liste hängt.

```
dyntype HAT2 = HAT +
  cons  Hilf :  → ref(Listh)

  mets  Remove :  ref(Listh); ref(Aufg) × ref(Empf)
        Init :  ref(Listh), ref(Listh); ref(Aufg) × ref(Empf)
        Remove' :  ref(Listh), ref(Listh); ref(Aufg) × ref(Empf)
  defs  for all a:ref(Aufg), e:ref(Empf), l:Listh
        !Hat = [] → Remove(Hat;(a,e)) := [].  ∅
        !Hat = ((a',e')) ∧ (!a',!e') = (!a,!e) →
        Remove(Hat;(a,e)) := [].  ∅
        !Hat = ((a',e')) ∧ (!a',!e')≠ (!a,!e) →
        Remove(Hat;(a,e)) := !Hat.  ∅
        !Hat = (a',e')::l ∧ Not-Empty(l) →
        Remove(Hat;(a,e)) := !Hat.  {Init(Hat,Hilf;(a,e))}

        Init(Hat,Hilf;(a,e)) := (!Hat, []).  {Remove' (Hat,Hilf;(a,e))}

        Hd(!Hat) = (a',e') ∧ (!a',!e') = (!a,!e) →
        Remove'(Hat,Hilf;(a,e)) := (Tl(!Hat), !Hilf).  {Remove'(Hat,Hilf;(a,e))}
        Hd(!Hat) = (a',e') ∧ (!a',!e')≠ (!a,!e) →
        Remove'(Hat,Hilf;(a,e)) := (Tl(!Hat), !Hilf::(a',e')).
        {Remove'(Hat,Hilf;(a,e))}
        Empty(!Hat) → Remove'(Hat,Hilf;(a,e)) := (!Hilf, []).  ∅

end dyntype
```

Nun wird das HAT-Basismodell H definiert.

$$
\begin{array}{lcl}
H_{Listh} & = & (\mathrm{IN} \times \mathrm{IN})^* \\
H_{ref(Listh)} & = & \{Hat\} \\
H_{List(Aufg \times Empf)} & = & (A_{Aufg} \times A_{Empf})^* \\
H_{[]} & = & \lambda \\
H_{\_::\_}(x, w) & = & xw, \quad mit \quad w \in (\mathrm{IN} \times \mathrm{IN})^*, x \in \mathrm{IN} \times \mathrm{IN} \\
H_{\_::\_}(w, x) & = & wx, \quad mit \quad w \in (\mathrm{IN} \times \mathrm{IN})^*, x \in \mathrm{IN} \times \mathrm{IN} \\
H_{Hat} & = & Hat \\
H_{!} & = & \emptyset \\
H_{Empty} & = & \{\lambda\} \\
H_{Not-Empty} & = & (\mathrm{IN} \times \mathrm{IN})^+ \\
H_{Hd}(x :: w) & = & x \\
H_{Tl}(x :: w) & = & w \\
H_{Right}(a, e) & = & e \\
H_{Left}(a, e) & = & a \\
H_{Deref}(l) & = & \lambda, falls \quad l = \lambda, \quad undefiniert \quad sonst \\
H_{Not-in} & = & \{((a, e), l) | \forall (a', e') \in l, e \neq e'\} \\
H_{Kons} & = & \{l | \forall (a, e), (a', e') \in l, e \neq e'\}
\end{array}
$$

.

3 Konsistenznachweis

In diesem Abschnitt wird die Korrektheit der dynamischen Methoden, die im zweiten Abschnitt angegeben worden sind, gezeigt.
Eine Methode ist persistent, falls die Methode die Konsistenzbedingung durch Transformation zwischen zwei Zustände nicht verletzt. D.h. ein persistenter Zustand bleibt nach der Transformation immer noch persistent. Zunächst wird die Definition der persistenten Zustände betrachtet, die durch die Erweiterung des Modells um ein "environment" entstehen.

Definition 3.1 (persistente Zustände) *Ein Zustand A^{st} auf einem Basismodell A ist* **persistent**, *wenn der Zustandsmorphismus $\eta^{st} : A \to A^{st}$ bijektiv ist und $\eta^{st}(dom(A_\sigma)) = dom(A^{st}_\sigma)$ für alle $\sigma \in OP$, $\sigma \neq\ !_s$ $(s \in dom(ref))$ gilt.*

Satz 3.2 (Charakterisierung von persistenten Zuständen) *Sei $(A, e) \in Mod(BaseSPEC\ )$ wobei*

- *e die Interpretation von ! ist*
- *$dom(e) = \{d_1, ..., d_n\}$*
- *$e(d_i) = a_i, i = 1, ..., n$*

dann ist

- $(A, e) \cong A^{[\langle d_1,\ldots d_n\rangle := \langle a_1,\ldots,a_n\rangle]}$ *und*
- *(A,e) ist persistent*

