



**TGraphen und EER-Schemata
formale Grundlagen**

Peter Dahm, Jürgen Ebert, Angelika
Franzke, Manfred Kamp, Andreas Winter

12/98



Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz

E-mail: researchreports@infko.uni-koblenz.de,

WWW: <http://www.uni-koblenz.de/fb4/>

TGraphen und EER-Schemata formale Grundlagen

Peter Dahm
Jürgen Ebert
Angelika Franzke
Manfred Kamp
Andreas Winter

10. August 1998

Zusammenfassung

In diesem Papier wird der Umfang und die Semantik von EER-Schemata im Rahmen des EER/GRAL-Ansatzes formal spezifiziert. Dies schließt eine Spezifikation des Konzepts TGraph ein. Die Spezifikationssprache ist \mathcal{Z} .

Die Semantik eines EER-Schemas wird definiert durch die Menge der zu ihm passenden TGraphen. Dies geschieht letztendlich durch die Definition einer Relation, die festlegt, wann genau ein gegebener TGraph zu einem gegebenen EER-Schema paßt.

1 Einleitung

Mit dem EER/GRAL-Ansatz werden in verschiedenen Kontexten Objekt-Beziehungs-Geflechte modelliert und realisiert [Ebert et al., 1996]. Die Modellierung geschieht durch die Angabe eines *EER/GRAL-Schemas*. Dieses definiert eine Menge von zu diesem Schema passenden *TGraphen* [Ebert/Franzke, 1995], eine sog. *Graphklasse*. TGraphen (also die Instanzen eines EER/GRAL-Schemas) sind typisierte, gerichtete, attributierte, angeordnete Graphen, d.h. den Knoten und Kanten sind Typen und Attributwerte zugeordnet, die Kanten sind gerichtet und die zu einem Knoten inzidenten Kanten haben eine feste Ordnung.

Ein EER/GRAL-Schema besteht aus zwei Komponenten:

- In einem *EER-Schema*¹ werden Eigenschaften beschrieben, wie sie typischerweise durch EER-Diagramme ausgedrückt werden, z.B. die vorkommenden Objekt- und Beziehungstypen und deren Attributierung, die Inzidenzbeziehungen zwischen Objekten und Beziehungen, die Vererbungshierarchie auf Typen usw.

Auch im EER/GRAL-Ansatz wird ein EER-Schema durch ein *EER-Diagramm* notiert.

¹EER = *Extended Entity Relationship*

- Durch die Anotation von *GRAL-Zusicherungen* [Franzke, 1997] werden zusätzlich Eigenschaften der Schema-Instanzen beschrieben, die nicht durch ein EER-Diagramm ausgedrückt werden können, z.B. strukturelle Eigenschaften wie Kreisfreiheit.

Der Ansatz eröffnet die Möglichkeit eines nahtlosen Vorgehens vom Schema-Entwurf bis zur Implementation. In [Ebert et al., 1996] findet sich ein Überblick über den EER/GRAL-Ansatz und seine Anwendungsmöglichkeiten in den verschiedensten Kontexten mit entsprechenden Beispielen.

In diesem Papier wird die Semantik von EER-Schemata im Sinne des EER/GRAL-Ansatzes formal spezifiziert. Wie im gesamten EER/GRAL-Ansatz wird auch hier für die formale Spezifikation die Sprache \mathcal{Z} [Spivey, 1992] verwendet. Es werden aber keine Aussagen zur konkreten Notation eines EER-Schemas in Form eines Diagramms getroffen. Das Papier ist als Referenzpapier zur formalen Absicherung des Ansatzes gedacht und enthält demzufolge keine Beispiele.

Die Grundidee der folgenden Spezifikation ist, die Semantik eines EER-Schemas durch die Angabe der Menge von Graphen, die zu diesem EER-Schema „passen“, zu beschreiben.

Dazu wird zunächst in Abschnitt 2 eine Spezifikation des Konzepts TGraph in Form eines (\mathcal{Z} -)Schema *TGraph* angegeben.

In Abschnitt 3 wird schließlich auf der Grundlage der TGraphspezifikation das Schema *EERSpecification* entwickelt, welches letztendlich beschreibt, was genau ein EER-Schema (im EER/GRAL-Sinn) beinhaltet.

Abschließend wird dann in Abschnitt 4 die Semantik einer *EERSpecification* durch die Angabe der zu ihr passenden Graphen beschrieben. Dies wird durch eine Relation $\models_{EERSpecification}$ operationalisiert, durch die festgelegt ist, ob ein gegebener Graph eine gültige Instanz zu einem gegebenen EER-Schema ist.

2 TGraphen

TGraphen sind typisierte, attributierte, gerichtete, angeordnete Graphen (vgl. [Ebert/Franzke, 1995], [Franzke, 1997]), d.h.:

- Jeder Knoten und jede Kante hat einen Typ.
- Knoten und Kanten können attribuiert sein.
- Die Kanten sind gerichtet.
- Die Kanten sind angeordnet, d.h. auf den zu einem Knoten inzidenten Kanten existiert eine Ordnung.

Für die Spezifikation von TGraphen werden die Grundmengen *Id* und *Value* als gegeben vorausgesetzt.

$[Id, Value]$

Die Grundmenge *Id* steht für eine Menge gültiger Bezeichner. Sie wird an verschiedenen Stellen verwendet. Ist es erforderlich, daß Bezeichner für bestimmte Kontexte disjunkt sind, so ist dies explizit angegeben.

Value symbolisiert die Menge aller möglichen Werte, die Attributen zugeordnet werden können. *Value* wird hier nur vorläufig als Grundmenge eingeführt. Für eine konkrete Implementierung muß *Value* genauer beschrieben werden. Anhang A.1 enthält die entsprechende Konkretisierung für die im Graphenlabor [Dahm/Widmann, 1998] implementierte Wertewelt.

Graphenelemente, also Knoten (*VERTEX*) und Kanten (*EDGE*), werden in der Spezifikation zusammenfassend als *ELEMENT* bezeichnet. Sie werden in der folgenden freien Typdefinition eingeführt. Die natürlichen Zahlen werden verwendet, um eine Identifizierbarkeit der Graphenelemente zu garantieren, eine Ordnung über Knoten und Kanten wird damit *nicht* unterstellt.

$$ELEMENT ::= vertex\langle\mathbb{N}\rangle \mid edge\langle\mathbb{N}\rangle$$

$$VERTEX == \text{ran } vertex$$

$$EDGE == \text{ran } edge$$

Mit *Dir* werden Konstanten spezifiziert, die zur Symbolisierung von Kantenrichtungen (in einen Knoten eingehend bzw. von einem Knoten ausgehend) verwendet werden. *TypeId* und *AttrId* werden als Bezeichner eingeführt, um später Typ- und Attributbezeichner unterscheiden zu können. Ein *AttributeInstanceSet*, also eine Zuordnung von Werten zu Attributbezeichnern, beschreibt die Attributierung eines Graphenelements durch Bezeichner-Wert-Paare.

$$Dir ::= in \mid out \mid anyDir$$

$$TypeId == Id$$

$$AttrId == Id$$

$$AttributeInstanceSet == AttrId \mapsto Value$$

Im Schema *TGraph* werden nun die folgenden Eigenschaften von TGraphen spezifiziert:

- TGraphen bestehen aus einer endlichen, injektiven Folge von Knoten (*V*) und einer endlichen, injektiven Folge von Kanten (*E*).
- Jedem Knoten ist eine (möglicherweise leere) Liste von Kanten zugeordnet (Λ und [p1]), die in den Knoten entweder ein- oder ausgehen. Jede Kante taucht in genau einer dieser Listen als eingehend und in genau einer als ausgehend auf [p2]. Dies darf in beiden Fällen dieselbe Liste sein. (Die Kante bildet dann eine Schlinge im Graph).
- Jedem Knoten und jeder Kante ist genau ein Typ zugeordnet (*type* und [p3]).
- Jedem Knoten und jeder Kante ist eine (möglicherweise leere) endliche Menge von Attribut-Wert-Paaren zugeordnet (*value* und [p4]).

