Reducing the Search Space for Conceptual Schema Transformation

P. van Bommel [*] Th.P. van der Weide[†]

July 7, 1993

Published as: P. van Bommel and Th.P. van der Weide. Reducing the Search Space for Conceptual Schema Transformation. *Data & Knowledge Engineering*, 4(8):269–292, 1992.

Abstract

In this paper we focus on the transformation of a conceptual schema into an internal schema. For a given conceptual schema, quite a number of internal schemata can be derived. This number can be reduced by imposing restrictions on internal schemata.

We present a transformation algorithm that can generate internal schemata of several types (including the relational model and the NF^2 model). Guidance parameters are used to impose further restrictions.

We harmonise the different types of schemata by extending the conceptual language, such that both the conceptual and the internal models can be represented within the same language.

Keywords: Conceptual schema, internal schema, schema transformation, relational data model, NF^2 data model.

1 Introduction

The importance of conceptual modelling has been generally recognised. The advantage is that it gives the designer the opportunity to separate the concern of constructing a correct model from that of finding an efficient implementation. Once the conceptual model has been specified in a suitable language, an efficient

[*]Dept. of Information Systems, Faculty of Mathematics and Informatics, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, E-mail: pvb@cs.kun.nl

[†]This work has been partially supported by the ESPRIT project APPED (2499).

realisation can be (automatically) derived, using a specification language that is more machine oriented.

Several conceptual modelling techniques have an underlying object-role structure (e.g. ER [5] and NIAM [11], [15]). In these techniques a model consists of fact types, defined in terms of object types and roles. Depending on the power of the technique, fact types may be of any degree and can be treated as object types (objectified fact types). Furthermore, a distinction can be made between lexical and non-lexical object types and ISA (or subtype) relationships can be defined.

For a given conceptual model the number of correct internal models may be very large, depending on the complexity of the conceptual model and the language used for the internal model. Some internal models will result in an efficient system, others will not. The problem then is how to find a good candidate. The question how such a good candidate can be found will not be addressed in this paper. Our aim is to describe a mechanism for the reduction of the search space, setting a context for more sophisticated search algorithms. Furthermore, this reduction provides a base for structural translations in automated prototyping (see e.g. [12]). As design tools are becoming more important, the need for automatic translation becomes more pressing.

Various implementation oriented modelling techniques exist. A classical distinction can be made between relational, network and hierarchical models. Recently a lot of research has been done on nested relational models, also called non first normal form or NF^2 models (see [1], [6] and [19]). These models are interesting for special database applications, involving e.g. textual data, computer aided design or image processing.

Current approaches to the transformation of conceptual models into internal models focus mainly on the relational model. The result of the transformation is a relational schema in a certain normal form (see for instance [11], [13], [17], [21], [22] and [24]). Other approaches can be found in [2], [10], [20] and [25].

In this paper we present a framework for the transformation of object-role models. We describe a representation mechanism for internal structures, such that the conventional internal models can be represented. The advantage of this approach is, that we can describe conceptual-internal mappings within the scope of a single specification language. For example, the mapping of an internal structure into a Codasyl model can be done by a simple syntactic transformation. Furthermore, this approach gives the opportunity to describe intermediate results easily.

Then, the attention will be focussed on the question how this transformation process can be guided, in order to generate structures having certain predefined characteristics, involving e.g. redundancy, optionals and size of the generated structure.

The organisation of the paper is as follows. In section 2, we discuss a general object-role platform, called the Predicator Model. The central notion of this platform is the *predicator*, the combination of an object type and a role.

In section 3 we show how the NF^2 data structure can be recognised as a tree consisting of nodes of predicators. In section 4 we show how alternative tree representations can be generated for a given information structure, and how guidance parameters can be used to reduce the set of generated representations. In section 5 the generation algorithm is exemplified, starting from a very simple object-role information structure and tracing the generation process. Examples also show how populations (or instantiations) of the information structure correspond to populations of the generated structures. In section 6 we shortly address simplification during postprocessing.

With respect to the graphical representation of information structures, we make use of the drawing conventions of NIAM. Our approach however, is applicable in any model with an underlying object-role structure.

2 The Predicator Model

In the Predicator Model a schema $\Sigma = \langle \mathcal{I}, \mathcal{C} \rangle$ consists of an information structure $\mathcal{I}$ and a set of constraints $\mathcal{C}$ (see [3], [8] and [27]). The semantics of a schema is expressed in terms of populations (instantiations) of the information structure. Such a population should fit in the structure $\mathcal{I}$ and satisfy the requirements specified in $\mathcal{C}$.