Definition 3.3 (persistente Methode) *Ein Methodenausdruck ist persistent, wenn folgendes gilt: sei* $A^{st_1} \xrightarrow{met} A^{st_2}$ *eine Transition gemäß Def. 1.7 und* A^{st_1} *ein persistenter Zustand, dann ist auch* A^{st_2} *ein persistenter Zustand.*

3.1 Konsistenznachweis der Methode Create-Aufg

Im folgenden wird die Methode Create-Aufg betrachtet. Diese Methode kann als eine Transformation zwischen zwei Zuständen interpretiert werden. Zum Beispiel wird ein neuer Auftraggeber (Meier, Berlin) unter der Referenznummer 4 mit der Methode Create-Aufg(4; Meier,Berlin) erzeugt, falls die Referenznummer 4 nicht bereits vergeben ist.
Die Definition der Methode ist gemäß 1.5 und dem Beispiel auf Seite 13 wie folgt gegeben.

- $v = ref(Aufg), Name, Adresse \in S^*$
- $r_1 = r_2 = \langle\rangle$
- $m = Create - Aufg$
- $\Delta = p$ eine Referenz zur Sorte $ref(Aufg)$
- $t = na, ad \in Term(BaseSPEC\ )$
- $r = cons(na, ad) \in Term(BaseSPEC\ )$

Die Methode wird gemäß Def. 1.7 ausgewertet

- $A^{st} = A^{[d_u := a_u]} = (A, e_0)$ sei ein persistenter Zustand
- mit $4 \notin d_u$
- Create-Aufg(p; na,ad) := cons(na,ad) Definition der Methode
- a_v = (4; Meier, Berlin), Belegung der Variablen
- $A^{[d_u := a_u]} = (A, e_0)$ erfüllt die leere Bedingung
- b_w= (Meier, Berlin)

damit ist

- $A^{[d_u := a_u]} \xrightarrow{create-Aufg(4;(Meier,Berlin))} A^{[\langle 4,d_u\rangle := \langle (Meier,Berlin),a_u\rangle]}$

eine Transition.

Da $A^{[\langle 4,d_u\rangle := \langle (Meier,Berlin),a_u\rangle]} = (A, 4 \mapsto (Meier, Berlin)) \in Mod(BaseSPEC\ )$ mit $(4 \mapsto (Meier, Berlin)) =: e_1$, $dom(e_1) = \{4\}$ und $e_1(4) = (Meier, Berlin)$ gilt, ist der neue Zustand persistent.

3.2 Konsistenznachweis der Methode Add

Zunächst wird das Hat-Basismodell betrachtet. Da im Basismodell $H_! = \emptyset$ ist, existiert keine Liste von Referenzpaaren. Das Basismodell wird mit "environments" erweitert. Im folgenden wird die Methode Add gemäß Def 1.7 ausgewertet. Es gibt zwei Fälle zu betrachten:

- 1.Fall: $!Hat = []$
  - Der Zustand $(H, Hat \longmapsto \lambda) = (H, e_0)$ ist persistent, da die leere Liste die Konsistenzbedingung erfüllt.
  - Die Methodendefinition ist gegeben durch:

    Not-in((!a,!e),Deref(!Hat)) $\rightarrow$ Add(Hat; (a,e)) := (a,e) :: !Hat
  - Ein Element $h \in (N \times N)^*$ d.h. $h = (a, e)$
  - (H, e_0) erfüllt die Bedingung Not-in((!a,!e),Deref(!Hat)). Not-in liefert True, falls die Liste leer ist.
  - b_w = ((a,e))

  damit ist

  - $H^{[Hat:=\lambda]} \xrightarrow{Add(Hat;((a,e))} H^{[Hat:=((a,e))]}$

  eine Transition.

  Nun muß gezeigt werden, daß $(H, e_1) := (H, Hat \mapsto ((a, e))) \models Kons(Deref(!Hat))$. !Hat = ((a,e)), !a, !e und Deref([]) sind definiert. Die Funktion Deref liefert unter diesen Voraussetzungen die dereferenzierte Liste $((!a, !e))$. Nach obigen Voraussetzungen gilt Not-in((!a,!e),Deref(!Hat)). Wegen der Definition von H_{Not-in} und H_{Kons} folgt :

  - $\forall (a_i, e_i) \in !Hat, (a, e) \in !Hat, !e_i \neq !e$

  D.h.

