Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

A. Siebes, M.L. Kersten

A functional approach to database semantics

Computer Science/Department of Algorithmics & Architecture Report CS-R8804 December

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

A Functional Approach to Database Semantics

Arno Siebes
Martin L. Kersten

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

ABSTRACT

In this paper we describe a functional datamodel with structured domains and show how structured domains can be used to model both incomplete information and complex datatypes. The complex datatypes provide for a unified formal description of the semantics of functional, multivalued, and join dependencies of the relational model, within our functional model. In particular, sets of multivalued dependencies which suffer from the split left-hand-side anomaly and cyclic join dependencies are formally analysed.

CR Categories: H.2.1.
Keywords and Phrases : Functional datamodel, normal forms.

1. Introduction

The deficiency of the relational model as a basis for a semantic description of the Universe-of-Discourse is an important focus of database theory [3, 8, 4, 9, 6]. The deficiency is primarily caused by its weak integrity rules and the unrestricted application of the relational operators to derive new relations. The weak integrity rules can be attacked with an enriched model, such as those based on first-order logic [8]. Concerning the relational operators, more control over the application of operators can be obtained using Abstract Data Types [11] and Object-Oriented Programming Languages [7]. We too focus on the latter deficiency by requiring the database designer to explicate all semantically valid transformations. In [10] we showed that this way the relationship between database intension and extension can be formally described with functions and derivation of information is precisely controlled.

In this paper we extend our datamodel with semantic functions and structured domains. It is shown that these extensions suffice to deal with complex objects and incomplete information in a mathematically precise way. An important effect of our approach to semantic datamodelling is that functional, multivalued, and join dependencies can be seen as structural properties of the Universe of Discourse. This implies that their maintenance is automatically enforced when the data organisation rules are obeyed. The main result is the formal analysis of sets of these dependencies within a single framework. Especially sets of multivalued dependencies which suffer from the split left-hand-side anomaly and cyclic join dependencies are considered.

The paper is organised as follows. Section 2 describes the paradigm we adhere to for modelling database semantics. It concludes with a description of the consistency relation between database intension and extension. Next, in section 3, we address incomplete information as a domain structuring problem. Section 4 is an interlude on functional

Report CS-R8804
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

dependencies, which are used in section 5 to model complex objects. In turn, complex objects are used to study multivalued dependencies in section 6. An indication of our results on join dependency theory is described in section 7. The paper concludes with a summary and directions for future research.

The running example of this paper is taken from [2]. In relational terms, our Universe of Discourse (UoD) consists of four attributes: $B(uyer), V(endor), P(roduct), C(urrency)$, and two multivalued dependencies: $BV \rightarrow\rightarrow P \mid C$ and $PC \rightarrow\rightarrow B \mid V$. In the course of the paper we will show how this UoD is modelled in our functional datamodel.

2. An approach to modelling database semantics

In database design it is customary to distinguish two levels of abstraction: the *type level* and the *extension level*. They model the structure, described using the properties, and elements of the Universe-of-Discourse, described with facts, respectively. A property can be considered to be a predicate which might or might not be satisfied by a real world object. Such a predicate might be a static property , e.g. "x has a colour", but it might also be a dynamic property, e.g. "x can walk". On the other hand, a property can be considered to be an attribute in the relational model. The nature of properties is of no interest to us in this paper nor how they are represented, but the reader who reads them as attributes will not be contradicted. Some property sets are meaningful in the sense that the objects with these properties form a natural class within the UoD:

DEFINITION

a) An entity type is a set of properties. The set of all entity types is denoted by *ETS*. Moreover, there exists a function $domain: ETS \rightarrow Domains$, which assigns a domain to each entity type. Instead of $domain(e)$, we will often write D_e.

b) An entity t of type e is a pair <entity-type, value>, such that for an entity $t = <e,v>$, $v \in domain(e)$; t is also called an instance of e.

In reality, entities participate in complex relationships, which can be described in a database design as predicates over the participating entity types. Moreover, relationships have their own domains. Hence, relationships are represented in our model as entity types as well. For example, the property set of the relation may be the union of the property sets of the participating entity types. And the domain of this relation might be the cartesian product of the domains of the participating entity types.

In our example UoD there is a relationship between a *Vendor* and the *Product* she sells which can be modelled by an *On_sale* entity type; informally, $On\_sale = r(Vendor, Product)$. Moreover, $D_{On\_sale} = D_{Vendor} \times D_{Product}$

Unlike the relational model we do not permit unrestricted use of operators, because this leads to its major semantical deficiencies. In particular, the join and projection operators allow for the construction of meaningless relations. By analogy, a natural language is not the result of an alphabet and a concatenation operator, nor can one take arbitrary portions from words without loosing semantics. Contrary, we assume that all relevant entity types are given in the database schema. Hence a join operator is not needed and a projection is only allowed if the result entity type is defined.

More general, the database designer may define arbitrary functions between entity types as opposed to only projections. These functions describe transformations of entities of one type into entities of another type; e.g. an object that is described by its mass and acceleration can be transformed to an object that is subject to a certain force. Or, more in the line of traditional database applications, given a box with a dozen items and a price per item, the price per box can be calculated. As can be seen from this examples, functions transform entities from one type to entities of another type, i.e. functions 'live' on the extension level. But of course, it should be made visible at the type level that such transformations are

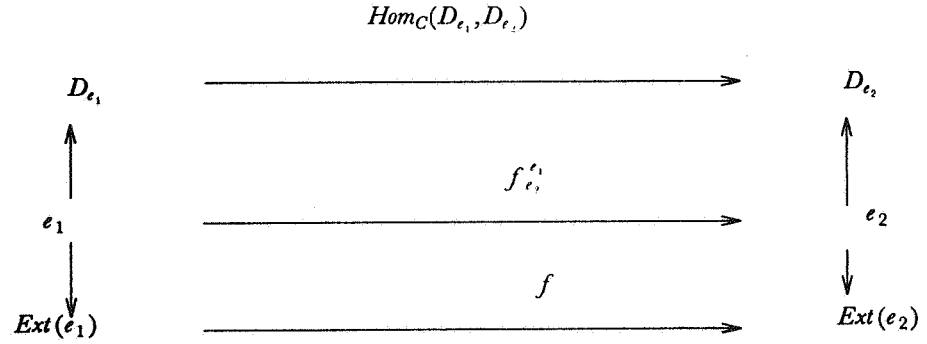
possible. To this end we will use function types. Apart from the indication that a function exists, we will also assign a domain to the function type. The domain of a function type shows what kind of functions transform the entities: e.g. if both domains of the entity types involved are Boolean algebras then the function might be a Boolean algebra homomorphism respecting this structure or simply a function disregarding this structure. Therefore, the domain will be denoted by $Hom_C(D_{e_1},D_{e_2})$, where C denotes the category ('structure') where the function should be chosen from.

DEFINITION

a) The function type $f^{e_1}_{e_2}$ between e_1 and e_2 has domain $Hom_C(D_{e_1},D_{e_2})$.

b) A semantic transformation f is a function $\in Hom_C(D_{e_1},D_{e_2})$.

The constellation of entity types and function types can be pictured as follows:



In the database schema of our example UoD, there should be an explicitly defined function type from *On_sale* to *Vendor*. Moreover, as we will indicate shortly, this function type denotes exactly one semantic transformation.