With respect to the separation of the information structure $\mathcal{I}$ and the constraints $\mathcal{C}$, note that constraints can also be seen as structural rules. However, constraints are usually expressed in a language that is derived from the components in $\mathcal{I}$ (see for example [3]).

We will first introduce the information structure, and then define the concept of population. Two special types of constraints, the uniqueness constraint and the total role constraint, will be discussed and illustrated using example populations.

2.1 The information structure

Definition 2.1

Information Structure
An *information structure* $\mathcal{I} = \langle \mathcal{P}, \mathcal{O}, \mathcal{F}, \mathsf{Base}, \mathsf{Sub}, \sqcap \rangle$, is a structure consisting of the following basic components:

- $\mathcal{P}$ is a set of *predicators*. A predicator is intended to model a connection between an object type and a role in a fact type. The associated object type is found by the operator $\mathsf{Base} : \mathcal{P} \rightarrow \mathcal{O}$.
- $\mathcal{O}$ is a set of *object types*.
- $\mathcal{F}$ is a partition of the set $\mathcal{P}$ of predicators. The elements of $\mathcal{F}$ are called *fact types*. Fact types are regarded as objects types: $\mathcal{F} \subseteq \mathcal{O}$. They are also referred to as *composed object types*.

The object types in $\mathcal{A} = \mathcal{O} - \mathcal{F}$ are called *atomic object types*. There are two different sorts of atomic object types: entity types ($\mathcal{E}$) and label types ($\mathcal{L}$). Note that in this model a composed object type can not correspond to an entity type.

- Sub is a partial order for atomic object types, with the convention that $a \,\mathsf{Sub}\, b$ is interpreted as: a is a subtype of b. Note that the name *atomic* only refers to being undividable in the sense of not consisting of predicators (as fact types are).

  Each element of $\mathcal{A}$ has associated a (unique) top element, its pater familias. It is found by the function $\sqcap : \mathcal{A} \rightarrow \mathcal{A}$ (which is similar to the top operator from lattice theory). This function satisfies:

  1. $a \,\mathsf{Sub}\, b \Rightarrow \sqcap(a) = \sqcap(b)$
  2. $a \neq \sqcap(a) \Rightarrow a \,\mathsf{Sub}\, \sqcap(a)$

We call predicators p and q *attached* to each other ($p \sim q$), when $\sqcap(\mathsf{Base}(p)) = \sqcap(\mathsf{Base}(q))$. The fact type that corresponds with a predicator is obtained by the operator:

$$\mathsf{Fact} : \mathcal{P} \rightarrow \mathcal{F}$$

It is defined by: $\mathsf{Fact}(p) = f \Leftrightarrow p \in f$.

A fact type is called objectified, if it occurs as the base of a predicator. A predicator is called an objectification, if its base is a fact type. The set $\mathcal{H}$ contains all objectifications:

$$\mathcal{H} = \{\, p \in \mathcal{P} \mid \mathsf{Base}(p) \in \mathcal{F} \,\}$$

Example 2.1 *In figure 1 we see an example of a simple information structure. In this structure we have:*

$$\begin{aligned} \mathcal{P} &= \{p, q, r, s\} \\ \mathcal{O} &= \{A, B, C, f, g\} \\ \mathcal{F} &= \{f, g\} \end{aligned}$$

The fact types are defined as follows: $f = \{p, q\}$, $g = \{r, s\}$. *With respect to the predicators we have:*

$$\begin{array}{lcllcl} \mathsf{Base}(p) & = & A & \mathsf{Fact}(p) & = & f \\ \mathsf{Base}(q) & = & B & \mathsf{Fact}(q) & = & f \\ \mathsf{Base}(r) & = & B & \mathsf{Fact}(r) & = & g \\ \mathsf{Base}(s) & = & C & \mathsf{Fact}(s) & = & g \end{array}$$

The black dots and double-headed arrows represent so-called constraints. Before these constraints are introduced, we first define the concept of population.

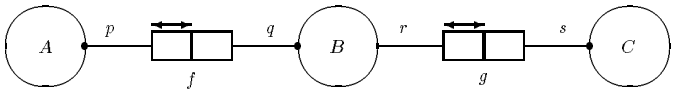

Figure 1: An example information structure

2.2 Population

For the population of an information structure we assume some universal domain (see [14]).

