11th ICDE 1995, Taipei, Taiwan

Record Subtyping in Flexible Relations by means of Attribute Dependencies

Christian Kalus Peter Dadam

Universität Ulm
Fakultät für Informatik
Abteilung Datenbanken und Informationssysteme
89069 Ulm, Germany

Abstract

The model of flexible relations supports heterogeneous sets of tuples in a strongly typed way. The elegance of the standard relational model is preserved by using a single, generic scheme constructor.In each model supporting structural variants the shape of some part of a heterogeneous scheme may be determined by the contents of some other part of the scheme. We formalize this relationship by a certain kind of integrity constraint we have called "attribute dependency" (AD). We motivate how ADs can be used, besides their application in type and integrity checking, to incorporate record subtyping into our extended relational model. Moreover, we show that ADs yield a stronger assertion than the traditional record subtyping rule as they consider interdependencies among refinements. We discuss how ADs are related to query processing and how they may help to identify redundant operations.

1. Introduction

The relational model as defined by Codd [6] forms the base of most contemporary data models. It constructs a database as a set of relations, a relation being defined over a set of attributes called its scheme. The instance of a relation is a set of tuples where each tuple is a mapping from the scheme attributes to values of given (atomic) domains.

The constraint of homogeneity, i.e. the fact that all tuples of a relation are defined over the same set of attributes, does often not meet the intuition of a relation as a container of *related* entities. Take an address (e.g. of a person's record) as a simple example. Each address comprises a zip code and a town. The town-local part of the address may be either a post-office box number or a street and, if it is a street, it is sometimes followed by a house number, sometimes not. This little example already exhibits many facets: ZipCode and Town are *unconditioned* or *homogeneous* components as they are always present. The town-local part is a *disjoint union* of PostOfficeBoxNumber and Street while Street may be accompanied by the *optional attribute* HouseNumber.

Another form of attribute relationship can be motivated by the "electronic communication part" of an address which is composed of a telephone-number, a FAX-number, and an electronic-mail-address. At least one of these three attributes should be present to constitute an electronic communication address, but two or all three of the attributes are allowed, too. This *non-disjoint union* is another relationship type between attributes, and many other forms of relationships can be found in addition.

The relationships mentioned above are based on the pure existence of attributes. I.e. from the sole existence/absence of a certain attribute we draw conclusions about the existence or absence of other attributes. In addition to these *existence-based attribute relationships* there occur *value-based attribute relationships* which take the influence of values (in some attributes) on the existence of (other) attributes into account.

Take an employee entity possessing the attributes salary and jobtype as an example. In addition,

- if the value of jobtype is 'secretary' then the attributes typing-speed and foreign-languages are present.
- if the jobtype is 'software engineer' the employee is further described by the products he is in charge of and the programming-languages he knows.
- if the jobtype is 'salesman' he possesses a sales-commission and again the products he is in charge of.

Another example is that the existence of a maiden-name is determined by the appropriate values in the attributes sex and marital-status.

The overall aim of the model of flexible relations is to bridge the gap between semantic data models and operational data models. Therefore it captures the attribute relationships described above, yet it utilizes a single, generic scheme construct to preserve the elegance of the relational model.

The paper is focussing on value-based attribute relationships. We will introduce a notation we have called *attribute dependencies* (ADs) that enables us to model these relationships as integrity constraints. We will motivate the usage of ADs, particularly their ability to model a strong notion of subtyping. Besides this employment their connection to constructs of semantic data models, the benefit of attribute dependencies in type checking and in host language coupling is discussed. The behaviour of ADs, especially in connection with a query language, will be formally described by an axiom system.

The rest of the paper is structured as follows: Section 2 introduces some basic notations of the model of flexibe relations needed as an environment in which attribute dependencies are to be integrated. The different purposes which ADs serve in our model, including subtyping, are discussed in section 3. In section 4 an axiom system for the derivation of attribute dependencies is developed and shown to be sound and complete. Section 5 compares our approach to related ones, while section 6 concludes with a summary and an outlook.

2. Attribute dependencies

2.1 Basic notations of the model of flexible relations

The elegance of the relational model is mainly due to the fact that it gets along with a single type constructor. To preserve this elegance as much as possible, we looked for an extended type constructor enabling us to describe the various variant (and also non-variant) structures in a single, generic fashion. It is obvious that specifying a scheme as a set of attributes, as the relational model does, is not expressive enough to model arbitrary attribute relationships. To achieve this goal we enhanced the scheme notation in the following way: a scheme is now composed of a set of attributes accompanied by a cardinality constraint in the form of two integer values determining how many components of the set have *at least* to be taken and how many components of the set are allowed *at most*. If we describe this construct as a three-tuple

< at-least value, at-most value, set of attributes >

then the various constructs introduced in section 1 can be expressed in the following way[1]

- a traditional relational scheme with attributes $A_1, \dots, A_n$ is denoted by $< n, n, \{ A_1, \dots, A_n \} >$, i.e. at least n and at most n (and therefore exactly n) of the attributes have to be present.
- a disjoint union of attributes $A_1, \dots, A_n$ is modeled by $< 1, 1, \{ A_1, \dots, A_n \} >$ telling that exactly one of the attributes may appear.
- a non-disjoint union of attributes $A_1, \dots, A_n$ is described by $< 1, n, \{ A_1, \dots, A_n \} >$, i.e. the electronic communication address of section 1 is expressed by < 1, 3, { tel-number, FAX-number, email-address } >.