A conceptual model of a database is then a type level description of the Universe-of-Discourse in terms of entity types, to represent facts, and function types, to represent semantic transformations. The type level is then a directed graph, called the type graph, which forms the description of the database intension. In a nutshell, the task of a database system is to keep track of the relevant entities while obeying the semantic transformations during manipulation. The instances can be derived from their entity type using an extension mapping:

DEFINITION

Let e be an entity type. Then $Ext(e)$, the extension of e, is a subset of the domain of e, i.e. $Ext(e) \in P(D_e)$.

2.1. Database design philosophy

As mentioned before, all relevant details of the UoD should be given explicitly at design time. As we are concerned with a static model of the UoD in this paper, the relevant details of the static structure should be supplied only. Using the terminology of the previous section we can formulate the design paradigm as follows:

DESIGN RULE 1

ALL RELEVANT SEMANTIC INFORMATION MUST BE MADE EXPLICIT, IN AN UNAMBIGUOUS WAY BY MEANS OF ENTITY TYPES, DOMAINS, AND FUNCTION SPECIFICATIONS.

Moreover, the distinction between the various levels in the model should be used to solve modelling problems on the right level of abstraction, e.g. in the following section we will show that incomplete information is a statement about the domain of an entity type, while multi-valued dependencies can be seen as a statement about the type graph representing

the UoD. Hence the following rule:

DESIGN RULE 2

SEMANTIC PROBLEMS MUST BE HANDLED AT THE PROPER LEVEL OF ABSTRACTION: DOMAIN LEVEL, ENTITY TYPE LEVEL, AND EXTENSION LEVEL.

According to our design rules, function types must be given explicitly for all relevant semantic transformations. In particular, with every entity type e the identity mapping f_e^e should be defined. Moreover, we have seen that a relationship is represented as an entity type. Thus, the information embodied in a relationship should be complemented with functions to denote their role, thereby describing the allowable projections. Hence:

PROPOSITION

If an entity type e denotes a relationship between entities types $e_1, \ldots, e_n$, then there are function types $f: e \rightarrow e_i$, for all $i \in \{1, \ldots, n\}$.□ †

In the example UoD there would be semantic transformations from *On_sale* to both *Vendor* and *Product* which model their roles within the relationship.

Furthermore, semantic transformations can be juxtaposed to describe a composed transformation, this is equivalent to using function composition. Hence:

1) When $f_{e_2}^{e_1}$ and $f_{e_3}^{e_2}$ exists then $f_{e_3}^{e_1}$ exists by definition.

And as usual (and it avoids semantic misinterpretation) we have:

2) Let f,g and h be functions that can be juxtaposed, then

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Finally, suppose that a function type $f_{e_2}^{e_1}$ has two semantic transformations, say f and g, which differ on $Ext(e_1)$; that is, there is a $v \in Ext(e_1)$ such that $f(v) \neq g(v)$. This implies that f and g give different semantics to the entity type e_2. This means that the possible interpretation of e_1 instances depends on the semantic transformations. In this sense, f and g are ambiguous transformations which should be avoided in the database design. The consequence for our model is that:

PROPOSITION

A function type has exactly one semantic transformation associated with it. □

This proposition allows us to use the same name for the function type and the semantic transformation it denotes. Moreover, it implies that if there are two paths in the type graph between two entity types, the composed functions specified by these two paths should be the same; the paths should commute. Alternatively, the information derived does not depend on the evaluation path chosen.

Given two entity types e_1 and e_2, and the function type $f_{e_2}^{e_1}$, every instance of e_1 can be transformed to an instance of e_2 as for every $t_1 \in D_{e_1}$, we have that applying the function $f_{e_2}^{e_1}$ to t_1 yields $f_{e_2}^{e_1}(t_1) \in D_{e_2}$. To obtain a well-defined function on the extension level, every $t_1 \in Ext(e_1)$ should be mapped to a member of $Ext(e_2)$. Hence we have the following containment condition:

CONTAINMENT CONDITION

Let e_1,e_2 and $f_{e_2}^{e_1}$ be given, then $\forall v \in Ext(e_1)$: $f_{e_2}^{e_1}(v) \in Ext(e_2)$.

---

† Due to space limitations proofs are omitted. However, if necessary the relevant details of a proof are discussed.

2.2. Functional enhancement of specialisation/generalisation

In our previous paper [10], we defined an entity type e_1 to be a specialisation of entity type e_2 iff the property set of e_2 is a subset of the property set of e_1. In that situation the map $f_{e_2}^{e_1}$ exists by definition and is called an ISA-relation. This observation suggests the following more general definition for specialisation:

DEFINITION

An entity type e_1 is a specialisation of entity type e_2 iff the function type $f_{e_2}^{e_1}$ is defined.

Thus, a semantic transformation defines one entity type (its domain) to be a specialisation of another entity type (its codomain). Therefore, if the database designer wishes an entity type e_1 to be a specialisation of entity type e_2, he should specify the function $f_{e_2}^{e_1}$. In the running example, *On_sale* is a specialisation of *Vendor*. A more general example, if the defining property set of e_2 is a subset of the defining property set of e_1, the database designer should provide for $f_{e_2}^{e_1}$, because e_1 is then a natural specialisation of e_2.
The set of all specialisations of an entity type e is given by: $S_e = \{e' \mid f_e^{e'} \textit{ is defined}\}$. A consequence of the Containment Condition is that each instance t' of such a specialisation e' of e also defines an instance t of e.

Generalisation is the dual concept of specialisation:

DEFINITION

An entity type e_1 is a generalisation of entity type e_2 iff the function type $f_{e_1}^{e_2}$ is defined.
The set of all generalisations of an entity type e can be given as: $G_e = \{e' \mid f_{e'}^{e} \textit{ is defined}\}$.

Note that, although generalisation and specialisation are dual concepts, G_e and S_e are not each others complements, as their intersection is non-empty and their union is not necessarily the whole entity type set.

3. Incomplete information

Let the domain of an entity type e be the finite set $D_e = \{v_i \mid i \in 1...n\}$. Incomplete information can be seen as a propositional logic formula over the members of D_e using the connectors $\wedge$, $\vee$ and $\neg$ [9]. Another way to model this is through a Boolean algebra with the following association of operators:

1) the greatest lower bound $\cap$ corresponds with $\wedge$,

2) the least upper bound $\cup$ corresponds with $\vee$,

3) the complement - corresponds with $\neg$.

The easiest Boolean algebra derived from a domain is its powerset. Taking the set $B_e = P(D_e)$, the elements of D_e correspond to the singletons of B_e, $v_j \vee v_k$ corresponds to $\{v_j, v_k\}$, $\neg v_1$ corresponds to $\{v_i \mid i \in 2...n\}$ and so forth. The empty set denotes that the information on the entity is inconsistent. This leads to the following proposition:

PROPOSITION

Incomplete information can be modelled in a conceptual schema by replacing each domain D_e by the enhanced domain $B_e = P(D_e)$.□

Note that we only support Null-values in the sense of 'value unknown', other kinds of Null-values such as 'no value allowed' are not considered. Suppose that the vendors in the example UoD are {*Jones,Smith,Brown*}. Then {*Jones,Smith*} indicates that we know that the vendor is either *Jones* or *Smith*, but not *Brown*.

Since we use functions over domains to model semantic meaningful transformations, we should discuss the interaction between the extended domains and these functions. For example, Boolean-algebra morphisms respect the structure of the domain, while partially ordered set morphisms only ensure a monotonicity condition. This means that

incompleteness is preserved by the functions. Unfortunately, finite set morphisms 'forget' the Boolean algebra structure of their domain. For example, if we introduce a semantic transformation from *Vendor* to *Location* then a finite set morphism would allow for mapping {*Jones,Brown*} to the location {*Amsterdam*} and at the same time allow for the mapping {*Jones*} to the location {*London*}.