Definition 2.2

Population
A *population* of an information structure $\mathcal{I} = \langle \mathcal{P}, \mathcal{O}, \mathcal{F}, \mathsf{Base}, \mathsf{Sub}, \sqcap \rangle$ (also called an instantiation or state) assigns to each object type in $\mathcal{O}$ a subset of the universal domain, conforming to the structure as prescribed in $\mathcal{P}$ and $\mathcal{F}$, respecting the subtype hierarchy Sub. In case of an atomic object type, this subset should comprise only elementary values of the universal domain. The population of a composed object type is a set of composed values (or tuples). A tuple t of a fact type f is a mapping of all its predicators to values of the appropriate type.

Example 2.2 $\{(p, a_1), (q, b_1)\}$ *is an example of a tuple of fact type* f *in figure 1.*

When no confusion is likely to occur, a tuple will be denoted simply as a set of values. In the example above this results in $\{a_1, b_1\}$.

Example 2.3 *In figure 2 we see an example population for the information structure shown in figure 1.*

f	
A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

g	
B	C
b_2	c_1
b_3	c_1

Figure 2: Example population

The set of possible populations of an information structure defines the semantics of that structure. Unwanted populations can be excluded from this set by static constraints. Changes of population (state transitions) can be forbidden by so-called dynamic constraints. In this paper we only use two types of static constraints: the *uniqueness* constraint and the *total role* constraint.

A uniqueness constraint expresses a functional dependency and is denoted as a double-headed arrow. A total role constraint expresses a mandatory role for a certain predicator: every instance of a certain object type (the base of the predicator) must play that role at least once. The notation used for this type of constraint is a black dot.

It is also possible to define uniqueness constraints over more than one fact type, or total role constraints over more than one predicator (with the same base). For a formal definition of syntax and semantics of these (and other) constraints, we refer to [3] and [27].

Example 2.4 *Consider the information structure in figure 1. We see a uniqueness constraint* unique($\{p\}$). *The population of fact type f given in figure 2 satisfies this constraint, since every instance of object type A occurs at most once in a tuple of f. This property does not hold for predicator q, as b_1 occurs twice.*

The constraint total($\{q\}$) *is also satisfied, since every instance of object type B occurs at least once in a tuple of fact type f. Note the absence of* total($\{r\}$).

Example 2.5 *In figure 3 we see objectified fact type f. This fact type is the base of predicator r. Note that this structure has the same definition as the structure in figure 1 (even the constraints are identical), except for the definition of* Base(r). *In this situation the constraint* unique(r) *means that each instance of f can be associated at most once with an instance of C.*

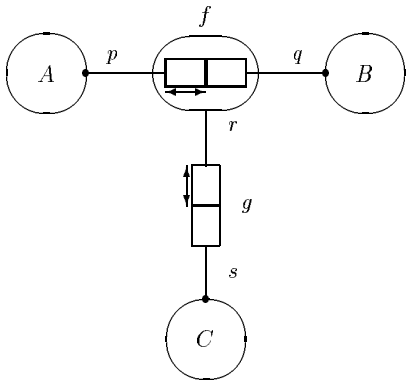

Figure 3: A simple information structure with objectification

For more examples of objectification and the meaning of constraints over objectification, see [3], [15] and [27].

2.3 Defoliation

In section 2.1 two kinds of atomic object types were distinguished: entity types and label types. The difference is that labels can, in contrast with entities, be

reproduced on a communication medium. Therefore it should be possible to uniquely identify an entity type via one or more label types. This unique identification is also called a *naming reference*. Usually, identification is guaranteed using uniqueness and total role constraints.

A naming reference very often consists of one single label type, specified via a so-called bridge type. A fact type f is called a bridge type, only if it has the form $f = \{p, q\}$ with $\mathsf{Base}(p) \in \mathcal{L}$ and $\mathsf{Base}(q) \in \mathcal{E}$.

In complex cases, identification is defined by more than one bridge type, or by a combination of bridge types and fact types.