[1] Let as usual be $\mathcal{U}$ the universe of attributes, A, B ... and A_i be single attributes, and V, ..., Z be attribute sets. Let XY denote the union of the attribute sets X and Y and treat attributes as singleton attribute sets when sets of attributes are expected. Tuples will be denoted by < ... >.

The above notation is not completely satisfying yet: a union might appear as only a part of a scheme, the variants in a union do not need to be single attributes but can be relational schemes, variants again, and so on. Therefore we have to extend our notation allowing the set components to be either single attributes or again three-tuples of our notation. This final version of a *flexible scheme* is presented by a more abstract example.

Example 1 An application demanding tuples with attributes A and B (unconditioned), either attribute C or D (i.e. a disjoint variant between C and D) and "some" of E, F and G (a non-disjoint union of E, F and G) yields the following flexible scheme FS

$$FS = < 4, 4, \{ A, B, < 1, 1, \{ C, D \} >, < 1, 3, \{ E, F, G \} > \} >$$

□

A flexible scheme is a very compact notation. For the purpose of a basic understanding one can unfold a flexible scheme yielding the allowed attribute combinations. As this unfolding can be interpreted as building the *disjunctive normal form* of a flexible scheme FS, we will refer to it as *dnf*(FS). Forming the DNF of the scheme of example 1 yields

dnf(FS) = { ABCE, ABDE, ABCF, ABDF, ABCG, ABDG, ABCEF, ABDEF, ABCEG, ABDEG, ABCFG, ABDFG, ABCEFG, ABDEFG }

Note that this unfolded version of a flexible scheme corresponds to the "set of objects" idea of Ed Sciore [13] (see also [11], chapter 12). The little example above should suffice as a motivation to find a compact description for variant schemes.

Now it is easy to define the domain of a flexible scheme. If *Tup*(X) denotes the set of tuples for a given attribute set X, then $dom(FS) = \bigcup_{X \in dnf(FS)} Tup(X)$. A *flexible relation* FR can then be defined as a two-tuple FR = < FS, inst > with *scheme*(FR) = FS being a flexible scheme and *inst*(FR) = inst being the instance of the relation, a finite set of tuples satisfying $inst(FR) \subset dom(scheme(FR))$. As a flexible scheme does not uniquely determine the shape of its tuples we assume the existence of a function *attr*(t) yielding the attribute set X, tuple t is defined on (of course $attr(t) \in dnf(FS)$ iff $t \in dom(FS)$).

2.2 Definition of attribute dependencies

Up to now a flexible scheme considers existential relationships of attributes and determines thus the basic shape of tuples and instances. Value-based constraints are not yet taken into account, a flexible scheme is therefore, following the notation of [12], a *primitive* scheme and a flexible relation only a *possible* relation (instance). The examples in section 1 have shown that flexible relations demand, besides the known types of constraints, a certain class of constraints concerning the variant structure of tuples. Referring to the *jobtype*-example of section 1 we may say that the value of the attribute jobtype determines the existence of the attributes in Y = { typing-speed, foreign-languages, products, programming-languages, sales-commission } in the way that

(1) t(jobtype) = 'secretary' $\rightarrow$
attr(t) $\cap$ Y = { typing-speed, foreign-languages }

(2) t(jobtype) = 'software engineer' $\rightarrow$
attr(t) $\cap$ Y = {products, programming-languages }

(3) t(jobtype) = 'salesman' $\rightarrow$
attr(t) $\cap$ Y = { products, sales-commission }

To prepare the formal definition of an attribute dependency consider the following points. While in the example above there is only one determining attribute (jobtype), in general there may be several ones (take sex and marital-status determining the existence of a maiden-name). Therefore we should say that the contents in the attribute set X determines which attributes in the attribute set Y exist. This general assertion can be refined by considering the legal variants explicitely. Each variant consists of an attribute set $Y_i \subseteq Y$ ($i=1..n$, n being the number of variants) and is determined by a set of values $V_i \subseteq Tup(X)$ with the obvious meaning that the attribute set Y_i occurs in a tuple t whenever $t[X] \in V_i$. When there is no V_i such that $t[X] \in V_i$ then it is intuitive to demand that tuple t does not possess any attribute of Y. Considering this we obtain the definition of an *attribute dependency*[2]