TGraph

$V : \text{iseq } VERTEX$

$E : \text{iseq } EDGE$

$\Lambda : VERTEX \rightsquigarrow \text{seq}(EDGE \times Dir)$

$type : ELEMENT \rightsquigarrow TypeId$

$value : ELEMENT \rightsquigarrow AttributeInstanceSet$

$\Lambda \in \text{ran } V \rightarrow \text{iseq}(\text{ran } E \times \{in, out\})$ [p1]

$\forall e : \text{ran } E \bullet \exists_1 v, w : \text{ran } V \bullet (e, in) \in \text{ran}(\Lambda(v)) \wedge (e, out) \in \text{ran}(\Lambda(w))$ [p2]

$\text{dom } type = \text{ran } V \cup \text{ran } E$ [p3]

$\text{dom } value = \text{ran } V \cup \text{ran } E$ [p4]

In diesem Schema wird ein Graphkonzept beschrieben, das an einigen Stellen allgemeiner ist als typische Instanzen von EER-Schemata. So wird z.B. *nicht* verlangt, daß Knoten und/oder Kanten desselben Typs auch die gleiche Attributstruktur haben. Auch können Knoten und Kanten denselben Typ haben. Diesbezügliche Einschränkungen werden erst nötig, wenn ein TGraph als Instanz eines EER-Schemas betrachtet werden soll (vgl. Abschnitt 4).

3 EER-Schemata

3.1 Überblick

In diesem Abschnitt wird beschrieben, was genau ein EER-Schema im EER/GRAL-Sinn ist, welche Bestandteile und Eigenschaften es hat und damit natürlich, welche Modellierungsmöglichkeiten es bietet. Diese sind zunächst einmal unabhängig von der intendierten Graphklassen-Semantik. Aus diesem Grund wird in diesem Abschnitt auch eine reine ER-Terminologie verwendet².

Grundsätzlich entsprechen die hier zu spezifizierenden EER-Schemata „üblichen“ EER-Beschreibungen. Es gibt (disjunkte) Entity- und Relationshiptypen, denen Attributschemata zugeordnet und die in einer Typhierarchie angeordnet sind. Es ist möglich, durch eine Sonderform der Relationshiptypen, den Rollentyp, Aggregationsbeziehungen auszudrücken.

Einen echten Konsens über die Eigenschaften von EER-Beschreibungen gibt es in der Softwaretechnik offenbar nicht. Die oben aufgezählten Punkte stellen eher so etwas wie einen kleinsten gemeinsamen Nenner dar. Zusätzlich werden in allen EER-Ansätzen Grundsatzentscheidungen getroffen, die recht unterschiedlich ausfallen können (Soll es Mehrfachvererbung geben? Soll es einen „obersten“ Typ geben? etc.). Für die EER-Schemata hier sind die wichtigsten dieser Entscheidungen in der folgenden Liste zusammengefaßt:

- Eine Typhierarchie existiert sowohl auf Entity- als auch auf Relationshiptypen.

²Bezogen auf TGraphen wird hier der Begriff „Entity“ synonym mit Knoten und „Relationship“ gleichbedeutend mit Kante verwendet.

- Es gibt in jedem EER–Schema einen obersten Entity- und einen obersten Relationship. Alle anderen Typen sind Subtypen dieses Typs.
- Relationships können attribuiert sein.
- Mehrfachvererbung ist zulässig.
- Vererbung von Attributen muß grundsätzlich konfliktfrei sein, d.h. wenn für einen Super- und einen Subtyp gleichnamige Attribute definiert werden, müssen sie den gleichen Wertebereich haben. Gleiches gilt, falls in mehreren Supertypen eines Typs gleichnamige Attribute definiert sind.
Ebenso darf durch die Vererbung von Kardinalitätsanforderungen kein Konflikt entstehen.
- Sowohl Entity- als auch Relationshipstypen können als „abstrakt“ ausgezeichnet werden. Von diesen Typen dürfen dann in einer Instanz des EER–Schemas keine Exemplare existieren.
- Relationshipstypen können als „injektiv“ ausgezeichnet werden. In Instanzen des EER–Schemas darf es dann keine zwei Beziehungen dieses Typs mit denselben Anfangs- und Endknoten (Doppelbeziehungen) geben.

In der folgenden Spezifikation sind die durch ein EER–Schema beschreibbaren Sachverhalte in drei aufeinander aufbauende Teile gegliedert.

- Im *Typsystem* werden die zulässigen Typen, deren Attributierung und die Vererbungshierarchie festgelegt.
- Im *Inzidenzsystem* wird mit Hilfe des Typsystems beschrieben, welche Relationships welche Entitys in Beziehung setzen dürfen.
- Im *Invariantensystem* schließlich werden aufbauend auf dem Inzidenzsystem Aussagen zur Kardinalität und Injektivität von Relationships gemacht.

Die folgende \mathcal{Z} –Spezifikation von EER–Beschreibungen ist natürlich vor dem Hintergrund der intendierten Semantik in Bezug auf Instanzgraphen (vgl. Abschnitt 4) entstanden. Die Anforderungen an ein konkretes EER–Schema sind trotzdem bewußt schwach gehalten. Sie garantieren zwar eine sinnvolle Interpretation jedes gültigen EER–Schemas, aber sie verhindern nicht „sinnlose“ EER–Schemata, deren einzig mögliche Instanz der leere Graph ist. Dies wäre z.B. der Fall, wenn ausschließlich abstrakte Typen definiert würden.

3.2 Typsystem

Im Typsystem (Schema *TypeSystem*) wird festgelegt, welche Typen es im EER–Schema gibt, wie sie attribuiert und wie sie in der Typhierarchie angeordnet sind.

Für die Festlegung der Attributierung wird die Menge der Attributschemata (*Attributschema*) und die Menge der Wertebereiche (*Domain*) benötigt.