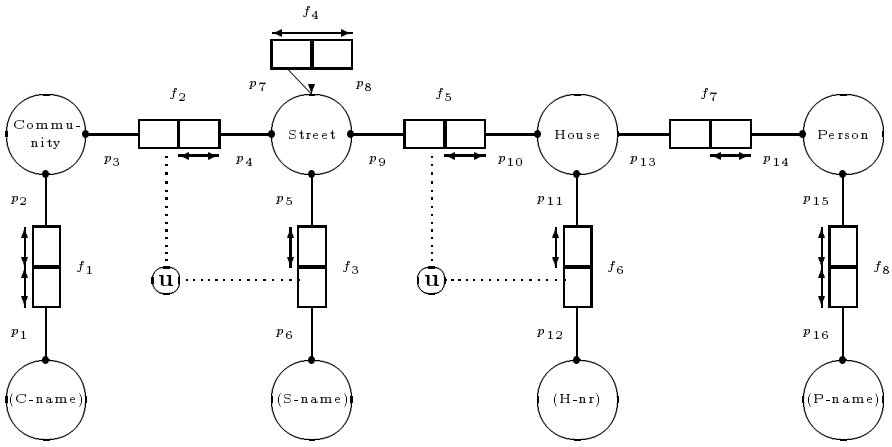

Figure 4: Schema with complex identification

Example 2.6 *Consider the schema in figure 4. In this schema persons live in a house (f_7), houses are located in a street (f_5), and streets cross other streets (f_4) and are located in a community (f_2). By convention a label type is denoted in parenthesis.*

There are two simple naming references: f_1 and f_8. The other naming references are complex. A street can be identified by the unique combination of its community (f_2) and its name (f_3). This combination is unique as a result of the constraints $\mathsf{unique}\,\{p_4\}$, $\mathsf{unique}\,\{p_5\}$ *and* $\mathsf{unique}\,\{p_3, p_6\}$. *A house can be identified by the unique combination of its street (f_5) and its number (f_6). These naming references fulfil the requirements for structural identification (see [3]).*

In internal models the distinction between label types and entity types is not made. Therefore, an information structure must be *defoliated* before it is transformed into an internal structure.

The term "defoliation" is based on the Object Relation Network view of an information structure (see [3]). The effect of defoliation is the restriction of the

information structure to the fact types that are not bridge types:

$$\mathcal{F}_d(\mathcal{I}) = \{\, f \in \mathcal{F} \mid \forall_{p \in f}\, [\mathsf{Base}(p) \notin \mathcal{L}] \,\}$$

Example 2.7 *Consider the schema in figure 4 again. For this schema we have* $\mathcal{F}_d = \{f_2, f_4, f_5, f_7\}$. *The defoliated information structure is presented in Figure 5.*

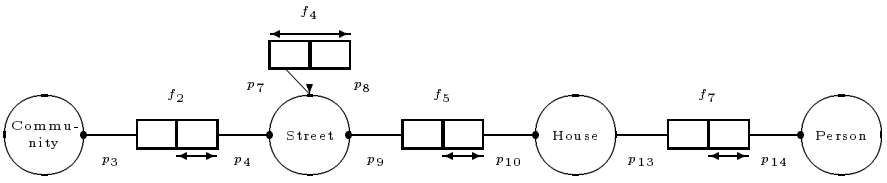

Figure 5: The defoliated information structure

The predicators in a defoliated information structure are specified by $\mathcal{P}_d = \cup\mathcal{F}_d$.

3 Embedding internal structures

In this section data structures that are used in internal models are embedded within the Predicator Model. We first introduce the nested relational model (also called non first normal form or NF^2) as the underlying data structure. Information structures from the examples in the previous section will be represented in alternative NF^2 relation types.

Then, tree structures consisting of nodes of predicators are defined. These trees represent information structures in a hierarchical way, such that NF^2 structures are easily recognised. They form the basis of the generation algorithm described in section 4.

3.1 The underlying data structure

An NF^2 model consists of relation types, where a relation type consists of atomic attribute types and other relation types, called relation-valued attribute types (see [6] and [19]). NF^2 relation types may be in one of the classical relational normal forms (see e.g. [26]), but they do not necessarily have to. In this way the relational theory can be exploited in a wider area.

Recent research involves the definition of a recursive relational algebra (see [6], [18] and [19]) and the introduction of nested normal forms (see e.g. [7] and [16]). Furthermore, the relationship with other modelling approaches is investigated.

Several representations have been introduced for NF^2 relation types, for instance a linear form, a tree representation and a tabular representation (see [19]). In this section we give some examples of the linear form and the tabular representation. In section 3.2 we give a formal definition of tree structures, consisting of nodes of predicators.