Definition 2.1 "explicit attribute dependency"

An explicit attribute dependency EAD has the syntactical form

$$\text{EAD} = < X \xrightarrow{\text{exp. attr}} Y, \{ V_1 \xrightarrow{\text{exp.attr}} Y_1, \ldots, V_n \xrightarrow{\text{exp.attr}} Y_n \} >$$

$$\begin{aligned} \text{where}\quad & X \subset \mathcal{U}, V_i \subseteq Tup(X) && (i = 1 .. n), \\ & Y \subset \mathcal{U}, Y_i \subseteq Y && (i = 1 .. n), \\ & i \neq j \rightarrow V_i \cap V_j = \varnothing && (i,j = 1 .. n) \end{aligned}$$

A flexible relation FR is said to satisfy the explicit attribute dependency EAD if $\forall t \in inst(\text{FR})$:

$$(\exists i : t[X] \in V_i) \rightarrow attr(t) \cap Y = Y_i$$
$$\wedge\ (\forall i : t[X] \notin V_i) \rightarrow attr(t) \cap Y = \varnothing \qquad \square$$

Example 2 The *jobtype*-example is formulated in EAD-notation by

< {jobtype} $\xrightarrow{\text{exp. attr}}$ { typing-speed, foreign-languages, products, programming-languages, sales-commission },
{ < jobtype : 'secretary' > $\xrightarrow{\text{exp. attr}}$
{ typing-speed, foreign-languages } ,
< jobtype : 'software engineer' > $\xrightarrow{\text{exp. attr}}$
{ products, programming-languages } ,
< jobtype : 'salesman' > $\xrightarrow{\text{exp. attr}}$
{ products, sales-commission } } >

$\square$

3. Usage of attribute dependencies

There are several streams motivating the use of ADs. The first one is to integrate semantic type constructs into an operational data model, thus bridging the gap between semantic and operational data models. Secondly we show that ADs may be used to incorporate subtyping into a relational data model. In addition, we discuss how ADs may be applied in decomposition and query evaluation.

3.1 Mapping of entity-relationship concepts onto flexible relations

Specialization is one of the enhanced entity relationship concepts ([7], chapter 15). A specialization that is encoded in the entity itself is called a *predicate defined specialization.* If one replaces the predicate p_i of the i-th specialization by its extension V_i, i.e. $V_i = \{ v \mid p_i(v) \text{ is true} \}$, then an attribute dependency is a one-to-one mapping of a predicate defined specialization. ER models further divide specialization into *disjoint* versus *overlapping* subclasses and *total* versus *partial* subclasses. This classification can be inferred from ADs as well: the variants of an AD are *disjoint* if $Y_i \cap Y_j = \varnothing$ ($i \neq j$) and they are *total* if $\bigcup_{i=1..n} V_i = Tup(X)$. The benefit of mapping these ER constructs onto the model of flexible relations is that they can now be exploited operationally.

The most important operational use of ADs is their application in type-checking, which is a central point of our model. Flexible schemes do serve this purpose already better than relational schemes as existential attribute relationships are already captured by them (see section 2). However, value-based dependencies cannot be type-checked by flexible schemes. For example, there is no scheme which would reject the tuple

[2] In section 4 we will use a slightly modified definition. To distinguish both we call the following definition an *explicit attribute dependency.*

< .. jobtype : 'salesman', typing-speed : high, foreign-languages : { french, russian } >

as { ... , jobtype, typing-speed, foreign-languages } is a valid attribute combination. The fact that jobtype = 'salesman' requires different attributes to be present has to be checked with the AD of example 2. Type checking based on ADs is initiated during insertion, update[3], and data retrieval, which will be discussed in more detail.

3.1.1 Decomposition along ADs In [7] four translation methods for predicate defined specializations into relations are described. Two of these methods result in a single relation with plenty of null values. Moreover, artificial attributes indicating the current variant have to be introduced - and have to be interpreted by the user. The benefit of our model is obvious: flexible schemes together with attribute dependencies relieve the user of the burden to set and control the correct variant.

The third and fourth translation method horizontally/vertically decompose the entity along the specialization. These techniques can be regarded as extensions of the traditional horizontal/vertical decomposition [5] to support structural variants. To restore the entity, an outer union instead of a simple union (horizontal decomposition) and a multiway join instead of a natural join (vertical decomposition) have to be performed.

3.1.2 Query optimization by the aid of ADs *Qualified relations* are used to extend algebraic equivalences to deecomposed relations [5]. Again we may say that a relation together with an AD is an extension of a qualified relation to support structural variants. As for qualified relations we can exploit each selection concerning the determining attributes of an AD to draw conclusion about redundant operations, e.g. unnecessary joins with variants that are known to be excluded. See [5,p.103ff] for a list of query rewrite rules.