Ein Attributschema ist definiert als (endliche) Abbildung von Attributbezeichnern auf Wertebereiche. Jeder Wertebereich beschreibt dabei eine Menge möglicher Attributwerte, die dem jeweiligen Attributbezeichner auf der Instanzebene zugewiesen werden können. Wie die Menge *Value* der möglichen Attributwerte muß auch *Domain* für eine konkrete Implementierung genauer beschrieben werden. Anhang A.2 enthält die entsprechende Konkretisierung für die im Graphenlabor implementierte Wertewelt.

[*Domain*]
 $AttributeSchema == AttrId \mapsto Domain$

Ferner verwendet die Definition von Typsystemen zwei globale Größen, nämlich den Obertyp (*Entity*) zu allen Entitytypen sowie den Obertyp *Relationship* zu allen Relationshiptypen.

<i>Entity</i> : <i>TypeId</i>
<i>Relationship</i> : <i>TypeId</i>
<i>Entity</i> \neq <i>Relationship</i>

Nach diesen Präliminarien werden Typsystem wie folgt definiert:

<i>TypeSystem</i>	
$types : \mathbb{F} TypeId$	
$abstractTypes : \mathbb{F} TypeId$	
$entityTypes : \mathbb{F} TypeId$	
$relationshipTypes : \mathbb{F} TypeId$	
$roleTypes : \mathbb{F} TypeId$	
$typeDefinitionSet : TypeId \mapsto AttributeSchema$	
$(_isA_) : TypeId \leftrightarrow TypeId$	
$abstractTypes \subseteq types$	[p1]
$\langle entityTypes, relationshipTypes \rangle$ partition $types$	[p2]
$roleTypes \subseteq relationshipTypes$	[p3]
$dom\ typeDefinitionSet = types$	[p4]
$(_isA_) \subseteq ((entityTypes \times entityTypes) \cup (relationshipTypes \times relationshipTypes))$	[p5]
$\forall r1, r2 : relationshipTypes \mid r1\ isA\ r2 \bullet r2 \in roleTypes \Rightarrow r1 \in roleTypes$	[p6]
$\forall t, t1, t2 : types \mid t\ isA^*\ t1 \wedge t\ isA^*\ t2 \bullet$ $\neg \exists a : AttrId \mid a \in dom\ typeDefinitionSet(t1) \cap dom\ typeDefinitionSet(t2) \bullet$ $typeDefinitionSet(t1)(a) \neq typeDefinitionSet(t2)(a)$	[p7]
$Entity \in entityTypes$	[p8]
$Relationship \in relationshipTypes$	[p9]
$\forall e : entityTypes \bullet e\ isA^*\ Entity$	[p10]
$\forall r : relationshipTypes \bullet r\ isA^*\ Relationship$	[p11]

Ein Typsystem besteht zunächst einmal aus einer Menge von Typbezeichnern (*types*). Eine Teilmenge davon kann abstrakt sein (*abstractTypes*) [p1].

Die Menge der Typen zerfällt in Entitytypen (*entityTypes*) und Relationshiptypen (*relationshipTypes*) [p2]. Von den Relationshiptypen wiederum kann für Rollentypen (*roleTypes*) eine Teilmenge gebildet werden. Damit ist jeder Rollentyp auch immer ein Relationshiptyp [p3].

Sowohl Entity- als auch Relationshiptypen sind attribuiert. Dazu wird *jedem* Typ durch die Abbildung (*typeDefinitionSet*) ein (möglicherweise leeres) Attributschema (*AttributeSchema*) zugeordnet [p4].

Typen können in einer Subtyp-Beziehung zueinander stehen. Dazu wird im Schema *TypeSystem* die Relation *_isA_* eingeführt. Es können nur Entity- bzw. Relationshiptypen untereinander in dieser Beziehung stehen [p5]. Des weiteren wird verlangt, daß alle Subtypen von Rollentypen wiederum Rollentypen sind [p6]. Ein Typ kann beliebig viele Supertypen haben. Ein Zyklus in der Subtyp-Relation ist nicht ausgeschlossen.

Für die Subtyp-Beziehung wird ferner verlangt, daß sie bzgl. der Attributierung konfliktfrei ist [p7]. D.h. hier, wenn bei zwei Typen, die in einer (auch indirekten) Subtyp-Beziehung zueinander stehen, ein gleichnamiges Attribut definiert ist, so muß diesem Attribut auch jeweils der genau gleiche Wertebereich zugeordnet sein.

Alle Entitytypen haben den gemeinsamen Supertyp „Entity“ [p10]. Alle Relationshiptypen haben den gemeinsamen Supertyp „Relationship“ [p11]. Diese Typen sind Bestandteil jedes EER-Schemas [p8, p9].

3.3 Inzidenzsystem

Im Inzidenzsystem (Schema *IncidenceSystem*) wird beschrieben, welche Entitytypen durch einen Relationshiptyp miteinander in Beziehung gesetzt werden (*relates*). Dazu muß natürlich auf das Typsystem zurückgegriffen werden.

<i>IncidenceSystem</i>	
<i>TypeSystem</i>	
$relates : TypeId \mapsto (TypeId \times TypeId)$	
$aggregatesForward : \mathbb{F} TypeId$	
$dom\ relates = relationshipTypes$	[p1]
$ran\ relates \subseteq entityTypes \times entityTypes$	[p2]
$\forall r1, r2 : relationshipTypes \mid r1\ isA^*\ r2 \bullet$	[p3]
$\quad first(relates(r1))\ isA^*\ first(relates(r2))$	
$\quad \wedge\ second(relates(r1))\ isA^*\ second(relates(r2))$	
$aggregatesForward \subseteq roleTypes$	[p4]
$\forall r1, r2 : roleTypes \mid r1\ isA^*\ r2 \bullet$	[p5]
$\quad r1 \in aggregatesForward \Leftrightarrow r2 \in aggregatesForward$	

Es wird gefordert, daß die Subtyp-Relationen der Relationshiptypen und der durch sie verbundenen Entitytypen zueinander „passen“. Konkret heißt das: Wird von einem Relationshiptyp

ein Subtyp gebildet, so muß der Start-Entitytyp des Subtyps wiederum ein Subtyp des (oder gleich dem) Start-Entitytyp des Supertyps sein. Für die Ziel-Entitytypen gilt dies analog [p3].

Bei Rollentypen spielen die beiden in Beziehung gesetzten Entitytypen unterschiedliche Rollen (Aggregat bzw. Komponente). Im Inzidenzsystem kann festgelegt werden, ob der Start- oder der Ziel-Entitytyp des Rollentyps die Rolle des Aggregats spielt (*aggregatesForward*) [p4]. Gehört ein Rollentyp der Menge *aggregatesForward* an, so stellt der Ziel-Entitytyp das Aggregat dar (der Rollentyp „aggregiert in Beziehungsrichtung“), sonst der Start-Entitytyp. Werden von einem Rollentyp Subtypen gebildet, übernehmen sie diese Eigenschaft [p5].

3.4 Invariantensystem

Im *Invariantensystem* (Schema *InvariantSystem*) werden Aussagen zur Kardinalität und Injektivität von Relationships gemacht.