Hence, the database designer should carefully select the right category for the functions and, of course, both the domain and the codomain of this function should be objects within this category. For example, if both the domain and the co-domain are Boolean algebras, the function could be any of the following morphisms: a Boolean algebra morphism, a partially ordered set morphism, or a finite set morphism.

Unfortunately, incompleteness as defined above only partially provides for the machinery needed to model an UoD. For a compound entity type, e that denotes a relation between the entity types $e_1, \ldots, e_n$, the database designer might find incompleteness in some components tolerable while it is intolerable in others . In such a case, some of the functions with e as their domain will be Boolean algebra morphisms, while others will be finite set morphisms; this we call partial incompleteness:

DEFINITION

Let e be an entity type, e supports partial incompleteness iff there is an $e' \in G_e$, such that $f^e_{e'}$ is a Boolean algebra morphism. Moreover, e supports (total) incompleteness if $\forall\, e' \in G_e$: $f^e_{e'}$ is a Boolean algebra morphism.

As an example, let the domain of e be a boolean algebra, say $B_e = P(D)$, such that we can represent incomplete information. In particular, the Null-value for e is represented by B_e. Often, one Null-value to represent a single partially known entity is not enough. For example, we may encounter two entities in the UoD for which we only know that they are characterised by {*Jones,Smith*}. Using a single representation would make them indistinguishable. Using partial incompleteness, however, we can extend the domain with an arbitrary large (finite) set of Null-values.
Therefore, define the new domain of e to be:

$$B_e' = N_n \times B_e,$$

where N_n denotes the natural numbers up to n. This yields the Null-values (i,D) for $i \in \{0, \ldots, n\}$. For example, the two partially known entities can now be represented as (13,{*Jones,Smith*}) and (14,{*Jones,Smith*}). Both denote an entity which may be either *Jones* or *Smith*, but they might or might not be equal. Moreover, all occurrences of (13,{*Jones,Smith*}) represent the same unknown entity.

Note that although we have an arbitrary large number of Null-values, this still might not be adequate to represent all the information about the UoD we have. As an example, we might have a 'third' partially known person about whom we know that he is either the same as (13,{*Jones,Smith*}) or the same as (14,{*Jones,Smith*}), but we do not know which of the two he is. Of course, this can be represented if we iterate the 'trick' used above; but then we can again pose a piece of non-representable information as above. A further discussion of this problem is however outside the scope of this paper. In the rest of this paper we use the extended domain interpretation. However, we use the D_e notation instead.

4. Functional Dependencies

Extending the domains to cope with incomplete information affects the way functional dependencies (fd's) are being dealt with. The relational definition of fd's can be re-phrased as follows (where the context gives the relation in which the dependency holds):

DEFINITION

Let $e_1, e_2, e_3 \in ETS$, such that $e_2, e_3 \in G_{e_1}$, then e_2 functionally determines e_3 in

the context of Ext(e_1), denoted $fd(e_2, e_3; e_1)$, iff for any two entities

$$t_1, t_2 \in Ext(e_1): f^{e_1}_{e_2}(t_1) = f^{e_1}_{e_2}(t_2) \rightarrow f^{e_1}_{e_3}(t_1) = f^{e_1}_{e_3}(t_2).$$

From the definition it is obvious that a functional dependency defines a partial function, because it is only defined on $f^{e_1}_{e_2}(Ext(e_1))$ and not on $Ext(e_2)$. To make the analysis of the interaction between extended domains and functional dependencies easier, it would be helpful if the partial function would be a complete function.

To achieve this, note that there are two kinds of e_2 entities, those that are 'generated' from e_1 entities and those that are not. Following our design rules, one would be tempted to 'split' e_2 into two entity types:$e_{2,0}$ and $e_{2,1}$, where $e_{2,0}$ denotes the e_2 entities that where generated by e_1 entities and where $e_{2,1}$ denotes the other e_2 entities. However, then we would still have the old e_2 entity type as the unifying entity type of $e_{2,0}$ and $e_{2,1}$ within our database schema. The functional dependency still holds on e_2; we haven't solved our problem!

To solve it, we introduce the notion of a subtype. A detailed discussion of subtypes is outside the scope of this paper. However, informally, subtypes can be characterised as follows:

> Let e_1 be an entity type, every function $f^{e_2}_{e_1}$ for some $e_2 \in S_{e_1}$ partitions an extension of e_1 in two parts, viz. those entities that are derived from e_2 entities and those that are not. The set of e_1 entities that are derived from e_2 entities is denoted by $e^{\,2}_{1}$, $e^{\,2}_{1}$ is a subtype of e_1.

Just as function types can be defined between entity types, they can also be defined between entity subtypes, and thus a functional dependency can be seen as a function between two entity subtypes. However, the use of extended domains, implies that we have obtained a generalisation of the 'standard' functional dependencies. To make this generalisation canonical, we have to require that this function is a Boolean algebra homomorphism; i.e.:

1) $f(A \cup B) = f(A) \cup f(B)$

2) $f(A \cap B) = f(A) \cap f(B)$

This discussion leads to the following proposition:

PROPOSITION

> Let $e_1 \in E$, and let $e_2, e_3 \in G_{e_1}$. Then the functional dependency $fd(e_2, e_3; e_1)$ holds iff the function type $f: e^{\,1}_{2} \rightarrow e^{\,1}_{3}$ exists, and denotes a Boolean algebra homomorphism. □

Therefore, there is no difference between an explicitly given function between subtypes and a functional dependency; as a result, we can use $fd(e_2, e_3; e_1)$ and $f: e^{\,1}_{2} \rightarrow e^{\,1}_{3}$ interchangeably. Moreover, note that this proposition enables us to see a functional dependency as a structural property of the UoD, as we had already shown in [10].

In our example UoD we might introduce the (artificial) constraint that a vendor may only sell one product, this would be modelled as $fd(Vendor, Product; On\_sale)$.

5. Causal Set Relationship Entity Types

The simpliest form of complex objects are sets of entities. This grouping of entities into sets naturally occurs during database design. For example, vendors sell a group of products and buyers have a stack of currencies. According to our design paradigm this grouping should be modelled explicitly. Moreover, associated with each group there is an entity which explicates the semantic reason for this grouping. For example, a vendor is an entity responsible for a group of products. And for each vendor there exists only one group of products.

At the type level this means that there is an entity type that models the possible entity sets and an associated grouping entity type. Moreover, each instance of this grouping entity type can be responsible for at most one set of entities. This leads to the following definition:

DEFINITION

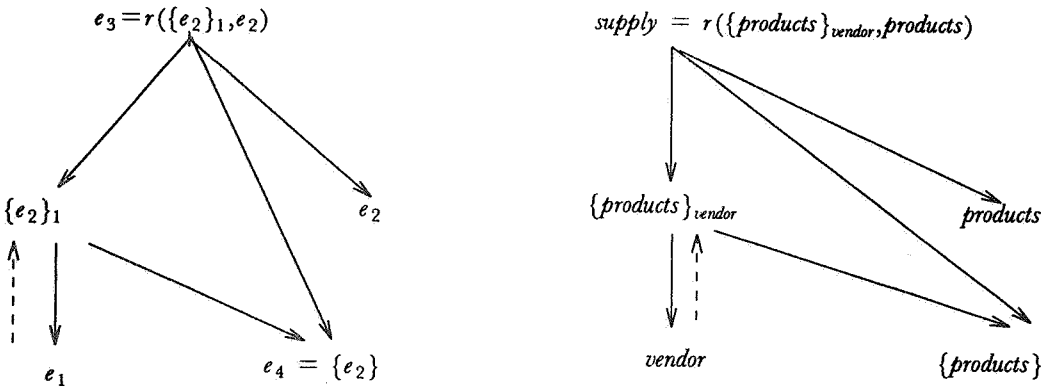
Let e_1 and e_2 be two entity types, the Causal Set Relationship of e_2 by e_1, denoted by $\{e_2\}_{e_1}$,† has as defining property: 'The e_1 entity causes the grouping of a set of e_2 entities' and has as domain: $D_{\{e_2\}_{e_1}} = D_{e_1} \times P(D_{e_2})$.