Another potential for optimization are type guards. Each model supporting heterogeneous collections possesses operations do not preserve the most specific type of an entity (see e.g. [3] among others). Type guards restore the lost type information by checking if an entity has a certain type or if certain attributes are available. ADs cannot only be used to implement type guards but can also recognize redundant type guards when additional information is sufficient to determine a more specific type.

3.2 Semantic preserving subtyping through ADs

At first glance a predicate defined specialization can as well be described by the traditional record subtyping rule (see [4] among many others)

$$\frac{t_i \leq u_i \ (i=1..n)}{< a_1 : t_1 , \dots , a_n : t_n , \dots , a_m : t_m > \leq < a_1 : u_1 , \dots , a_n : u_n >}$$

Let us first show that this inclusion rule can be expressed with an attribute dependency. Therefore, consider a flexible scheme FS with *attr*(FS) = W and let EAD = $< X \xrightarrow{\text{exp. attr}} Y , \{ V_1 \xrightarrow{\text{exp. attr}} Y_1 , \dots , V_n \xrightarrow{\text{exp. attr}} Y_n \} >$ be an attribute dependency for FS. Then the corresponding supertype contains the attributes W - Y and the domain of X consists of *Tup*(X), i.e. the domain of X is unrestricted in the supertype. Further we can derive from EAD that there are *n* subtypes possessing the attributes $(W - Y) \cup Y_i$, having the domain of X restricted to V_i (i=1..n). We may therefore say that attribute dependencies incorporate record subtyping into the model of flexible relations.

Now, what is the benefit of using attribute dependencies instead of the traditional subtyping rule? This rule is obviously sufficient to state that *secretary*, *salesman* and *software engineer* type are subtypes of a more general *employee* type.

Example 3 The following *employee* type and its predicate defined subtypes could have been inferred from the *jobtype*-EAD of example 2:

employee_type = < ... , salary : float, jobtype : { 'secretary', 'software engineer', 'salesman' } >
secretary_type = < ... , salary : float, jobtype : { 'secretary' }, typing-speed : ... , foreign-lang : ... >
salesman_type = < ... , salary : float, jobtype : {'salesman' }, products : ... , sales-commission : ... >
softw_eng_type = < ... , salary : float, jobtype : {'software eng' }, products : ... , programming-languages : ... > □

Note that for each of the three subtypes there are two type changes causing the subtype relation: The domain of jobtype is restricted and some attributes are added to the subtypes. These simultaneous type changes are considered to be purely accidental by the record subtyping rule. The type < ... , salary : float > (without attribute jobtype) is therefore treated as a valid supertype of the subtypes presented above, although the connection between the determining attribute jobtype and the subtypes is destroyed. To prevent this from happening or at least to notify the loss of this connection, it is necessary to treat these type changes as causal related, like attribute dependencies do.

3.3 Further usage of attribute dependencies

A last usage, which shall only be sketched here, is that ADs are an encoding of general sums (see [10] as an entry point). This equivalence can be exploited when embedding of flexible relations into programming languages is discussed. It can be shown that a flexible

[3] While there are no further type-related consequences when the salary of an employee is updated, the change of his jobtype causes a type change, too.

scheme can be translated into an appropriate programming language type (e.g. a variant record in PASCAL) if each existential attribute relationship is accompanied by an AD. If necessary, this can be obtained by introducing artificial ADs with artificial determining attributes.

The applications discussed in this section pose different requirements on ADs. In some cases, like insertion, whole tuples of a flexible relation are considered. Here, it is sufficient to apply the ADs specified in the scheme. But most applications, like update, retrieval or programming language embedding, are referring only to parts of a tuple or to tuples which may even have been transformed by (query language) operations. Therefore it is also necessary to know how ADs behave under transformations. This question is also the central point when ADs are exploited for (semantic preserving) subtyping. To answer this question we will develop an axiom system for the implication of attribute dependencies. This is done in the next section.

4. Axiom systems for attribute dependencies

4.1 An axiom system regarding ADs separately

Before we define the axiom system we slightly modify the definition of an AD. This is only done for the sake of readability and to better illustrate the similarity to other forms of dependencies, but does not change its intention.

From definition 2.1 we can derive that, given an explicit attribute dependency $< X \xrightarrow{\text{exp. attr}} Y \ldots >$, whenever two tuples t_1, t_2 agree on X, then they possess the same subset of Y as attributes. So we can define

Definition 4.1 "attribute dependency"

Let $X, Y \subset \mathcal{U}$. A flexible relation FR is said to satisfy the attribute dependency $X \xrightarrow{\text{attr}} Y$ if $\forall t_1, t_2 \in \mathit{inst}(\mathrm{FR})$:

$$X \subseteq \mathit{attr}(t_1) \wedge X \subseteq \mathit{attr}(t_2) \wedge t_1[X] = t_2[X] \rightarrow \mathit{attr}(t_1) \cap Y = \mathit{attr}(t_2) \cap Y$$

□

The axiom system $\mathcal{A}$ that manages attribute dependencies consists of the following four rules:

(A1) $X \xrightarrow{\text{attr}} YZ \vdash X \xrightarrow{\text{attr}} Y$ (projectivity)

(A2) $\{ X \xrightarrow{\text{attr}} Y, X \xrightarrow{\text{attr}} Z \} \vdash X \xrightarrow{\text{attr}} YZ$ (additivity)

(A3) $\emptyset \vdash X \xrightarrow{\text{attr}} Y$ if $Y \subseteq X$ (reflexivity)

(A4) $X \xrightarrow{\text{attr}} Y \vdash XZ \xrightarrow{\text{attr}} Y$ (left augmentation)

A remarkable point about this rule system is that transitivity is not valid for attribute dependencies. This stems from the fact that we do not draw any conclusion about the contents of the determined attributes. The correctness of the axiom system is stated by the following theorem[4]

Theorem 4.1 $\mathcal{A}$ is a sound, complete and non-redundant system of axioms for the implication of attribute dependencies.

□

Note that all rules could have been defined for explicit attribute dependencies as well. For example, the additivity rule would be

$$\begin{aligned}
&\{ < X \xrightarrow{\text{exp. attr}} Y, \{ V_{1_1} \xrightarrow{\text{exp. attr}} Y_{1_1}, \ldots, V_{1_n} \xrightarrow{\text{exp. attr}} {}_{1_n} \} >, \\
&< X \xrightarrow{\text{exp. attr}} Z, \{ V_{2_1} \xrightarrow{\text{exp. attr}} Z_{2_1}, \ldots, V_{2_m} \xrightarrow{\text{exp. attr}} Z_{2_m} \} > \} \\
&\vdash < X \xrightarrow{\text{exp. attr}} YZ, \{ V_{1_1} \cap V_{2_1} \xrightarrow{\text{exp. attr}} Y_{1_1} Z_{2_1}, \ldots, \\
&\qquad V_{1_n} \cap V_{2_m} \xrightarrow{\text{exp. attr}} Y_{1_n} Z_{2_m} \} >
\end{aligned}$$

This lengthy definition hampers of course the readability, thus making the abbreviated definition of attribute dependencies more favorable for our purpose. Nevertheless we stress again that the presented axiom system works for explicit attribute dependencies as well.

The following example motivates the arguments of section 3.1.2 bydemonstrating how a type guard can be recognized being redundant.

Example 4 Imagine a query containing a selection with the formula "salary > 5000 AND jobtype = 'secretary' " followed by a type guard checking for the presence of the attribute typing-speed. The redundancy of the type guard can be shown by the following derivation based on the *jobtype*-EAD defined in example 2:

Projecting the right side of the *jobtype*-EAD onto { typing-speed } yields (cf. rule (A1))

$<\{\text{jobtype}\} \xrightarrow{\text{exp. attr}} \{\text{typing-speed}\}$, { < jobtype : 'secretary' > $\xrightarrow{\text{exp. attr}}$ {typing-speed} } >

Augmenting the left side of this EAD with the attribute salary yields (cf. rule (A4))

$< \{ \text{jobtype, salary} \} \xrightarrow{\text{exp. attr}} \{\text{typing-speed}\}$,
{ < jobtype : 'secretary' , salary : s >
$\xrightarrow{\text{exp. attr}}$ {typing-speed} } >

where s is an arbitrary value of *dom*(salary). We may conclude that the presence of the attribute typing-speed can be deduced from the selection formula and that the type guard is therefore redundant.

□

[4] Due to space limitations we skip the proof of soundness and non-redundancy. The completeness is subsumed by the completeness of the extended axiom system $\mathcal{AF}$ (see theorem 4.2) which is proved in the appendix. The complete proofs can be found in [8].

4.2 An extended axiom system capturing functional and attribute dependencies

There are several reasons why one should regard FDs together with ADs. The first and most practical reason origins from the discussion how to embed flexible relations into programming languages. Take PASCAL as an example: Although its variant record type resembles ADs, there are some syntactic restrictions that have to be obeyed. One of them is that only a single attribute may appear as determinant of a variant record. Iamgine an attribute dependency $X \xrightarrow{attr} Y$ with X consisting of at least two attributes. There is an intuitive way to circumvent the aforementioned syntactic restriction: Introduce an artificial attribute A, replace $X \xrightarrow{attr} Y$ by $A \xrightarrow{attr} Y$ and make the value of A dependent on the value of X, i.e. extend the constraints by $X \xrightarrow{func} A$. The validity of this and other replacements may be verified by the aid of a rule system combining functional and attribute dependencies.

Secondly, the assertion of the reflexivity rule for ADs is too weak as X does not only determine the existence of Y if Y is a subset of X but also its value (see reflexivity rule (F1) below). In a combined system we can sharpen this assertion, making the axiom system more expressive.