  - $((!a, !e)) \in H_{Kons}$

Es gilt:

$(H, e_1) := (H, Hat \mapsto ((a, e))) = H^{[Hat:=((a,e))]} \in Mod(BaseSPEC\ )$ mit $e_1(Hat) = ((a, e))$ und $dom(e_1) = \{Hat\}$

- 2.Fall: $!Hat = ((a_1, e_1), ..., (a_n, e_n))$
  - Der Zustand $(H, e_0) = (A, Hat \longmapsto ((a_1, e_1), ..., (a_n, e_n))$ mit $dom(e_0) = \{Hat\}$ und $e_0(Hat) = ((a_1, e_1), ..., (a_n, e_n))$ ist konsistent. D.h. dieser Zustand erfüllt die Bedingung $Kons(Deref(!Hat))$
  - Die Methodendefinition ist gegeben durch:

    Not-in((!a,!e),Deref(!Hat)) $\rightarrow$ Add(Hat; (a,e)) := (a,e)::!Hat
  - (A, e_0) erfüllt die Bedingung Not-in((!a,!e),Deref(!Hat)). D.h. entweder ist das Paar nicht in der Liste vorhanden oder sein Empfänger mit keinem anderen Empfänger in der Liste gleich
  - $b_w = ((a, e), (a_1, e_1), ..., (a_n, e_n))$

  damit ist

  - $H^{[Hat:=((a_1,e_1),(a_2,e_2),...(a_n,e_n))]} \xrightarrow{Add(Hat;(a,e))} H^{[Hat:=((a,e),(a_1,e_1),(a_2,e_2),...(a_n,e_n))]}$

  eine Transition.

  Zu zeigen ist: $(H, e') := (H, Hat \mapsto ((a, e), (a_1, e_1), (a_2, e_2), ...(a_n, e_n))) \models Kons(Deref(!Hat))$ Nach der Voraussetzung gilt $!Hat = ((a, e), (a_1, e_1), (a_2, e_2), ..., (a_n, e_n))$ und $\forall i \in \{1, ..., n\}, !a_i, !e_i$ und !a,!e sind definiert. D.h. es existiert eine dereferenzierte Liste: $((!a, !e), (!a_1, !e_1), (!a_2, !e_2), ..., (!a_n, !e_n))$. Aus Not-in((!a,!e),Deref(!Hat)) folgt:

  - $\forall (a_i, e_i) \in !Hat, !e_i \neq !e$

  D.h.

  - $((!a, !e), (!a_1, !e_1), (!a_2, !e_2), ..., (!a_n, !e_n)) \in H_{Kons}$.

Es gilt also:

$(H, e') := (H, Hat \mapsto ((a, e), (a_1, e_1), (a_2, e_2), ...(a_n, e_n)) = H^{[Hat:=((a,e),(a_1,e_1),(a_2,e_2),...(a_n,e_n))]} \in Mod(BaseSPEC\ )$ mit $e'(Hat) = ((a, e), (a_1, e_1), (a_2, e_2), ...(a_n, e_n))$ und $dom(e') = \{Hat\}$.

4 Schlußbemerkung

Ein wichtiger Vorteil der Transitionsspezifikationen ist die Möglichkeit, Spezifikationen und Algorithmen miteinander zu vergleichen. Die Konsistenzbeweisen bieten eine gute Möglichkeit, mögliche Fehler in den Spezifikationen zu finden und zu beseitigen. Der Zeitaufwand dieses Formalismus ist im Vergleich zu algebraischen Spezifikationsmethoden geringer.

Der Umgang mit Transitionsspezifikationsmethoden ist nicht so einfach zu erlernen. Dies ist ein sehr abstrakter Formalismus. Ihr Einsatz in der Praxis würde qualifizierte Spezifikateure erfordern.

Literatur

[GR95] Martin Große-Rhode. Specification of Transition Categories, An Approach to Dynamic Abstract Data Types, Dissertation, TU Berlin, 1995.

[EM85] H. Ehrig, B. Mahr. Fundamentals of Algebraic Specification 1: Equations und Initial Semanitics, volume 6 of EATCS Monographs on Theoretical Computer Science. Springer, Berlin, 1985.

[CGW92] I. Claßen, M. Große-Rhode und U. Wolter. Categorical concepts for parameterized partial specifications. Technical report 92-46, Technische Universität Berlin, 1992.

[VI94] Verena Isenmann. Ansätze zur Integration semiformaler und formaler Spezifikationsmethoden für den Softwareentwurf. Diplomarbeit am Institut für Theoretische Informatik der Technischen Universität Berlin, 1994.

[EB94] H. Ehrig, R. Bardohl. Specification Techniques using Dynamic Abstract Data Types and Application to Shipping Software. Technical University of Berlin, Proceedings Shanghai Workshop "Advanced Software Engineering", 1994.

[BB94] R. Bardohl, B. Bardohl. Allgemeines über Speditionen. Interne Notiz, TU Berlin, 1994.

[GR92] Martin Große-Rhode. Kategorielle und Graphische Algebra. Manuskript, TU Berlin, 1992.