Moreover, $e_1 \in S_{\{e_2\}_{e_1}}$, and $fd(e_1, \{e_2\}_{e_1}; \{e_2\}_{e_1})$

Furthermore, there is a need for a 'member-of' relation between the flat entity type e_2 and the Causal-Set-Relationship (CSR) entity type $\{e_2\}_{e_1}$, denoting which instances belong to the set. Due to domain incompatibility, we can not define a function relating the CSR $\{e_2\}_{e_1}$ and flat entity type e_2. However, the membership relation can be modelled by an entity type e_3, with the defining property: 'entities of type e_2 which are member of a set of e_2 instances caused by an e_1 entity', and as domain $D_{e_3} = D_{e_2} \times D_{\{e_2\}_{e_1}}$.

Now we should ensure that every 'flat' entity is indeed a member of the group as indicated by this membership relation; 'member-of(a,A)' should imply that $a \in A$. To start with, we can define the following semantic transformation from the membership relation to the CSR: $f : e_3 \rightarrow \{e_2\}_1$ that maps $(a,(x,A))$ to $(x, \{a\} \cup A)$. Following, we have to check that this semantic transformation is equivalent to a simple projection. For this purpose, we use a fourth entity type e_4 that simply denotes sets of e_2 entities; i.e. there is no grouping entity type; it is informally denoted by $\{e_2\}$. Now, define semantic transformations from the membership relation e_3 and from the CSR $\{e_2\}_{e_1}$ to e_4 as simple projections. The commutativity property of paths in the type graph gives that 'member-of(a,A)' implies that $a \in A$, as required above.

The construction made above, can be pictured as follows.



In our example UoD, the prototypical example of CSR is formed by the relation between a vendor and her products. This gives the CSR:

$$\{products\}_{vendor}$$

which groups products by the vendors supplying them. Moreover:

$$D_{\{products\}_{vendor}} = D_{vendor} \times P(D_{products})$$

and each vendor is associated with one set of products:

$$fd(vendor, \{products\}_{vendor}; \{products\}_{vendor})$$

This situation is pictured above, where *supply* describes the membership relation between *products* and $\{products\}_{vendor}$ and $\{products\}$ ensures the validity of this membership relation as

† e_1 is the grouping entity type

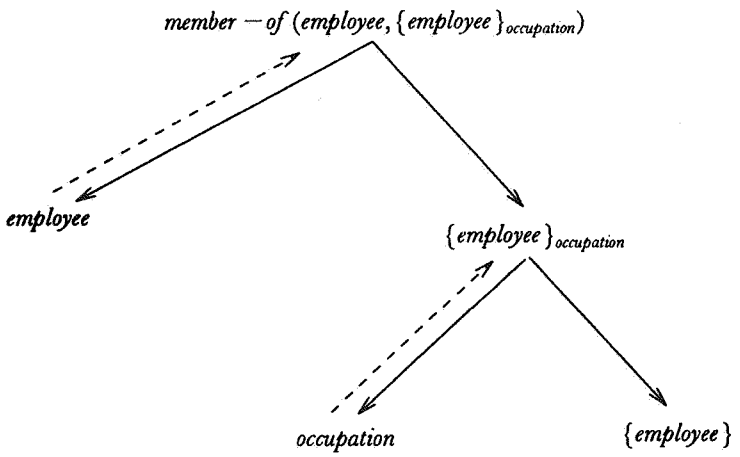
described above.

Although we have made sure that 'member_of(a,A)' implies $a \in A$, we cannot guarantee that $a \in A$ implies 'member_of(a,A). For example it could happen that $\{car,bike\}_{Jones}$ is an instance of $\{products\}_{vendor}$, while we only find $(car, \{car,bike\}_{Jones})$ in the associated membership relation; i.e. although *bike* is clearly a member of $\{car,bike\}$ it is not given as such in the membership relation!

This problem would be solved, if we did not give a CSR $\{e_2\}_{e_1}$ the explicit domain $D_{e_1} \times P(D_{e_2})$, but instead a domain of symbolic names. Inspecting the membership relation for such a symbolic name would then reveal which set of flat entities is represented by that symbolic name. However, using all the elements of a set, e.g. for aggregates, is then much harder to represent. The more general model, in which this poses no problem is outside the scope of this paper. A more pragmatical solution to the problem posed above is to view it as a problem in the dynamic specification of the UoD; if the behaviour of the UoD is adequately modelled, problems as the one above cannot occur. This is the point of view, we will adopt in the rest of this paper.

The CSR concept can be used to model concepts popular in different database design methods. We will give two examples:

In the Entity-Relationship model and its derivatives, there are various ways to denote that an entity type is the union of disjoint subtypes. For example, *employees* are either *engineer* or *manager*. Let *occupation* denote a generalisation of *employee* with domain $\{engineer,manager\}$, then we can construct the CSR $\{employee\}_{occupation}$, with the associated membership relation $member-of(employee, \{employee\}_{occupation})$. The fact that an *employee* is either an *engineer* or a *manager* simply translates to the observation that *employee* functionally determines $\{employee\}_{occupation}$ in the membership relation. In a picture:



The second example is chosen from the theory of relational databases, where operations such as average, sum or min deliver a value computed over a set of tuples. Such operations are called aggregate functions. In other words, an aggregate function is a semantic transformation from a set to a single value. Using CSR's, we can define aggregate functions as semantic transformations from a CSR-type to a flat entity type. However, the projection from a CSR to the grouping entity type should not be considered as an aggregate function:

DEFINITION

An aggregate function is a semantic transformation from a CSR-entity type to a flat entity type.

Thus we can model functions between two flat entity types, between two CSR's, from a flat entity type to a CSR (e.g. the fd from the grouping entity type) and from a CSR to a flat entity type. Thus the database designer should be careful defining transformations. Moreover, defining CSR entity types, we paid no attention to the Boolean algebra nature

of domains; this poses no real problem, as long as the database designer makes sure that his functions are Boolean algebra morphisms (when appropriate).

In practice, it can happen that there is more than one potential grouping entity type for one CSR; say we have the grouping entity types e_2 and e_3 for e_1. Formally, we have to construct the entity type e_4, where $e_4 = r(e_2,e_3)$, such that $fd(e_2,e_4;e_4)$ and $fd(e_3,e_3;e_4)$. And then we should construct this CSR using e_4 as the grouping entity type. In the rest of this paper, we will simply use $\{e_1\}_{e_2,e_3}$ to denote such a situation, in order not to complicate the entity type graph. In a picture:

6. Multivalued dependencies

In this section we show that causal set relationships can be used to analyse the semantics of multivalued dependencies. The relational definition of multivalued dependencies (mvd's) can be translated to our concepts as follows:

DEFINITION

Let e_1,e_2,e_3,e_4 be entity types such that e_1 denotes $r(e_2,e_3,e_4)$ and $G_{e_4} \subseteq G_{e_1} - G_{e_2} - G_{e_3}$. Then an extension $\mathrm{Ext}(e_1)$ satisfies the mvd $e_2 \twoheadrightarrow e_3$, denoted by $mvd(e_2,e_3;e_1)$, if for any two entities $t_1,t_2 \in \mathrm{Ext}(e_1)$ such that $f^{e_1}_{e_2}(t_1) = f^{e_1}_{e_2}(t_2)$, there exists an entity $t_3 \in \mathrm{Ext}(e_1)$, such that: $f^{e_1}_{e_2}(t_3) = f^{e_1}_{e_2}(t_1)$, $f^{e_1}_{e_3}(t_3) = f^{e_1}_{e_3}(t_2)$ and $f^{e_1}_{e_4}(t_3) = f^{e_1}_{e_4}(t_1)$.