Infinity : \mathbb{N}_1 ³

<i>Cardinality</i>
<i>min</i> : \mathbb{N}
<i>max</i> : \mathbb{N}_1
$min \leq max \leq Infinity$

Eine Kardinalitätsanforderung (Schema *Cardinality*) wird durch ein Paar von natürlichen Zahlen dargestellt, die eine Unter- bzw. Obergrenze angeben. Dabei muß die Obergrenze größer oder gleich der Untergrenze sein.

<i>InvariantSystem</i>
<i>IncidenceSystem</i>
<i>limits</i> : <i>TypeId</i> \leftrightarrow (<i>Cardinality</i> \times <i>Cardinality</i>)
<i>injective</i> : \mathbb{F} <i>TypeId</i>
<i>dom limits</i> = <i>relationshipTypes</i> [p1]
<i>injective</i> \subseteq <i>relationshipTypes</i> [p2]
$\forall r1, r2 : relationshipTypes \mid r1 \text{ isA}^* r2 \bullet r2 \in injective \Rightarrow r1 \in injective$ [p3]

Jedem Relationshiptyp wird ein Paar von Kardinalitäten zugeordnet (*limits*) [p1]. Diese geben letztendlich an, wieviele Beziehungen dieses Typs ein Entity vom Start- bzw. Zieltyp des Relationshiptyps eingehen darf. Auf die Angabe von zusätzlichen Bedingungen für Kardinalitäten im Zusammenhang mit der Subtyp-Relation wird hier bewußt verzichtet. Dies führt dazu,

³Diese Konstante wird aus pragmatischen Gründen eingeführt. Sie symbolisiert den höchsten Wert, der als Maximumwert einer Kardinalität angegeben werden kann. Eigentlich müßte hierfür ein ∞ -Symbol eingeführt werden. Dies würde aber zum einen die weitere Spezifikation unangemessen erschweren. In einer Implementation muß für *Infinity* ein konkreter Wert festgelegt werden.

daß EER–Spezifikationen möglich sind, in denen es, nach der im folgenden Kapitel definierten Semantik, zu bestimmten Entitytypen keine Instanzen geben kann (z.B. könnte man für einen Entitytyp genau eine ausgehende Beziehung eines bestimmten Typs fordern, für einen seiner Subtypen aber genau zwei). Trotzdem sind solche EER–Beschreibungen noch korrekt interpretierbar und werden deshalb hier nicht ausgeschlossen.

Außer den Kardinalitäten kann für Relationstypen noch festgelegt werden, daß sie „injektiv“ sind, daß also keine zwei Beziehungen dieses Typs die gleichen Entitäts in der gleichen Richtung verbinden dürfen [p2]. Ist ein Relationstyp injektiv, so müssen dies auch alle seine Subtypen sein [p3].

3.5 EER–Schema

Ein komplettes EER–Schema (Schema $EERSchema$) besteht aus einem Invariantensystem (welches ja ein Inzidenz- und damit auch ein Typsystem enthält) und einem Bezeichner.

$EERSchema$ <i>InvariantSystem</i> <i>name : Id</i>

4 Semantik von EER–Schemata

In diese Abschnitt wird nun die Semantik von EER–Schemata spezifiziert. Durch ein EER–Schema wird *eine Menge von TGraphen (Graphklasse)* definiert. Diese wird gebildet durch diejenigen Graphen, die zu dem EER–Spezifikation „passen“. Ein Graph, der zu einem EER–Schema paßt, ist ein Element der durch das Schema definierten Graphklasse.

Im folgenden wird auch die terminologische Brücke zwischen der Welt der EER–Beschreibungen und der Welt der TGraphen geschlagen: Knoten werden als Instanzen von Entitytypen, also als Objekte bzw. Entities, und Kanten als Instanzen von Relationstypen, also als Beziehungen bzw. Relationships, betrachtet.

Die Relation $- \models_{EERSchema} -$ bestimmt, wann genau ein Graph zu einem Schema paßt. Deren Spezifikation ist im folgenden analog zum vorherigen Kapitel aufgebaut, d.h. sie orientiert sich an der Aufteilung in Typsystem, Inzidenzsystem und Invariantensystem.

Diese Relation erlaubt es auch, sehr einfach die *Semantik* $\mathcal{D}_{EERSchema}$ einer EER–Beschreibung formal anzugeben.

$\mathcal{D}_{EERSchema} : EERSchema \rightarrow \mathbb{P} TGraph$ $\forall eerSpec : EERSchema \bullet$ $\mathcal{D}_{EERSchema}[[eerSpec]] = \{g : TGraph \mid g \models_{EERSchema} eerSpec\}$
--

4.1 Typsystem

Ein Graph paßt zu einem Typsystem, wenn der Typ jedes Knotens in den Entitytypen [p1] und der Typ jeder Kante in den Relationshiptypen [p2] enthalten ist. Von abstrakten Typen darf es keine Instanzen geben, d.h. kein Knoten und keine Kante kann einen Typ haben, der als abstrakt definiert ist [p3].

Außerdem muß die Attributbelegung ($value(elem)$) für jeden Knoten und jede Kante mit dem Attributierungsschema des entsprechenden Typs ($typeDefinitionSet(t)$) und seiner Obertypen übereinstimmen [p4]. Durch diese Forderung wird die *Vererbung von Attributen* realisiert. In der folgenden axiomatischen Definition „sammelt“ darum der Ausdruck

$$\bigcup \{t : types \mid G.type(elem) \text{ isA}^* t \bullet typeDefinitionSet(t)\}$$

die Attributdefinitionen des fraglichen Typs und aller seiner Obertypen ein. Dabei liefert $typeDefinitionSet(t)$ zu einem Typ t dessen *AttributeSchema*, also eine Menge von Paaren von Attributbezeichnern und Wertebereichen. Der gesamte Ausdruck liefert also die Vereinigung dieser Paarmengen für den Typ des Knotens bzw. der Kante $elem$ und aller seiner Supertypen und somit wieder ein *AttributeSchema*.

$(- \models_{TypeSystem} -) : TGraph \leftrightarrow TypeSystem$	
$\forall G : TGraph; TypeSystem \bullet$	
$G \models_{TypeSystem} \theta TypeSystem \Leftrightarrow$	
$\forall v : \text{ran } G.V \bullet type(v) \in entityTypes$	[p1]
$\forall e : \text{ran } G.E \bullet type(e) \in relationshipTypes$	[p2]
$\forall elem : \text{ran } G.V \cup \text{ran } G.E \bullet G.type(elem) \notin abstractTypes$	[p3]
$\forall elem : \text{ran } G.V \cup \text{ran } G.E \bullet$	[p4]
$G.value(elem) \models_{AttributeSchema}$	
$\bigcup \{t : types \mid G.type(elem) \text{ isA}^* t \bullet typeDefinitionSet(t)\}$	