As a preliminary we have to adapt the notion of FDs to fit into our model. The adaption simply consists of the adding of a type guard "$X \subseteq attr(t)$" as the access of values must be preceded by a type guard when structural variants are allowed. The axiom system, consisting of the reflexivity rule, the transitivity rule, and the augmentation rule (see (F1), (F2) and (F3) below), is borrowed from [16,p.384ff]. Its soundness and completeness is not affected by the adaption to our model.

Definition 4.2 "FD (adapted to flexible relations)"

Let $X,Y \subset \mathcal{U}$. A flexible relation FR is said to satisfy the functional dependency $X \xrightarrow{func} Y$ if $\forall t_1, t_2 \in inst(FR)$:

$$X \subseteq attr(t_1) \wedge X \subseteq attr(t_2) \wedge t_1[X] = t_2[X] \rightarrow$$
$$Y \subseteq attr(t_1) \wedge Y \subseteq attr(t_2) \wedge t_1[Y] = t_2[Y]$$

□

The combined axiom system $\mathcal{AF}$ for functional and attribute dependencies consists of the following seven rules:

(AF1) $X \xrightarrow{func} Y \vdash X \xrightarrow{attr} Y$ (subsumption)

(AF2) $\{X \xrightarrow{func} Y, Y \xrightarrow{attr} Z\} \vdash X \xrightarrow{attr} Z$ (combined transitivity)

(A1) $X \xrightarrow{attr} YZ \vdash X \xrightarrow{attr} Y$ (projectivity)

(A2) $\{X \xrightarrow{attr} Y, X \xrightarrow{attr} Z\} \vdash X \xrightarrow{attr} YZ$ (additivity)

(F1) $\emptyset \vdash X \xrightarrow{func} Y$ if $Y \subseteq X$ (reflexivity)

(F2) $X \xrightarrow{func} Y \vdash XZ \xrightarrow{func} YZ$ (augmentation)

(F3) $\{X \xrightarrow{func} Y, Y \xrightarrow{func} Z\} \vdash X \xrightarrow{func} Z$ (transitivity)

Theorem 4.2 $\mathcal{AF}$ is a sound, complete and non-redundant system of rules for the implication of functional and attribute dependencies.

□

Referring to the motivation for this combined rule system, one can see that the pragmatic "work-around" for PASCAL's variant record type is valid (see the combined transitivity rule (AF2)).

In addition we could save two rules, thus making the axiom system more compact. The reflexivity rule (A3) and the left augmentation rule (A4), still needed in $\mathcal{A}$ to produce a complete axiom system, can now be inferred from $\mathcal{AF}$ with rather simple derivation sequences.

4.3 Attribute dependencies versus record subtyping - the impact of transformations

At the end of this section we want to discuss the differences and similarities of attribute dependencies and traditional record subtyping. To do so, we sketch, with the aid of the developed axiom system, how transformations affect attribute dependencies. As the formal description of the algebra for the model of flexible relations is beyond the scope of this paper, we will rely upon well-known algebraic operator, providing the intuitive meaning in our model, too.

The most remarkable difference between record subtyping and attribute dependencies shows the project operator. While record subtyping tells us that any projection yields a valid supertype [14], two cases have to be discriminated for attribute dependencies: Suppose that a flexible relation is to be projected onto the attribute set X. As there is no rule telling us that an attribute dependency may hold if attributes at its left side are omitted, all $V \xrightarrow{attr} W$ with $V \not\subseteq X$ are invalidated. If on the other hand $V \subseteq X$ then the projection rule tells us that $V \xrightarrow{attr} W \cap X$ holds in the projection.

The two notions of subtyping perform similar when the result "enlarges" the input relation(s). This holds e.g. for the extension operator and the cartesian product. The behaviour of attribute dependencies under algebraic transformations can be summarized as follows:

Theorem 4.3 Let $ads(FR)$ be the set of attribute dependencies that hold in the flexible relation FR. The following rules describe the propagation of attribute dependencies:

(1) $ads(FR_1 \times FR_2) = ads(FR_1) \cup ads(FR_2)$

(2) $ads(\pi_X(FR)) = \{ V \xrightarrow{attr} W \cap X \mid V \xrightarrow{attr} W \in ads(FR) \wedge V \subseteq X \}$

(3) $ads(\sigma_F(FR)) = ads(FR)$

(4) $ads(FR_1 \cup FR_2) = \emptyset$

(5) $ads(FR_1 - FR_2) = ads(FR_1)$

□

The theorem shows that besides the projection the union operator causes a problem, too. First of all note that without appropriate precautions no dependency at all holds in the result of a union, as one cannot decide from which input relation the tuples do come from, i.e. this is not a special problem of attribute dependencies. To make dependencies hold, one has to tag both input relations before performing the union. The tagging can be realized with the extension operator $\varepsilon_{A:a}(FR)$, which extends each tuple of FR by attribute A with value 'a'.