This definition requires that after each update the multivalued dependency is checked. However, a 'structural' definition, i.e. a restriction on the structure of the entity type level, would automatically enforce the constraint. Thus we have to determine what constitutes a 'good' structure [5].

The notion of a good structure is not new in database theory. The prime example is the normalisation of relational databases. The multivalued dependency $A \twoheadrightarrow B \mid C$ on $R = \{ABC\}$ is known to be equivalent to $R = \pi_{AB}(R) * \pi_{AC}(R)$ and not R, but $\pi_{AB}(R)$ and $\pi_{AC}(R)$ can be stored and then R is derived when necessary. Then the insertion of $t = (a,b,c)$, triggers the generation of the tuples necessary to satisfy the multivalued dependency automatically.

Of course, a solution in which the extra tuples are automatically generated is readily available in our model: let e_1 and e_2 be two entity types both denoting a relation $r(A,B,C)$, and both with $P(D_A \times D_B \times D_C)$ as their domain, such that e_1 represents the 'inserted' tuples and e_2 represents the complete set of tuples; and thus the semantic transformation $f^{e_1}_{e_2}$ maps a set of tuples to its closure under the multivalued dependency. However, we feel that this is a non-solution, as it completely disregards the structure of the UoD implied by the dependency. In fact, every dependency can be modelled using two entity types instead of one and a generating function from one to the other; this shows both its syntactic power and its semantic weakness.

Ideally, larger extensions are built from the smaller ones by adding pieces of certain fixed kinds; then extensions are determined up to isomorphism by how many pieces of each kind were added. In a relation without dependencies, there is one kind of pieces, viz. tuples. If the relation includes, however, a multivalued dependency, then the insertion of one tuple potentially generates a set of tuples that should also be included to satisfy the dependency.

Thus, we would like to decompose the extension in (maximal) independent sets, under a suitable notion of independence, such that insertion leads to either:

a) adding elements to an independent set;

b) adding a new independent set;

c) or both.

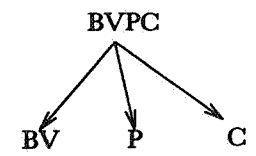
Thus, 'independence' should imply that if two sets X_1 and X_2 are independent, an insertion in X_1 neither relies on X_2 nor does it affect X_2; i.e. X_1, X_2 and $X_1 \cup X_2$ are valid extensions and when an update in the X_1 part of $X_1 \cup X_2$ is performed, the validness of the new extension $X_1 \cup X_2$ is independent of it's X_2 part. Moreover, let $Ext_1(e)$ and $Ext_2(e)$ be two extensions of e such that $Ext_1(e) \subseteq Ext_2(e)$ and X_1 and X_2 are two independent components in E_1. Then there are also two independent components Y_1 and Y_2 in $Ext_2(e)$ such that $X_1 \subseteq Y_1$ and $X_2 \subseteq Y_2$.

Let $R = \{ABC\}$ with the multivalued dependency $A \rightarrow\rightarrow B$ and two valid extensions of R called r_1 and r_2. Then $r_1 \cup r_2$ is a valid extension of R iff $\pi_A(r_1) \cap \pi_A(r_2) = \emptyset$. So an independent component of an extension of R has exactly one A value. Moreover, such an A value is uniquely associated with a set of B values that does not depend on the C values. †

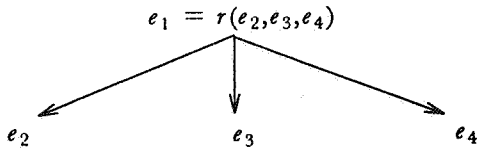
The discussion above suggests that instead of using a relation with a multivalued dependency, we might as well model a relation using a CSR:

> Let e_1, e_2, e_3, e_4 be entity types such that $e_1 = r(e_2, e_3, e_4)$ and the $mvd(e_2, e_3; e_1)$ holds. Then the UoD might as well be modelled using the entity types e_1 and $\{e_3\}_{e_2}$, where $e_1 = r(\{e_3\}_{e_2}, e_4)$.

Now we have a definition of a multivalued dependency and an alternative technique to model relations with a multivalued dependency. In our example we have the mvd $BV \rightarrow\rightarrow P$, using the definition and the alternative modelling technique, we get, informally, the following two pictures:

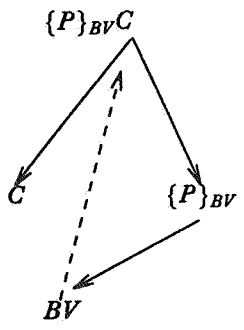


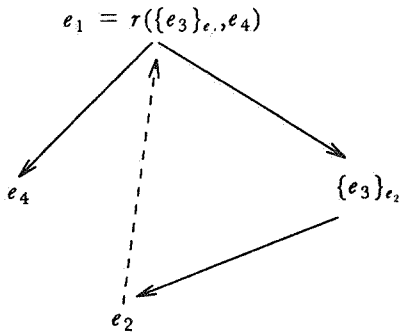
with $mvd(BV,P;BVPC)$



with $mvd(e_2, e_3; e_1)$

using the definition of multivalued dependencies, and





† This result is already well-known in the literature of relational database theory. However, independent components are used when we discuss sets of multivalued dependencies. To avoid an overloaded discussion we introduce them here already.

modelled alternatively (we only pictured the essential entity types and arrows, to keep the picture simple).

Thus, the equivalence of these two modelling techniques should be proven. This is done by proving that the models of the UoD generated by the two techniques are equivalent. A detailed study of equivalent datamodels is beyond the scope of this paper, however the equivalence of the two definitions will be proved using the following definitions:

DEFINITION
The extension of the type level, denoted by $Ext(ETS)$, is defined as the direct sum of the extensions of the entity types at the type level:

$$Ext(ETS) = \oplus_{e \in ETS} Ext(e)$$

DEFINITION
Two models of the same UoD, ETS_1 and ETS_2 are called equivalent iff there is a bijection between $Ext(ETS_1)$ and $Ext(ETS_2)$ that maps, in either direction, valid extensions into valid extensions.

The proof of the equivalence of the two techniques to model a UoD with a multivalued dependency can now be constructed informally as follows:
Let ETS_1, be the B,V,P,C world with the mvd $BV \rightarrow\rightarrow P$ as in the first picture and let ETS_2 be the B,V,P,C world with the mvd $BV \rightarrow\rightarrow P$ as in the second picture above. We only have to define mappings f on $BVPC$ and g on $\{P\}_{BV}C$, as the extensions of their generalisations follow immediately:

$$f(X) = \bigcup_{bv \in \pi_{BV}(X)} \bigcup_{c \in \pi_C(\sigma_{bv}(X))} \{(bv, \pi_P(\sigma_{bv}(X)), c)\}$$

$$g(Y) = \bigcup_{bv \in f_{BV}^{\{P\}BVC}(Y)} \bigcup_{c \in f_C^{\{P\}BVC}(\sigma_{bv}(Y))} \{(bvpc) \mid p \in f_{\{P\}}^{\{P\}_{BV}C}(\sigma_{bv}(Y))\}.$$

LEMMA
The above defined mappings f and g are each other inverses. □

THEOREM
The two techniques to model a UoD with a multivalued dependency are equivalent. □

Consequently, every multivalued dependency can be modelled structurally, i.e. at the type level. In accordance with our design paradigm this implies that multivalued dependencies should be modelled at the type level, because they describe essential semantic information about the UoD.