In obiger Definition wird [p4] die Relation ($- \models_{AttributeSchema} -$) verwendet, die definiert, wann eine Attributbelegung zu einem Attributierungsschema paßt. Dies ist der Fall, wenn in der Belegung jedem Attributbezeichner aus dem Attributierungsschema ein Wert zugeordnet ist und die Attributbelegung keine weiteren Attributbezeichner–Wert–Paare enthält. Ein Wert muß aus dem Wertebereich stammen, der dem jeweiligen Attributbezeichner im Attributierungsschema zugeordnet ist. Welche Werte für einen bestimmten Wertebereich zugelassen sind, definiert die Funktion $valueSetOf$. Für eine konkrete Implementierung muß diese Funktion passend zur Festlegung der Mengen *Value* und *Domain* näher beschrieben werden. Anhang A.3 enthält die entsprechende Konkretisierung für die im Graphenlabor implementierte Wertewelt.

$$\begin{array}{l} \text{valueSetOf} : \text{Domain} \rightarrow \mathbb{P} \text{Value} \\ \hline (- \models_{\text{AttributeSchema}} -) : \text{AttributeInstanceSet} \leftrightarrow \text{AttributeSchema} \end{array}$$

$$\begin{array}{l} \forall \text{inst} : \text{AttributeInstanceSet}; \text{def} : \text{AttributeSchema} \bullet \\ \quad \text{inst} \models_{\text{AttributeSchema}} \text{def} \Leftrightarrow \\ \quad \text{dom inst} = \text{dom def} \\ \quad \forall \text{attr} : \text{dom inst} \bullet \\ \quad \quad \text{inst}(\text{attr}) \in \text{valueSetOf}(\text{def}(\text{attr})) \end{array}$$

4.2 Inzidenzsystem

Zu einem Inzidenzsystem paßt ein Graph $(- \models_{\text{IncidenceSystem}} -)$, wenn er zum enthaltenen Typsystem paßt [p1] und die über *relates* definierten Inzidenzen beachtet werden [p2].

In der Definition von $- \models_{\text{IncidenceSystem}} -$ wird neben dieser Einschränkung auch die *Vererbung von inzidenten Beziehungen* definiert [p2].

$$\begin{array}{l} \hline (- \models_{\text{IncidenceSystem}} -) : \text{TGraph} \leftrightarrow \text{IncidenceSystem} \\ \hline \forall G : \text{TGraph}; \text{IncidenceSystem} \bullet \\ \quad G \models_{\text{IncidenceSystem}} \theta \text{IncidenceSystem} \Leftrightarrow \\ \quad \quad G \models_{\text{TypeSystem}} \theta \text{TypeSystem} \quad \quad \quad \text{[p1]} \\ \quad \forall e : \text{ran } G.E \bullet \quad \quad \quad \text{[p2]} \\ \quad \quad G.\text{type}(\alpha(G, e)) \text{ isA}^* \text{first}(\text{relates}(G.\text{type}(e))) \quad 4 \\ \quad \quad \wedge \quad G.\text{type}(\omega(G, e)) \text{ isA}^* \text{second}(\text{relates}(G.\text{type}(e))) \end{array}$$

Durch obige Definition erhalten Rollenbeziehungen *keine besondere Semantik*. Davon wurde abgesehen, weil eine sinnvolle Semantik für Aggregationen je nach Anwendungskontext recht verschiedenartig sein kann. Bei Bedarf in einer konkreten Anwendung sollte die Definition von $- \models_{\text{IncidenceSystem}} -$ entsprechend ergänzt werden.

4.3 Invariantensystem

Ein Graph paßt zu einem Invariantensystem $(- \models_{\text{InvariantSystem}} -)$ genau dann, wenn er zum enthaltenen Inzidenzsystem paßt [p1] und die Kardinalitäts- und Injektivitätsanforderungen erfüllt werden. Das bedeutet, jeder Knoten darf nicht mehr bzw. weniger ein- bzw. ausgehende Kanten eines bestimmten Typs haben, als in *limits* festgelegt ist [p2], und für injektive Relationstypen dürfen keine Doppelkanten existieren [p3].

In der folgenden axiomatischen Definition wird zum einen festgelegt, daß sich das erste Element eines Tupels aus dem Wertebereich von *limits* immer auf den Start-Entitytyp eines

⁴Die hier verwendeten Funktionen α und ω sind im Anhang B spezifiziert. $\alpha(e)$ berechnet den Startknoten, $\omega(e)$ den Zielknoten der Kante e .

Relationshipstyp und das zweite auf dessen Ziel-Entitytyp bezieht. Zum anderen wird den Kardinalitätsangaben eine Klassensemantik zugeordnet, d.h. sie beziehen sich auf den Relationshipstyp selbst und alle seine Untertypen (wegen $type(e) \text{ isA}^* r$) [p2]. Ebenso wird festgelegt, daß Injektivität eine Eigenschaft einer Relationship-Klasse [p3] und nicht des Typs ist.

$$\begin{array}{l}
\hline
(- \models_{InvariantSystem} -) : TGraph \leftrightarrow InvariantSystem \\
\hline
\forall G : TGraph; InvariantSystem \bullet \\
\quad G \models_{InvariantSystem} \theta EERSchema \Leftrightarrow \\
\quad \quad G \models_{IncidenceSystem} \theta IncidenceSystem \quad [p1] \\
\quad \forall r : relationshipTypes \bullet \quad [p2] \\
\quad \quad \forall v : \text{ran } G.V \mid G.type(v) \text{ isA}^* first(relates(r)) \bullet \\
\quad \quad \quad first(limits(r)).min \\
\quad \quad \quad \leq \#(G.\Lambda(v) \upharpoonright \{e : \text{ran } G.E \mid type(e) \text{ isA}^* r \bullet (e, out)\}) \\
\quad \quad \quad \leq first(limits(r)).max \\
\quad \quad \wedge \\
\quad \quad \forall v : \text{ran } G.V \mid G.type(v) \text{ isA}^* second(relates(r)) \bullet \\
\quad \quad \quad second(limits(r)).min \\
\quad \quad \quad \leq \#(G.\Lambda(v) \upharpoonright \{e : \text{ran } G.E \mid type(e) \text{ isA}^* r \bullet (e, in)\}) \\
\quad \quad \quad \leq second(limits(r)).max \\
\quad \quad \forall r : injective \bullet \\
\quad \quad \quad \forall e_1, e_2 : \text{ran } G.E \mid G.type(e_1) \text{ isA}^* r \wedge G.type(e_2) \text{ isA}^* r \\
\quad \quad \quad \alpha(G, e_1) = \alpha(G, e_2) \wedge \omega(G, e_1) = \omega(G, e_2) \Rightarrow e_1 = e_2
\end{array}$$

4.4 EER-Schema

Ein Graph paßt schließlich zu einem EER-Schema genau dann, wenn er zum enthaltenen Invariantensystem paßt.

$$\begin{array}{l}
\hline
(- \models_{EERSchema} -) : TGraph \leftrightarrow EERSchema \\
\hline
\forall G : TGraph; EERSchema \bullet \\
\quad G \models_{EERSchema} \theta EERSchema \Leftrightarrow \\
\quad \quad G \models_{InvariantSystem} \theta InvariantSystem
\end{array}$$

5 Schlußwort

Das vorliegende Papier stellt die nötige Formalisierung von EER-Schemata, TGraphen und deren Zusammenhang zur Verfügung. Es ist in erster Linie als Referenzpapier zur Absicherung des EER-Anteils des EER/GRAL-Ansatzes gedacht.