In the linear form, a relation type can be denoted as a regular expression, in the so-called nested bracket notation. Then it is an enumeration of the attribute types it contains, separated by brackets.

Example 3.1 *We may for instance have the following relation type:*

$[A, [B, C]\,\mathsf{rep}]$

This relation type consists of two attribute types, an atomic attribute type A and a relation-valued attribute type $[B, C]\,\mathsf{rep}$. The relation-valued attribute type consists of two atomic attribute types B and C.

In this structure an A value can be related to one or more B, C combinations. The keyword rep expresses *repeating*.

If each A value occurs only once, denoted as $\overline{A}$, the relation type is said to be in *partitioned normal form* (PNF). With this basic normal form it can be guaranteed that *nest* is always the inverse operator for *unnest*, an important property for e.g. query optimization. For more details see [18]).

In this paper we restrict ourselves to relation types in PNF. As a further refinement, we use the keyword op to denote an *optional* attribute type.

Example 3.2
The schema in figure 1 can be represented by the following alternative relation types:

1. $[\,\overline{A}\,, B, C\;\mathsf{op}]$
2. $[\,\overline{B}\,, A\;\mathsf{rep}, C\;\mathsf{op}]$
3. $[\,\overline{C}\,, [\,\overline{B}\,, A\;\mathsf{rep}]\;\mathsf{rep}]$

Each relation type corresponds to a hierarchical view of the information structure, with either A, B or C as root. The root of a (sub)tree is a key of the corresponding relation type. The formal definition of this hierarchical view is given in section 3.2.

Consider the first alternative. We will show how this alternative is obtained by taking object type A as root. Direct translation of the structure in figure 1 into a regular expression results in $[\,\overline{A}\,, [\,\overline{B}\,, [\,\overline{C}\,]\;\mathsf{op}]]$. As a notational simplification, unary groups are usually denoted without surrounding brackets. Structural simplifications are also possible. For example, $[\,\overline{A}\,, [\,\overline{B}\,, C]]$ is equivalent to $[\,\overline{A}\,, B, C]$, as the subgroup $[\,\overline{B}\,, C]$ is neither optional nor repeating.

Note the effect of the constraints, specified in figure 1, on the candidate relation types. For instance, the absence of constraint $\mathsf{total}(r)$ caused an optional attribute type $C\,\mathsf{op}$ in the first and second alternative. The absence of constraint $\mathsf{unique}(q)$ caused a repeating (relation-valued) attribute type $A\,\mathsf{rep}$ in the second and third alternative.

The second alternative mentioned above is illustrated at the instance level in figure 6.

$\overline{B}$	$A\,\mathsf{rep}$	$C\,\mathsf{op}$
b_1	a_1 a_2	ε
b_2	a_3	c_1
b_3	a_4	c_1

Figure 6: Example population in second alternative

The population in figure 6 corresponds to the example population of the original information structure (see figure 2). Note that we have used ε to denote the empty value.

Example 3.3
The schema in figure 3 can be represented by the following relation types:

1. $[\,\overline{A}\,, B, C\,\mathsf{op}]$
2. $[\,\overline{B}\,, [\,\overline{A}\,, C\,\mathsf{op}]\,\mathsf{rep}]$

For very simple information structures it is sufficient to use object types for the denotation of the resulting NF^2 attribute types. However, to be able to represent information structures of arbitrary complexity, a predicator based approach should be used. Instead of using the associated object types, attribute types then are denoted as sets of predicators.

Example 3.4
For the alternatives given in example 3.2 this leads to:

1. $\left[\,\overline{\{p\}}\,, \{q, r\}\,, \{s\}\ \mathsf{op}\right]$
2. $\left[\,\overline{\{q, r\}}\,, \{p\}\ \mathsf{rep}, \{s\}\ \mathsf{op}\right]$
3. $\left[\,\overline{\{s\}}\,, \left[\,\overline{\{q, r\}}\,, \{p\}\ \mathsf{rep}\right]\mathsf{rep}\right]$

3.2 Tree representations

In this section we extend the Predicator Model by a representation mechanism corresponding to an internal structure, in the form of a hierarchical relation type, as we encountered in the previous section. The hierarchical nature is represented in a tree, where each node is a set of predicators.