It is well-known that if in a relation $R = \{ABC\}$ the multivalued dependency $A \rightarrow\rightarrow B$ holds, the multivalued dependency $A \rightarrow\rightarrow C$ also holds. This inspires yet another technique:

> Let e_1, e_2, e_3, e_4 be entity types such that $e_1 = r(e_2, e_3, e_4)$ and the $mvd(e_2, e_3; e_1)$ holds. Then the UoD might also be modelled using the entity types e_1, the CSR $\{e_3\}_{e_2}$ and the CSR $\{e_4\}_{e_2}$, where $e_1 = r(\{e_3\}_{e_2}, \{e_4\}_{e_2})$.

And with a proof similar to the one sketched above, we have:

THEOREM
The three techniques to model a UoD with a multivalued dependency are equivalent. □

The choice between the two structural solutions for multivalued dependencies is only a matter of taste.

6.1. Sets of multivalued dependencies

In the previous subsection, we have seen that every multivalued dependency can be modelled structurally. However, if more than one multivalued dependency is required to hold within an entity type, the structural modelling of the two mvd's at the same time might be impossible, as the two structures 'interfere'. This interference of multivalued dependencies is known in normalisation theory as conflicting sets of multivalued dependencies. A conflicting set of multivalued dependencies is a set of multivalued dependencies that suffers either from the intersection anomaly or from the split left-hand-side (lhs) anomaly or from both.

A full set of multivalued dependencies M is said to suffer from the intersection anomaly iff there are left-hand-sides X and Y in M such that $(X \cap Y) \rightarrow\rightarrow (Dep(X) \cap Dep(Y))$ † is not derivable from M.

A full set of multivalued dependencies M is said to suffer from the split left-hand-side anomaly iff there are two left hand sides X and Y in M, such that for distinct V and W in $Dep(X)$, both $V \cap Y \neq \emptyset$ and $W \cap Y \neq \emptyset$. During normalisation, the dependency having X as its lhs would split Y over various relations, while the dependency with Y as its lhs would keep Y as a unit.

In the first subsection the set of multivalued dependencies is 'split-left-hand-side free', in the second subsection we discuss the split left-hand-side property.

6.1.1. Split left-hand-side free sets of multivalued dependencies

In this subsection we show that every entity type equipped with a split left-hand-side free set of multivalued dependencies has an equivalent structural counterpart. The detailed proof of this theorem is beyond the scope of this paper. However, to give the reader a feeling for the structure that arises we will give a detailed analysis of a split left-hand-side free set of two multivalued dependencies. To enlighten the discussion, we will describe the various cases first in relational terms.

Let, in some relation R, the two multivalued dependencies be given by $X_1 \rightarrow\rightarrow Y_1 \mid Z_1$ and $X_2 \rightarrow\rightarrow Y_2 \mid Z_2$. We have the following complete set of cases:

1) $X_2 \subseteq Z_1, Y_2 \subseteq Z_1$.
2) $X_2 \subseteq Y_1, Y_2 \subseteq Z_1$.
3) $X_2 \subseteq Y_1, Y_2 \cap Y_1 \neq \emptyset, Y_2 \cap Z_1 \neq \emptyset$.
4) $X_2 \subseteq X_1, Y_2 = Y_1$.
5) $X_2 \subseteq X_1, Y_2 \cap Y_1 \neq \emptyset, Y_2 \cap Z_1 \neq \emptyset$.
6) $X_2 \cap X_1 \neq \emptyset, Y_2 \subseteq Y_1$.
7) $X_2 \cap X_1 \neq \emptyset, Y_2 \cap Y_1 \neq \emptyset, Y_2 \cap Z_1 \neq \emptyset$.

We analyse these cases with an example. The conflict free cases (the first five) and the cases with intersection anomaly (the last two cases) are analysed separately.

For the first five cases, we use $R = \{ABCDE\}$ and the first mvd as $A \rightarrow\rightarrow BC \mid DE$. This yields the entity types $e_1,\ldots,e_8$, such that $e_7 = r(e_3,e_4)$, $e_8 = r(e_5,e_6)$ and

† $Dep(X)$ are all multivalued dependencies with X as their lhs.

$e_1 = r(e_2,e_7,e_8)$; where $e_2 = A$, $e_3 = B$, $e_4 = C$, $e_5 = D$ and $e_6 = E$. The results can be found in Table 1, we will discuss two of them in detail.

case	mvd	formal	informal
1	$D \rightarrow\rightarrow E$ $mvd(e_5,e_6;e_{10})$	$e_9 = \{e_7\}_{e_2}$ $e_{10} = \{e_6\}_{e_5}$ $e_1' = r(e_9,e_{10})$	$R = \{BC\}_A\{E\}_D$
2a	$C \rightarrow\rightarrow E$; $mvd(e_4,e_6;e_1)$	$e_9 = \{e_6\}_{e_2,e_6}$ $e_{10} = \{e_5\}_{e_2}$ $e_1' = r(e_9,e_{10})$	$R = B\{D\}_A\{C\}_{C,A}$
2b	$C \rightarrow\rightarrow DE$ $mvd(e_4,e_8;e_1)$	$e_9 = \{e_8\}_{e_2,e_4}$ $e_1' = r(e_3,e_9)$	$R = B\{DE\}_{A,C}$
2c	$BC \rightarrow\rightarrow E$ $mvd(e_7,e_6;e_1)$	$e_9 = \{e_6\}_{e_2,e_7}$ $e_{10} = \{e_5\}_{e_2}$ $e_1' = r(e_9,e_{10})$	$R = \{D\}_A\{E\}_{BC,A}$
2d	$BC \rightarrow\rightarrow DE$ $mvd(e_7,e_8;e_1)$	$e_1' = \{e_8\}_{e_2,e_7}$	$R = \{DE\}_{A,BC}$
3a	$B \rightarrow\rightarrow CE \mid D$ $e_9 = r(e_4,e_6)$ $e_1 = r(e_2,e_7,e_8,e_9)$ $mvd(e_3,e_9;e_1)$	$e_{10} = \{e_6\}_{e_2,e_3}$ $e_{11} = \{e_4\}_{e_3}$ $e_{12} = \{e_5\}_{e_2}$ $e_1' = r(e_{10},e_{11},e_{12})$	$R = \{C\}_B\{D\}_A\{E\}_{A,B}$
3b	$B \rightarrow\rightarrow CDE$ $e_9 = r(e_4,e_5,e_6)$ $e_1 = r(e_2,e_7,e_8,e_9)$ $mvd(e_3,e_9,e_1)$	$e_{10} = \{e_8\}_{e_2,e_3}$ $e_{11} = \{e_4\}_{e_3}$ $e_1' = r(e_{10},e_{11})$	$R = \{C\}_B\{DE\}_{A,B}$
4	$A \rightarrow\rightarrow B \mid CDE$ $mvd(e_2,e_3,e_1)$	$e_9 = \{e_3\}_{e_2}$ $e_{10} = \{e_4\}_{e_2}$ $e_{11} = \{e_8\}_{e_2}$ $e_1' = r(e_9,e_{10},e_{11})$	$R = \{B\}_A\{C\}_A\{DE\}_A$
5	$AB \rightarrow\rightarrow CD \mid E$ $e_9 = r(e_2,e_3)$ $e_{10} = r(e_4,e_5)$ $e_1 = r(e_7,e_8,e_9,e_{10})$ $mvd(e_{10},e_4;e_1)$	$e_{11} = \{e_5\}_{e_2}$ $e_{12} = \{e_6\}_{e_2}$ $e_{13} = \{e_7\}_{e_2}$ $e_1' = r(e_{11},e_{12},e_{13})$	$R = \{BC\}_A\{D\}_A\{E\}_A$