Literatur

- [Dahm/Widmann, 1998] P. Dahm, F. Widmann. Das Graphenlabor. Fachberichte Informatik 11–98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.
- [Ebert/Franzke, 1995] J. Ebert, A. Franzke. A Declarative Approach to Graph Based Modeling. in E. Mayr, G. Schmidt, G. Tinhofer, (Hrsg.), Graphtheoretic Concepts in Computer Science, in: Lecture Notes in Computer Science, Band 903, S. 38–50. Springer, Berlin, 1995.
- [Ebert et al., 1996] J. Ebert, A. Winter, P. Dahm, R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. in B. Thalheim, (Hrsg.), 15th International Conference on Conceptual Modeling (ER'96), Proceedings, S. 163–178. Springer, 1996.
- [Franzke, 1997] A. Franzke. GRAL 2.0: A Reference Manual. Research Report 3-97, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1997.
- [Spivey, 1992] J. M. Spivey. The Z Notation: A Reference Manual, in: International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2. Auflage, 1992.

A Werte und Wertebereiche des Graphenlabors

Dieser Anhang beschreibt die Werte (*Value*) und Wertebereiche (*Domain*) des Graphenlabors und den Zusammenhang zwischen ihnen.

Abschnitt A.1 führt zunächst die Struktur von Graphenlaborwerten ein. Abschnitt A.2 beschreibt die Struktur von Wertebereichen und in Abschnitt A.3 schließlich wird die Semantik von Wertebereichen als die ihnen zugeordneten Mengen von Werten definiert.

A.1 Werte

Werte sind entweder *Basiswerte* oder entstehen durch Kombination von Werten (*komplexe Werte*). Als Basiswerte gibt es ganze Zahlen *Int*⁵, Dezimalzahlen *Real*, Zeichenketten *String* und Wahrheitswerte *Bool*. Weitere Basiswerte sind Konstanten aus Aufzählungstypen *EnumConst*. Komplexe Werte entstehen durch Bildung von Listen *ListValue*, Tupeln *TupleValue* und Records *RecordValue* aus anderen Werten. Bei Records werden die enthaltenen Werte an Selektorbezeichner *SelId* gebunden.

```
[Int, Real, String, Bool]
EnumConst == Id
SelId == Id
Value ::=
    IntValue⟨⟨Int⟩⟩
    | RealValue⟨⟨Real⟩⟩
    | StringValue⟨⟨String⟩⟩
    | BoolValue⟨⟨Bool⟩⟩
    | EnumValue⟨⟨EnumConst⟩⟩
    | ListValue⟨⟨seq Value⟩⟩
    | TupleValue⟨⟨seq Value⟩⟩
    | RecordValue⟨⟨SelId ↦ Value⟩⟩
```

Es ist zu beachten, daß nicht alle Werte der Menge *Value* auch Elemente eines der im folgenden Abschnitt definierten Wertebereiche sind. Es ist zum Beispiel möglich, daß die in einer Liste kombinierten Werte nicht den gleichen Typ aufweisen. So ist zum Beispiel

```
ListValue(⟨IntValue(3), StringValue("Hello")⟩)
```

ein denkbarer Wert, der aber in keinem der Wertebereiche enthalten ist. Denn dort werden nur homogene Listen zugelassen.

⁵Im Prinzip könnte hier auch die mathematische Menge der ganzen Zahlen \mathbb{Z} verwendet werden. Die Verwendung von *Int* verweist auf die konkrete Implementierung.

A.2 Wertebereiche

Wertebereiche sind elementare Wertebereiche (*IntDomain*, *RealDomain*, *StringDomain*, *BoolDomain*, *EnumDomain*) oder werden mit Wertebereichskonstruktoren erzeugt. Mit ihnen können Listenwertebereiche (*ListDomain*) über einem Wertebereich, Tupelwertebereiche (*TupleDomain*) über einer Folge von Wertebereichen sowie Recordwertebereiche (*RecordDomain*) über einer Zuordnung von Selektorbezeichnern zu Wertebereichen gebildet werden. Aufzählungstypen (*EnumDomain*) werden über eine endliche Folge von Aufzählungskonstanten definiert. Durch Verwendung der Folge wird implizit eine Ordnung auf den Konstanten bezogen auf den Aufzählungswertebereich definiert.

$$\begin{array}{l} \textit{Domain} ::= \\ \quad \textit{IntDomain} \\ \quad | \quad \textit{RealDomain} \\ \quad | \quad \textit{StringDomain} \\ \quad | \quad \textit{BoolDomain} \\ \quad | \quad \textit{EnumDomain} \langle \langle \textit{seq}_1 \textit{EnumConst} \rangle \rangle \\ \quad | \quad \textit{ListDomain} \langle \langle \textit{Domain} \rangle \rangle \\ \quad | \quad \textit{TupleDomain} \langle \langle \textit{seq Domain} \rangle \rangle \\ \quad | \quad \textit{RecordDomain} \langle \langle \textit{SelId} \mapsto \textit{Domain} \rangle \rangle \end{array}$$

A.3 Werte eines Wertebereichs

Die Funktion *valueSetOf* ordnet jedem Wertebereich eine Menge von Werten zu und definiert so die *Semantik* der Wertebereiche. Die Relation $_ \in_{\textit{Domain}} _$ dient nur der Schreibabkürzung.

$$\left| \begin{array}{l} \textit{valueSetOf} : \textit{Domain} \rightarrow \mathbb{P} \textit{Value} \\ _ \in_{\textit{Domain}} _ : \textit{Value} \leftrightarrow \textit{Domain} \\ \hline \forall v : \textit{Value}, d : \textit{Domain} \bullet \\ \quad v \in_{\textit{Domain}} d \Leftrightarrow v \in \textit{valueSetOf}(d) \end{array} \right.$$

Basiswertebereiche beschreiben die Mengen der elementaren Werte.

$$\left| \begin{array}{l} \textit{valueSetOf}(\textit{IntDomain}) = \textit{ran IntValue} \\ \textit{valueSetOf}(\textit{RealDomain}) = \textit{ran RealValue} \\ \textit{valueSetOf}(\textit{StringDomain}) = \textit{ran StringValue} \\ \textit{valueSetOf}(\textit{BoolDomain}) = \textit{ran BoolValue} \end{array} \right.$$

Ein Aufzählungsdomain beschreibt die Menge aller aufgezählten Konstanten.

$$\left| \begin{array}{l} \forall \textit{enumSeq} : \textit{seq}_1 \textit{EnumConst} \bullet \\ \quad \textit{valueSetOf}(\textit{EnumDomain}(\textit{enumSeq})) \\ \quad = \{ \textit{enumConst} : \textit{ran enumSeq} \bullet \textit{EnumValue}(\textit{enumConst}) \} \end{array} \right.$$

Die Wertemenge eines Listenwertebereichs ist die Menge aller (endlichen) Folgen von Werten aus dem zugrundeliegenden Wertebereich. Listen sind immer homogen.