The left augmentation rule allows us to replace any $X \xrightarrow{attr} Y$ occuring in one of the input relations by $AX \xrightarrow{attr} Y$ in its extended counterpart. The extended attribute dependencies now remain valid in the result relation, i.e. we obtain

(6) $ads((\varepsilon_{A:a_1}(FR_1)) \cup (\varepsilon_{A:a_2}(FR_2))) = \{ AX \xrightarrow{attr} Y \mid X \xrightarrow{attr} Y \in ads(FR_1) \vee X \xrightarrow{attr} Y \in ads(FR_2) \}$

5. Related work

The concept of subtyping is present in each object-oriented data model [2]. To obey the desirable closure property, these data models should discuss how the subtype relation is affected by algebraic tranformations. This has been done e.g. for the COCOON model in [14] and for the ENCORE model in [15]. Differences to our notion of record subtyping have been discussed in section 3.2, the comparison of the behaviour under transformations is contained in section 4.3.

From the viewpoint of data dependencies, several attempts have been made to consider value-oriented dependencies in the presence of null values ([9], [17]), but considering a merely existential consequence without any value-oriented assertion seems to be a novelty in the context of an operational data model.

An approach which pursuits the idea of [7] to decompose an entity subtree into a master relation and depending relations containing the variant information is the "multirelation" model of Ahad and Basu [1]. They improve the decomposition by keeping track of the connection between the master relation and the depending relations so that the restoration of the complete information can be automated. The recording of the connection between master and depending relation is done via so-called "image attributes", attributes possessing relation names as their domain. Image attributes can be regarded as a special case of an attribute dependency using a single artificial attribute as determinant, this approach is therefore completely covered by our results.

6. Summary and outlook

In this paper we have motivated attribute dependencies as constraints naturally arising when variant structures are considered. It turned out that they can be used to incorporate record subtyping in a relational model, yielding an even stronger notion of subtyping, as attribute dependencies consider causal connections between type refinements. In addition we could show that the several forms of specialization arising in (enhanced) entity relationship models can be one-to-one mapped onto attribute dependencies, with the benefit that they can be operationally employed in type and integrity checking.

Our approach to model these features as a dependency allowed us develop an axiom system for their derivation. The axiom system has been shown to be sound, non-redundant and complete, which enables us to precisely predict the effect of arbitrary transformations (like query language operations) on attribute dependencies. Furthermore the connection between attribute and functional dependencies has been discussed and an extended axiom system capturing both forms of dependencies has been evaluated.

Although attribute dependencies were regarded in the context of the model of flexible relations they are rather loosely tied to particularities of this model (see section 2). Thus there seem to be no major problems to integrate attribute dependencies into other data models supporting variant structures or appropriate null values.

In this paper the connection between attribute dependencies and subtyping has been shown. However, the second axiom system presented here, capturing both functional and attribute dependencies, has been motivated differently (see section 4.2). Additional work should be put on the question if this combined rule system can be exploited to put further semantics into the subtype relationship.

Acknowledgements

The authors would like to thank Marc Scholl, Franz Schweiggert and Klaus Gassner for valuable suggestions on a previous version of the paper. Also thanks go to the anonymous referees for their valuable comments.

References

[1] R. Ahad, A. Basu, "ESQL: A Query Language for the Relation Model Supporting Image Domains", 7th Int. Conf. on Data Engineering, April 1991, pp. 550 - 559

[2] M. Atkinson, F. Bancilhon et al., "The Object-Oriented Database System Manifesto", 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan, Dec. 1989, pp. 40 - 57

[3] P. Buneman, A. Ohori, "A Type System that reconciles Classes and Extents", 3rd Int. Workshop on Database Programming Languages, Greece, 1991, pp. 191 - 202

[4] L. Cardelli, P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys, vol. 17, no. 4, Dec. 1985, pp. 471 - 522

[5] S. Ceri, G. Pelagetti, "Distributes Databases, Principles and Systems", McGraw-Hill Book Company, 1984

[6] E.F. Codd, "A Relational Model for Large Shared Data Bases", Comm. of the ACM, vol. 13, no. 6, June 1970

[7] R. Elmasri, S.B. Navathe, "Fundamentals of Database Systems", Benjamin/Cummings Publ. Comp., 1989

[8] C. Kalus, P. Dadam, "Incorporating record subtyping into a relational data model", Tech. Report Nr. 94-06, University of Ulm, Fac. of Computer Science, 1994

[9] Y.E. Lien, "Multivalued Dependencies with Null Values in Relational Databases", 5th Int. Conf. on Very Large Data Bases, Brazil, 1979, pp. 155 - 168

[10] D. MacQueen, "Using Dependent Types to Express Modular Structure", 13th ACM Symp. on Principles of Programming Languages, 1986, pp. 277 - 286

[11] D. Maier, "The Theory of Relational Databases", Computer Science Press, 1983