TABLE 1. The Conflict Free Cases

The result of case 1 is obvious, because in normalisation theory the multivalued dependencies $A \rightarrow\rightarrow BC \mid DE$ and $D \rightarrow\rightarrow E$ on R yield as decomposition: $R = \pi_{ABC}(R) * \pi_{AD}(R) * \pi_{DE}(R)$

To enhance the reader's feeling, we give the results of all the possibilities in the cases 2 and 3. To illustrate the table, we will analyse 2a in detail:

Let r_1 and r_2 be two legal extensions of R under the two multivalued dependencies, $r_1 \cup r_2$ is a legal extension iff:

1) $\pi_A(r_1) \cap \pi_A(r_2) = \emptyset$

2) $\pi_C(r_1) \cap \pi_C(r_2) = \emptyset$

Following, let $\pi_A(r_1) = a_1$, $\pi_A(r_2) = a_2$, $\pi_C(r_1) = \{c_1,c_2\}$, $\pi_C(r_2) = \{c_2,c_3\}$ and let r be the smallest legal extension such that $r_1 \subseteq r$ and $r_2 \subseteq r$ then $\pi_E(\sigma_{a_1,c_1}(r)) = \pi_E(\sigma_{a_2,c_2}(r))$,

et cetera.

Next, let $\pi_C(r_1) = c_1$, $\pi_C(r_2) = c_2$, $\pi_A(r_1) = \{a_1,a_2\}$, $\pi_A(r_2) = \{a_2,a_3\}$ and let r be the smallest legal extension such that $r_1 \subseteq r$ and $r_2 \subseteq r$ then $\pi_E(\sigma_{a_1,c_1}(r)) = \pi_E(\sigma_{a_3,c_2}(r))$, et cetera.

Finally, let $\pi_A(r_1) \cap \pi_A(r_2) = \{a\}$, and let r be the smallest legal extension such that $r_1 \subseteq r$ and $r_2 \subseteq r$ then $\pi_D(\sigma_a(r)) = \pi_D(\sigma_a(r_1)) \cup \pi_D(\sigma_a(r_2))$.

So a constructive form of R is informally: $B\{D\}_A\{E\}_{A,C}$. The other results in the table are derived analogous.

For the last two cases, suffering from the intersection anomaly, we need a slightly different example, because the fixed multivalued dependency introduced above doesn't allow for the intersection anomaly to occur. Therefore, we choose the first mvd as $AB \rightarrow\rightarrow C \mid DE$; thus we need an entity type in our standard repertoire, viz: $e_9 = r(e_2,e_3)$ representing AB. Moreover, as both examples use AD, we will also use $e_{10} = r(e_2,e_5)$ representing AD and finally, $e_1 = r(e_4,e_6,e_9,e_{10})$.

case	mvd	formal	informal
6	$AD \rightarrow\rightarrow C \mid BE$ $mvd(e_{10},e_4;e_1)$	$e_{11} = \{e_9\}_{e_{9,10}}$ $e_1' = r(e_6,e_{11})$	$R = \{C\}_{AB,AD}E$
7	$AD \rightarrow\rightarrow CE \mid B$ $mvd(e_{10},e_3;e_1)$	$e_{11} = \{e_4\}_{e_9,e_{10}}$ $e_{12} = \{e_6\}_{e_{10}}$ $e_1' = r(e_{11},e_{12})$	$R = \{C\}_{AB,AD}\{E\}_{AD}$

TABLE 2. Cases with Intersection Anomaly

Although the intersection anomaly poses no problem in our model, we will briefly look into a solution for the anomaly given in [2]. The authors use an extra attribute and replace the two mvd's by three new ones to eliminate the intersection anomaly. In our model, this boils down to the definition of an new entity type as follows:
Let e_1 and e_2 be two entity types such that $mvd(e_1,e_3;e_4)$ and $mvd(e_2,e_3;e_4)$ show the intersection anomaly. Moreover, let e_5 represent the 'intersection' of e_1 and e_2. Then we can define an entity type $e_6 = r(e_1,e_2)$, representing those entities of type e_5 that specialise both in e_1 and e_2. Using e_6, we can remodel the multivalued dependencies in an analogous fashion to [2], removing the intersection anomaly. We make this observation as, in general, if an entity type e has two different specialisations e_1 and e_2, there is a relation between e_1 and e_2 denoting the e-entities that specialise in both. And our paradigm requires that this relation is explicitly modelled. Thus, intersection anomalies will not occur when the UoD is correctly modelled.

To summarise the analysis, every split left-hand-side free set of multivalued dependencies has a structural model:

THEOREM

Let e be an entity type, equipped with a split left-hand-side free set of multivalued dependencies m then there exists an entity type e', such that e' is equivalent to e with m.
□

6.1.2. Split left-hand-side sets of multivalued dependencies

In the previous subsection we have seen that the intersection anomaly poses no problems for a structural translation of a set of multivalued dependencies. The split left-hand-side anomaly, however, is more difficult to handle. To illustrate, we will use the two multivalued dependencies from our running example, i.e.: $BV \rightarrow\rightarrow P \mid C$ and $PC \rightarrow\rightarrow B \mid V$. Note that both left-hand-sides are split.

To unravel the structure of the extensions of R, let r_1 and r_2 be two disjoint valid extensions of R. They can be independent components iff $r_1 \cup r_2$ is also a valid extension. Straightforward calculation yields that $r_1 \cup r_2$ is valid if:

1) $\pi_{BV}(r_1) \cap \pi_{BV}(r_2) = \emptyset$

2) $\pi_{PC}(r_1) \cap \pi_{PC}(r_2) = \emptyset$

Let r_1 and r_2 satisfy these conditions. The insertion of a tuple t such that $\pi_{BV}(r_1) \cap \pi_{BV}(t) \neq \emptyset$ and $\pi_{PC}(t) \cap \pi_{PC}(r_2) \neq \emptyset$, causes a partition of $r_1 \cup r_2 \cup \{t\}$ such that neither r_1 nor r_2 is a subset of one of the new components; thus violating one of our requirements of a good structure.

A second option to model these two dependencies requires that the domain is structured in a different way:
Let $R' = \{\{B\}\{V\}\{P\}\{C\}\}$, then $t_1 = (\{B_1\}\{V_1\}\{P_1\}\{C_1\}\})$ and
$t_2 = (\{B_2\}\{V_2\}\{P_2\}\{C_2\}\})$ yield the tuples:

$t_3 = (\{B_1 \cup B_2\}\{V_1 \cup V_2\}\{P_1 \cap P_2\}\{C_1 \cap C_2\}\})$

$t_4 = (\{B_1 \cap B_2\}\{V_1 \cap V_2\}\{P_1 \cup P_2\}\{C_1 \cup C_2\}\})$.