Definition 3.1

Tree Representation
A forest (set of trees) $\mathcal{T} = \langle \mathcal{N}, E, \ell \rangle$ is a *complete tree representation* of information structure $\mathcal{I} = \langle \mathcal{P}, \mathcal{O}, \mathcal{F}, \mathsf{Base}, \mathsf{Sub}, \sqcap \rangle$, iff it satisfies the following conditions:

t_1 : A node is a set of predicators. The set of nodes $\mathcal{N}$ forms a partition of $\mathcal{P}_d(\mathcal{I})$ (see section 2.3). All predicators in a node are mutually attached to each other. As a result, all object types that are involved in a node belong to the same subtype hierarchy. The associated pater familias is denoted as the base of the node: $\mathsf{Base}(n)$ for node n.

t_2 : No two predicators of the same fact type may participate in the same node:

$$\forall_{n \in \mathcal{N}, f \in \mathcal{F}} [|n \cap f| \leq 1]$$

t_3 : Objectifications (elements of $\mathcal{H}$) are treated separately:

$$\forall_{n \in \mathcal{N}} [|n \cap \mathcal{H}| \leq 1]$$

t_4 : E is a set of edges. The pair $\langle m, n \rangle$ represents an edge from node m to node n. Edges are labelled by fact types in the obvious way, with some care to handle objectification. The function $\ell : E \rightarrow \mathcal{F}$ assigns $\ell(\langle m, n \rangle) = f$, iff:

1. $n \cap f \neq \emptyset$
2. $m \cap f = \emptyset \Rightarrow \mathsf{Base}(m) = f$

t_5 : Fact types are located around a single father:

$$\ell(\langle m_1, n_1 \rangle) = \ell(\langle m_2, n_2 \rangle) \;\Rightarrow\; n_1 = n_2$$

t_6 : In order to guarantee that the complete information structure is represented, each predicator must be involved in some edge:

$$\forall_{p \in \mathcal{P}_d} \exists_{\langle m,n \rangle \in E} [p \in m \cup n]$$

We call a tree representation incomplete if it does not necessarily fulfil the last requirement.

Example 3.5
The tree in figure 7 is a possible tree representation for the information structure of figure 1, according to:

$$\begin{aligned}
\mathcal{N} &= \{n_1, n_2, n_3\} \text{ with:} \\
&\quad n_1 = \{q, r\}, n_2 = \{p\} \text{ and } n_3 = \{s\} \\
E &= \{e_1, e_2\} \text{ with:} \\
&\quad e_1 = \langle n_2, n_1 \rangle \text{ and } e_2 = \langle n_3, n_1 \rangle \\
\ell(e_1) &= f \\
\ell(e_2) &= g
\end{aligned}$$

This tree representation corresponds to relation type $\left[\overline{\{q,r\}}, \{p\}\ \mathsf{rep}, \{s\}\ \mathsf{op}\right]$, *which is the second alternative of example 3.4.*

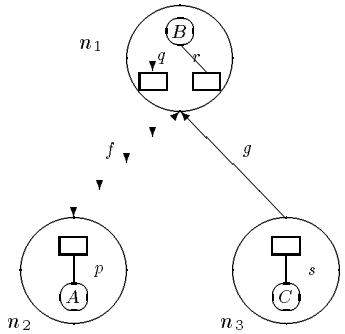

Figure 7: Simple tree

Example 3.6
The tree in figure 8 is a possible tree representation for the information structure of figure 3, according to:

$$\begin{aligned}
\mathcal{N} &= \{n_1, n_2, n_3, n_4\} \text{ with:} \\
&\quad n_1 = \{q\}, n_2 = \{p\}, n_3 = \{r\} \text{ and} \\
&\quad n_4 = \{s\} \\
E &= \{e_1, e_2, e_3\} \text{ with:} \\
&\quad e_1 = \langle n_2, n_1 \rangle, e_2 = \langle n_3, n_1 \rangle \text{ and} \\
&\quad e_3 = \langle n_4, n_3 \rangle \\
\ell(e_1) &= f \\
\ell(e_2) &= f \\
\ell(e_3) &= g
\end{aligned}$$

This tree representation corresponds to relation type $\left[\overline{\{q\}}, \left[\overline{\{p\}}, \{s\}\ \mathsf{op}\right] \mathsf{rep}\right]$, *which is the second alternative of example 3.3.*