$$\left| \begin{array}{l} \forall dom : Domain \bullet \\ \quad valueSetOf(ListDomain(dom)) \\ = \{ S : seq(valueSetOf(dom)) \bullet ListValue(S) \} \end{array} \right.$$

Die Wertemenge eines Tupelwertebereichs ist die Menge aller Tupel aus Werten der zugrundeliegenden Wertebereiche. Das Tupel muß die gleiche Stelligkeit haben wie der Tupelwertebereich. Die Elemente des Tupels müssen Werte aus den der Position entsprechenden Wertebereichen sein.

$$\left| \begin{array}{l} \forall dSeq : seq Domain \bullet \\ \quad valueSetOf(TupleDomain(dSeq)) \\ = \{ vSeq : seq Value \\ \quad | \#vSeq = \#dSeq \\ \quad \wedge \forall i : 1..\#vSeq \bullet vSeq(i) \in_{Domain} dSeq(i) \\ \bullet TupleValue(vSeq) \} \end{array} \right.$$

Die Wertemenge eines Recordwertebereichs ist die Menge aller Records, in denen allen Selektoren des Wertebereichs geeignete Werte zugeordnet sind. Es ist nicht zulässig, einem im Wertebereich auftretenden Selektor keinen Wert zuzuordnen.

$$\left| \begin{array}{l} \forall selBind : SelId \mapsto Domain \bullet \\ \quad valueSetOf(RecordDomain(selBind)) \\ = \{ vBind : SelId \mapsto Value \\ \quad | dom vBind = dom selBind \\ \quad \wedge \forall sel : dom vBind \bullet vBind(sel) \in_{Domain} selBind(sel) \\ \bullet RecordValue(vBind) \} \end{array} \right.$$

B Zusätzliche Funktionen

Die folgende Spezifikation ist aus [Franzke, 1997] entnommen. Die Funktionen α und ω werden im Abschnitt 4.2 verwendet.

Gegeben ein Graph und eine Kante des Graphen, berechnet α den Startknoten, ω den Zielknoten der Kante.

$\alpha : TGraph \times EDGE \rightarrow VERTEX$
 $\omega : TGraph \times EDGE \rightarrow VERTEX$

$\alpha =$
 $(\lambda G : TGraph; e : EDGE \mid$
 $e \in \text{ran } G.E \bullet$
 $(\mu v : \text{ran } G.V \mid (e, out) \in \text{ran } G.\Lambda(v)))$

$\omega =$
 $(\lambda G : TGraph; e : EDGE \mid$
 $e \in \text{ran } G.E \bullet$
 $(\mu v : \text{ran } G.V \mid (e, in) \in \text{ran } G.\Lambda(v)))$

Available Research Reports (since 1995):

1998

- 12/98** *Peter Dahm, Jürgen Ebert, Angelika Franzke, Manfred Kamp, Andreas Winter.* TGraphen und EER-Schemata – formale Grundlagen.
- 11/98** *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.
- 10/98** *Jörg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.
- 9/98** *Dieter Zöbel.* Schedulability criteria for age constraint processes in hard real-time systems.
- 8/98** *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.
- 7/98** *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.
- 6/98** *Ulrich Furbach, Michael Kühn, Frieder Stolzenburg.* Model-Guided Proof Debugging.
- 5/98** *Peter Baumgartner, Dorothea Schäfer.* Model Elimination with Simplification and its Application to Software Verification.
- 4/98** *Bernt Kullbach, Andreas Winter, Peter Dahm, Jürgen Ebert.* Program Comprehension in Multi-Language Systems.
- 3/98** *Jürgen Dix, Jorge Lobo.* Logic Programming and Nonmonotonic Reasoning.
- 2/98** *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme.* Zeitstempelnetze in technischen Anwendungen.
- 1/98** *Manfred Kamp.* Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.
- 28/97** *Jürgen Ebert, Bernt Kullbach, Andreas Panse.* The Extract-Transform-Rewrite Cycle - A Step towards MetaCARE.
- 27/97** *Jose Arrazola, Jürgen Dix, Mauricio Osorio.* Confluent Rewriting Systems for Logic Programming Semantics.
- 26/97** *Lutz Priese.* A Note on Nondeterministic Reversible Computations.
- 25/97** *Stephan Philippi.* System modelling using Object-Oriented Pr/T-Nets.
- 24/97** *Lutz Priese, Yurii Rogojine, Maurice Margenstern.* Finite H-Systems with 3 Test Tubes are not Predictable.
- 23/97** *Peter Baumgartner (Hrsg.).* Jahrestreffen der GI-Fachgruppe 1.2.1 'Deduktionssysteme' — Kurzfassungen der Vorträge.
- 22/97** *Jens M. Felderhoff, Thomas Marx.* Erkennung semantischer Integritätsbedingungen in Datenbankanwendungen.
- 21/97** *Angelika Franzke.* Specifying Object Oriented Systems using GDMO, ZEST and SDL'92.
- 20/97** *Angelika Franzke.* Recommendations for an Improvement of GDMO.
- 19/97** *Jürgen Dix, Luís Moniz Pereira, Teodor Przymusiński.* Logic Programming and Knowledge Representation (LPKR '97) (Proceedings of the ILPS '97 Postconference Workshop).
- 18/97** *Lutz Priese, Harro Wimmel.* A Uniform Approach to True-Concurrency and Interleaving Semantics for Petri Nets.
- 17/97** *Ulrich Furbach (Ed.).* IJCAI-97 Workshop on Model Based Automated Reasoning.
- 16/97** *Jürgen Dix, Frieder Stolzenburg.* A Framework to Incorporate Non-Monotonic Reasoning into Constraint Logic Programming.

1997

- 32/97** *Peter Baumgartner.* Hyper Tableaux — The Next Generation.
- 31/97** *Jens Woch.* A component-based and abstractivistic Agent Architecture for the modelling of MAS in the Social Sciences.
- 30/97** *Marcel Bresink.* A Software Test-Bed for Global Illumination Research.
- 29/97** *Marcel Bresink.* Deutschsprachige Terminologie des Radiosity-Verfahrens.
- 15/97** *Carlo Simon, Hanno Ridder, Thomas Marx.* The Petri Net Tools Neptun and Poseidon.
- 14/97** *Juha-Pekka Tolvanen, Andreas Winter (Eds.).* CAISE'97 — 4th Doctoral Consortium on Advanced Information Systems Engineering, Barcelona, June 16-17, 1997, Proceedings.
- 13/97** *Jürgen Ebert, Roger Süttenbach.* An OMT Metamodel.
- 12/97** *Stefan Brass, Jürgen Dix, Teodor Przymusiński.* Super Logic Programs.