Thus every multivalued dependency m introduces an operator $\times_m$ on the domain, such that every valid extension is closed under $\times_m$. Hence, every multivalued dependency turns the domain into a semi-group and an extension is valid iff it is a sub semi-group.

Thus a set of multivalued dependencies suffering from the split left-hand-side anomaly can be modelled by turning the domain in a mathematical structure reflecting the various semi-group structures. However, this requires that the user knows this structure as an extension should be a sub semi-group for all the semi-group structures. This seems an unnatural and unwieldy solution.

The third option to model such a set of multivalued dependencies is as derived data. Although this is an 'unsemantical' solution, it appears as the most natural solution in the light of the above discussion. Moreover, all the examples of the split left-hand-side anomaly we have seen in the literature, including the one above, are deductive in nature, thus implying a derived data solution.

7. Join Dependencies

In section 2 we argued that the unrestricted use of relational operators is a major cause for the deficiency of the relational model as a basis for a semantic description of the UoD. In particular, we did not introduce a join operator. Instead, all the relevant entity types are modelled explicitly. However, in the relational model the join is not only used as an operator, but also to express semantics via join dependencies. In this section we indicate how the semantics of a join dependency can be modelled structurally, i.e. using the structure of the type level and properties of the domains.

The definition of a join dependency in relational database theory is as follows:

DEFINITION

Let R = { A_1,...A_n }, let X_i be subsets of R, a relation r of R satisfies the join dependency X_1 * ... * X_m if $r = \pi_{X_1}(r)$ * ... * $\pi_{X_m}(r)$.

The simplest form of a join dependency is the multivalued dependency. We have shown in the previous section that a multivalued dependency can be modelled in a structural way using CSRs. Moreover, a well-known result in database theory is that an acyclic join dependency is equivalent to a conflict free set of multivalued dependencies [1] and, hence, using the results of the previous section, acyclic join dependencies can be modelled structural way as well.

Cyclic join dependencies are much harder. We will start with an example:
Let $R = ABC$, equipped with the join dependency $AB * BC * CA$. In order to unravel the structure of the extensions of R, let r_1 and r_2 be two disjoint valid extensions of R. They can be independent components iff $r_1 \cup r_2$ is also a valid extension. Straightforward calculation yields that $r_1 \cup r_2$ is valid if:

1) $\pi_{AB}(r_1) \cap \bigcup_{c \in \pi_C(r_2)} \pi_A\sigma_c(r_2) \times \pi_B\sigma_c(r_2) = \emptyset$

2) $\pi_{AB}(r_2) \cap \bigcup_{c \in \pi_C(r_1)} \pi_A\sigma_c(r_1) \times \pi_B\sigma_c(r_1) = \emptyset$

3) $\pi_{BC}(r_1) \cap \bigcup_{a \in \pi_A(r_2)} \pi_B\sigma_a(r_2) \times \pi_C\sigma_a(r_2) = \emptyset$

4) $\pi_{BC}(r_2) \cap \bigcup_{a \in \pi_A(r_1)} \pi_B\sigma_a(r_1) \times \pi_C\sigma_a(r_1) = \emptyset$

5) $\pi_{AC}(r_1) \cap \bigcup_{b \in \pi_B(r_2)} \pi_A\sigma_b(r_2) \times \pi_C\sigma_b(r_2) = \emptyset$

6) $\pi_{AC}(r_2) \cap \bigcup_{b \in \pi_B(r_1)} \pi_A\sigma_b(r_1) \times \pi_C\sigma_b(r_1) = \emptyset$

These conditions are intuitive, using the following equality:

$$\pi_{AB}(r) * \pi_{BC}(r) * \pi_{CA}(r) = \pi_{AB}(r) * \pi_{BC}(r) \cap \pi_{BC}(r) * \pi_{CA}(r) \cap \pi_{AC}(r) * \pi_{CA}(r).$$

This intuition can be generalised:

DEFINITION
A join dependency is called simple cyclic if its hypergraph † consists of exactly one cycle.

An easy result on simple cyclic join dependencies is:

LEMMA
Every simple cyclic join dependency is equal to the intersection of a set of acyclic join dependencies □

Moreover, we have that

LEMMA
$(R_1 \cap R_2) * R_3 = (R_1 * R_3) \cap (R_2 * R_3)$ □
This leads to the following theorem:

THEOREM
Every join dependency can be written as the intersection of a set of acyclic join dependencies. □

Thus, for every join dependency, the requirements for components to be independent will be similar to the ones in our simple example.

As with sets of multivalued dependencies suffering from the split left-hand-side anomaly, the requirements show that the decomposition of the extensions in independent components cannot lead to a good structure. Once again the dependency can be modelled by giving the domain a complicated structure. But the derived data solution appears again to be the most natural.

---

† The hypergraph of a join dependency has as nodes the attributes of the relation and as its hyperedges the components of the jd.

8. Summary

In this paper we have introduced a functional datamodel with structured domains, and showed that the structure of the type level can naturally be translated into a structure at the extension level. Moreover, using the structured domains and causal-set-relations we showed that constraints that manifest at the extension level can be cast into a structural constraint at the type level. It means that guaranteeing the data organisation prescribed by the entity type graph ensures validity of the constraints in our database. Furthermore, we used this theory to study sets of multivalued dependencies suffering from the split left-hand-side anomaly formally; it was suggested that the underlying cause is the deductive nature of such a set of multivalued dependencies. In conclusion, we have bridged the gap between database intension and extension using a formal model with design axioms that highlight the semantic bonds in the Universe of Discourse.

The research reported in this paper can be extended in several ways. Since we have developed a formal datamodel that explicates the semantics of the application area, it becomes possible to formally compare alternative descriptions of those areas more readily. Finally, an architecture of a database design aid based on this model can be envisioned.

Acknowledgement

We would like to thank Lambert Meertens for constructive criticism on earlier versions of this paper.

References

[1] Beeri, C., Fagin, R., Maier, D., Mendelzon, A., Ullman, J., and Yannakakis, M., "Properties of Acyclic Database Schemes," *ACM STOC*, pp.355-362, 1981.

[2] Beeri, C. and Kifer, M., "Elimination of intersection anomalies from database schemes," *Journal of the ACM*, vol. 30, July 1986.

[3] Gallaire, H. and Minkers), J., in Logic and Databases, Plenum Press, New York (1978).

[4] Gallaire, H., Minker, J., and Nicolas, J-M., "Logic and Database: A Deductive Approach," *ACM Computing Surveys*, vol. 16, no. 2, pp.153-185, June 1984.

[5] Hodges, Wilfrid, "What is a structure theory," *The Bulletin of the London Mathematical Society*, vol. 19, pp.209-237.

[6] Maier, David, "Null Values Partial Information and Database Semantics," pp. 371-438 in The Theory of Relational Databases (1983).

[7] Maier, D., "Why Object Oriented Databases Can Succeed Where Others Have Failed," *1986*, p.227.

[8] Reiter, R., "Databases: a Logical Perspective," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling. SIGPLAN Notices*, vol. 16, no. 1, pp.174-176, Jan 1981.

[9] Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory," pp. 191-233 in On Conceptual Modelling, ed. J.W. Schmidt (1984).

[10] Siebes, Arno and Kersten, Martin L., "Using Design Axioms and Topology to Model Database Semantics," *Proceedings of the 13th International Conference on Very Large Databases*, pp.51-59, 1987.

[11] Stonebraker, M., "Object Management in POSTGRES Using Procedures," *1986*, pp.66-73.