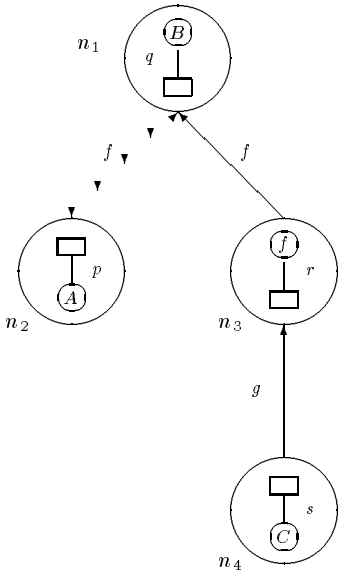

Figure 8: Example tree with objectification

Note the duality of both partitions $\mathcal{F}$ and $\mathcal{N}$ of the predicators. The first is based on equality of fact type, the second is related to attachedness. The unique node in which a predicator is contained can be found by the function:

$$\mathsf{Node} : \mathcal{P}_d \rightarrow \mathcal{N}$$

It is defined by: $\mathsf{Node}(p) = n \Leftrightarrow p \in n$, analogous with the function Fact (see section 2).

A node m, being the source of an edge, is anchored to this edge by a unique predicator $p \in m$:

Lemma 3.1 *Let $e = \langle m, n\rangle$ be an edge and let $\ell(e) = f$. If node m has an atomic base, it has the following property:*

$$|m \cap f| = 1$$

Proof: Let $\langle m, n\rangle \in E$ with $\mathsf{Base}(m) \in \mathcal{A}$ and $\ell(\langle m, n\rangle) = f$. From condition t_4 we conclude that $m \cap f \neq \emptyset$. Now the result can be obtained from condition t_2.

□

For node m this unique predicator is denoted as $\mathsf{Anchor}(m)$. During the construction of tree representations this anchor will play an important role in the guidance of the generation process.

The application of lemma 3.1 in the example tree representations is trivial:

Example 3.7
In the tree shown in figure 7 we have $\mathsf{Anchor}(n_2) = p$ *and* $\mathsf{Anchor}(n_3) = s$. *For* n_1 *it is not defined, since* n_1 *is not the source of any edge.*

Example 3.8
In the tree shown in figure 8 we have $\mathsf{Anchor}(n_2) = p$ *and* $\mathsf{Anchor}(n_4) = s$. *For the root* n_1 *and the objectification* n_3 *it is not defined.*

For the nodes, being the destination of some edge, an analogous property holds:

Lemma 3.2 *Edge* $e = \langle m, n \rangle$ *with* $\ell(e) = f$ *has the following property:*

$$|n \cap f| = 1$$

Proof: Let $\langle m, n \rangle \in E$ with $\ell(\langle m, n \rangle) = f$. From condition t_4 we conclude that $n \cap f \neq \emptyset$. Now the result can be obtained from condition t_2.

□

$\mathcal{R} \subseteq \mathcal{N}$ is the set of nodes being the root of some tree:

$$\mathcal{R} = \{\, x \in \mathcal{N} \mid \forall_{\langle m,n \rangle \in E} [x \neq m] \,\}$$

A basic property of a forest $\mathcal{T}$ of information structure $\mathcal{I}$ involves the number of nodes in $\mathcal{T}$. It can be expressed in terms of the number of predicators, fact types and objectifications in $\mathcal{I}$ and the number of trees in $\mathcal{T}$:

Lemma 3.3 $|\mathcal{N}| = |\mathcal{P}_d| - |\mathcal{F}_d| + |\mathcal{H}| + |\mathcal{R}|$

Proof: Suppose $e_1, \ldots, e_k$ are all edges labelled with the same fact type f. Then all these edges have the same destination node (condition t_5). This destination node contains precisely one predicator of f (see lemma 3.2). From lemma 3.1 we conclude that the sources of the edges either contain precisely one predicator of f, or correspond to an objectification of f.

We denote the number of objectifications of f as $\|f\|$. We then have:

$$\begin{aligned} &\text{number of edges labelled with } f \\ = \;& |f| - 1 + \|f\| \end{aligned}$$

Summation of all fact types yields:

$$\begin{aligned} |E| &= \sum_{f \in \mathcal{F}_d} \text{number of edges labelled with } f \\ &= \sum_{f \in \mathcal{F}_d} (|f| - 1 + \|f\|) \\ &= |\mathcal{P}_d| - |\mathcal{F}_d| + |\mathcal{H}| \end{aligned}$$

The number of nodes equals the number of edges plus the number of trees. As a result:

$$|\mathcal{N}| = |E| + |\mathcal{R}| = |\mathcal{P}_d| - |\mathcal{F}_d| + |\mathcal{H}| + |\mathcal{R}|$$

□

4 Generation of tree representations

4.1 The generation process

The intention of the generation process that we describe is to generate all tree representations for a given information structure $\mathcal{I}$. However, in order to avoid the overhead of the backtrack mechanism, we let the algorithm search for a random tree structure.