- 11/97** *Jürgen Dix, Mauricio Osorio.* Towards Well-Behaved Semantics Suitable for Aggregation.
- 10/97** *Chandrabose Aravindan, Peter Baumgartner.* A Rational and Efficient Algorithm for View Deletion in Databases.
- 9/97** *Wolfgang Albrecht, Dieter Zöbel.* Integrating Fixed Priority and Static Scheduling to Maintain External Consistency.
- 8/97** *Jürgen Ebert, Alexander Fronk.* Operational Semantics of Visual Notations.
- 7/97** *Thomas Marx.* APRIL - Visualisierung der Anforderungen.
- 6/97** *Jürgen Ebert, Manfred Kamp, Andreas Winter.* A Generic System to Support Multi-Level Understanding of Heterogeneous Software.
- 5/97** *Roger Süttenbach, Jürgen Ebert.* A Booch Metamodel.
- 4/97** *Jürgen Dix, Luis Pereira, Teodor Przymusiński.* Prolegomena to Logic Programming for Non-Monotonic Reasoning.
- 3/97** *Angelika Franzke.* GRAL 2.0: A Reference Manual.
- 2/97** *Ulrich Furbach.* A View to Automated Reasoning in Artificial Intelligence.
- 1/97** *Chandrabose Aravindan, Jürgen Dix, Ilkka Niemelä .* DisLoP: A Research Project on Disjunctive Logic Programming.

1996

- 28/96** *Wolfgang Albrecht.* Echtzeitplanung für Alters- oder Reaktionszeitanforderungen.
- 27/96** *Kurt Lautenbach.* Action Logical Correctness Proving.
- 26/96** *Frieder Stolzenburg, Stephan Höhne, Ulrich Koch, Martin Volk.* Constraint Logic Programming for Computational Linguistics.
- 25/96** *Kurt Lautenbach, Hanno Ridder.* Die Lineare Algebra der Verklemmungsvermeidung — Ein Petri-Netz-Ansatz.
- 24/96** *Peter Baumgartner, Ulrich Furbach.* Refinements for Restart Model Elimination.
- 23/96** *Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, Wolfgang Nejdl.* Tableaux for Diagnosis Applications.
- 22/96** *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE in Practice: a Case for KOGGE.
- 21/96** *Harro Wimmel, Lutz Priese.* Algebraic Characterization of Petri Net Pomset Semantics.
- 20/96** *Wenjin Lu.* Minimal Model Generation Based on E-Hyper Tableaux.
- 19/96** *Frieder Stolzenburg.* A Flexible System for Constraint Disjunctive Logic Programming.
- 18/96** *Ilkka Niemelä (Ed.).* Proceedings of the ECAI'96 Workshop on Integrating Nonmonotonicity into Automated Reasoning Systems.
- 17/96** *Jürgen Dix, Luis Moniz Pereira, Teodor Przymusiński.* Non-monotonic Extensions of Logic Programming: Theory, Implementation and Applications (Proceedings of the JICSLP '96 Postconference Workshop W1).
- 16/96** *Chandrabose Aravindan.* DisLoP: A Disjunctive Logic Programming System Based on PROTEIN Theorem Prover.
- 15/96** *Jürgen Dix, Gerhard Brewka.* Knowledge Representation with Logic Programs.
- 14/96** *Harro Wimmel, Lutz Priese.* An Application of Compositional Petri Net Semantics.
- 13/96** *Peter Baumgartner, Ulrich Furbach.* Calculi for Disjunctive Logic Programming.
- 12/96** *Klaus Zitzmann.* Physically Based Volume Rendering of Gaseous Objects.
- 11/96** *J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach.* Graph Based Modeling and Implementation with EER/GRAL.
- 10/96** *Angelika Franzke.* Querying Graph Structures with G²QL.
- 9/96** *Chandrabose Aravindan.* An abductive framework for negation in disjunctive logic programming.
- 8/96** *Peter Baumgartner, Ulrich Furbach, Ilkka Niemelä .* Hyper Tableaux.
- 7/96** *Ilkka Niemelä, Patrik Simons.* Efficient Implementation of the Well-founded and Stable Model Semantics.
- 6/96** *Ilkka Niemelä .* Implementing Circumscription Using a Tableau Method.
- 5/96** *Ilkka Niemelä .* A Tableau Calculus for Minimal Model Reasoning.
- 4/96** *Stefan Brass, Jürgen Dix, Teodor. C. Przymusiński.* Characterizations and Implementation of Static Semantics of Disjunctive Programs.

- 3/96** *Jürgen Ebert, Manfred Kamp, Andreas Winter.* Generic Support for Understanding Heterogeneous Software.
- 2/96** *Stefan Brass, Jürgen Dix, Ilkka Niemelä, Teodor. C. Przymusiński.* A Comparison of STATIC Semantics with D-WFS.
- 1/96** *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner, Bad Honnef 1995.

1995

- 21/95** *J. Dix and U. Furbach.* Logisches Programmieren mit Negation und Disjunktion.
- 20/95** *L. Priese, H. Wimmel.* On Some Compositional Petri Net Semantics.
- 19/95** *J. Ebert, G. Engels.* Specification of Object Life Cycle Definitions.
- 18/95** *J. Dix, D. Gottlob, V. Marek.* Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations.
- 17/95** *P. Baumgartner, J. Dix, U. Furbach, D. Schäfer, F. Stolzenburg.* Deduktion und Logisches Programmieren.
- 16/95** *Doris Nolte, Lutz Priese.* Abstract Fairness and Semantics.
- 15/95** *Volker Rehrmann (Hrsg.).* 1. Workshop Farbbildverarbeitung.
- 14/95** *Frieder Stolzenburg, Bernd Thomas.* Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints.
- 13/95** *Frieder Stolzenburg.* Membership-Constraints and Complexity in Logic Programming with Sets.
- 12/95** *Stefan Brass, Jürgen Dix.* D-WFS: A Confluent Calculus and an Equivalent Characterization..
- 11/95** *Thomas Marx.* NetCASE — A Petri Net based Method for Database Application Design and Generation.
- 10/95** *Kurt Lautenbach, Hanno Ridder.* A Completion of the S-invariance Technique by means of Fixed Point Algorithms.
- 9/95** *Christian Fahrner, Thomas Marx, Stephan Philippi.* Integration of Integrity Constraints into Object-Oriented Database Schema according to ODMG-93.
- 8/95** *Christoph Steigner, Andreas Weihrauch.* Modelling Timeouts in Protocol Design..
- 7/95** *Jürgen Ebert, Gottfried Vossen.* I-Serializability: Generalized Correctness for Transaction-Based Environments.
- 6/95** *P. Baumgartner, S. Brüning.* A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion.
- 5/95** *P. Baumgartner, J. Schumann.* Implementing Restart Model Elimination and Theory Model Elimination on top of SETHEO.
- 4/95** *Lutz Priese, Jens Klieber, Raimund Lakmann, Volker Rehrmann, Rainer Schian.* Echtzeit-Verkehrszeichenerkennung mit dem Color Structure Code — Ein Projektbericht.
- 3/95** *Lutz Priese.* A Class of Fully Abstract Semantics for Petri-Nets.
- 2/95** *P. Baumgartner, R. Hähnle, J. Posegga (Hrsg.).* 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods — Poster Session and Short Papers.
- 1/95** *P. Baumgartner, U. Furbach, F. Stolzenburg.* Model Elimination, Logic Programming and Computing Answers.