[12] J. Paredaens, P. DeBra, M.Gyssens, D. VanGucht, "The Structure of the Relational Database Model", Springer-Verlag, 1989

[13] E. Sciore, "The Universal Instance and Database Design", Doctoral diss., Princeton Univ., , Oct. 1980

[14] M.H. Scholl, H.-J. Schek, "A Relational Object Model", Third Int. Conf. on Database Theory, LNCS 470, Paris, Dec. 1990, pp. 89 - 105

[15] G.M. Shaw, S.B. Zdonik, "A Query Algebra for Object-Oriented Databases", 6th Int. Conf. on Data Engineering, Los Angeles, Feb. 1990, pp. 154 - 162

[16] J.D. Ullman, "Principles of Database and Knowledge-Base Systems, Volume 1", Computer Sci. Press, 1988

[17] Y. Vassilou, "Functional Dependencies and Incomplete Information", 6th Int. Conf. on Very Large Data Bases, Montreal, Canada, 1980, pp. 260 - 269

Appendix

Proof of completeness of the axiom sytem $\mathcal{AF}$.

Let *AF* be a set of ADs and FDs and let AF^+/AF^- be the set of all dependencies that can/cannot be derived from *AF* by the axioms in $\mathcal{AF}$. X^+_{func}, the closure of FDs for an attribute set X is known from literature (see e.g. [16]). Let X^+_{attr} be the corresponding closure for ADs. The relationship between both is $X^+_{attr} \supseteq X^+_{func}$ as any FD implies an AD due to the subsumption rule (AF1).

To prove completeness, for each $X \longrightarrow Y \in AF^-$ we have to construct a flexible relation FR that satisfies all dependencies in AF^+, but not $X \longrightarrow Y$. Analogous to the proof for FDs we construct a two-tuple (flexible) relation with the following specification (independent on if we regard $X \xrightarrow{attr} Y \in AF^-$ or $X \xrightarrow{func} Y \in AF^-$)

$attr(t_1) = \mathcal{U}$
$\forall A \in \mathcal{U} : t_1(A) = 1$
$attr(t_2) = X^+_{attr}$
$\forall A \in X^+_{func} : t_2(A) = 1$
$\forall A \in X^+_{attr} - X^+_{func} : t_2(A) = 0$

This relation can be visualized as follows (with //// symbolizing non-existent attributes)

$$\begin{array}{ccc} \text{attributes of } X^+_{func} & \text{attributes of } X^+_{attr} - X^+_{func} & \text{attributes of } \mathcal{U} - X^+_{attr} \\ \overbrace{1\,1 \ldots 1} & \overbrace{1\,1 \ldots 1} & \overbrace{1\,1 \ldots 1} \\ 1\,1 \ldots 1 & 0\,0 \ldots 0 & /////// \end{array}$$

Suppose we have to show that $X \xrightarrow{attr} Y$ is not satisfied by this flexible relation. By reflexivity, $X \subseteq X^+_{func}$, so by construction $t_1[X] = t_2[X]$. Y cannot be a subset of X^+_{attr}, otherwise it would have been inferred by the closure property. So by construction $attr(t_1) \cap Y \neq attr(t_2) \cap Y$, i.e. $X \xrightarrow{attr} Y$ is not satisfied.

Suppose at the other hand that we have to show that $X \xrightarrow{func} Y$ is not satisfied by this relation. By the closure property Y cannot be a subset of X^+_{func}, so by construction either $Y \not\subseteq attr(t_2)$ or at least $t_1[Y] \neq t_2[Y]$. In both cases $X \xrightarrow{func} Y$ is not satisfied.

In addition we have to show that FR is a legal relation, i.e. that all dependencies in AF^+ are satisfied. Let $W \xrightarrow{func} Z \in AF^+$. If $W \not\subseteq X^+_{func}$, then t_1 and t_2 disagree on W, and the dependency is trivially satisfied by FR. Let on the other hand $W \subseteq X^+_{func}$. Then by the closure property $X \xrightarrow{func} W$ and by transitivity $X \xrightarrow{func} Z$. Using the closure property again we get $Z \subseteq X^+_{func}$ and now, by construction, $t_1[Z] = t_2[Z]$. Hence $W \xrightarrow{func} Z$ is satisfied by FR.

Take now $W \xrightarrow{attr} Z \in AF^+$. Again, if $W \not\subseteq X^+_{func}$, then the dependency is trivially satisfied by FR. Assume on the other hand $W \subseteq X^+_{func}$. From the functional closure property we can infer that $X \xrightarrow{func} W$. Now the combined transitivity rule applies and yields that $X \xrightarrow{attr} Z$ holds. The attribute closure property asserts that $Z \subseteq X^+_{attr}$ and, by construction, $attr(t_1) \cap Z = attr(t_2) \cap Z$. Hence $W \xrightarrow{attr} Z$ is satisfied by FR. That is, the axioms are complete.