The construction first sets up an initial incomplete tree representation. This incomplete representation is transformed step by step until it is complete. In each step a fact type will be incorporated in the tree representation. As a consequence, the number of steps equals the number of fact types in the defoliated information structure.

Initially each predicator constitutes its own isolated node, not linked to any other node. This initialisation establishes the following precondition:

Lemma 4.1 Precondition

$\mathcal{I}$ is an information structure

$\mathcal{N} := \{\ \{p\} \mid p \in \mathcal{P}_d(\mathcal{I})\ \}$;
$E := \emptyset$;
$\ell := \emptyset$;
$\mathcal{T} := \langle \mathcal{N}, E, \ell \rangle$;

Forest($\mathcal{T}, \mathcal{I}$)

Proof: Let $\mathcal{I}$ be an information structure. After the initialisation, $\langle \mathcal{N}, E, \ell \rangle$ satisfies the conditions t_1, t_2 and t_3, because each node contains exactly

one predicator. The conditions t_4 and t_5 are void, because there are no edges. From this we conclude that $\mathcal{T}$ is an incomplete tree representation of $\mathcal{I}$, denoted as $\mathsf{Forest}(\mathcal{T},\mathcal{I})$.

□

In each step of the algorithm a fact type should be selected to be incorporated in the tree representation. This is done by selecting a predicator of a yet unprocessed fact type. This predicator then will act as a handle for the extension of the tree representation.

We use the set $\mathcal{U}$ to record the unprocessed isolated nodes. The procedure CanExtend selects an unprocessed node $m \in \mathcal{U}$, in combination with another node $n \in \mathcal{N}$. The actual extension is performed by the procedure ProcessFactType. The algorithm:

proc GenerateForest ($\mathcal{I}$: Information Structure):Forest;
  $\mathcal{N} := \{\ \{p\} \mid p \in \mathcal{P}_d(\mathcal{I})\ \}$;
  $E := \emptyset$;
  $\ell := \emptyset$;
  $\mathcal{U} := \mathcal{N}$;
  while CanExtend (m, n) **do**
    ProcessFactType (m, n)
  od ;
  $\mathcal{T} := \langle \mathcal{N}, E, \ell \rangle$
endproc GenerateForest

The extension of a tree representation either enlarges an already existing tree, or adds a new tree to the forest. If the unprocessed node $m \in \mathcal{U}$ is chosen in combination with a processed node $n \in \mathcal{N} - \mathcal{U}$, an existing tree is extended. A choice for $n = m$ leads to the creation of a new tree. Other combinations of m and n are not allowed, because they do not result in a correct tree representation. The procedure CanExtend will check this. Furthermore it will guarantee that all object types within the same node belong to the same subtype hierarchy:

$$\sqcap(\mathsf{Base}(m)) = \mathsf{Base}(n)$$

It may be desirable to further restrict the possibilities for extension, e.g. when the result of the generation must fulfil certain design criteria. This further restriction is discussed in the next section.

For the extension of an existing tree and for the creation of a new tree one single strategy can be used. This strategy consists of the processing of the fact type, which is implicitly specified via the unprocessed unary node m.

The extension starts with the union of the selected nodes m and n. Then the entire fact type f, specified via m, is processed by the addition of edges with label f:

proc ProcessFactType $(m, n$: Node);
  Let $m = \{p\}$;
  if $m \neq n$ **then** $\mathcal{N} := \mathcal{N} - \{m\}$;
  $n := m \cup n$;
  for $x \in \mathsf{Fact}(p) - m$ **do**
    $E := E \cup \{\langle \mathsf{Node}(x), n\rangle\}$;
    $\ell(\langle \mathsf{Node}(x), n\rangle) := \mathsf{Fact}(p)$
  od ;
  $\mathcal{U} := \mathcal{U} - \mathsf{Fact}(p)$;
  if $\exists_{\{x\}\in\mathcal{U}} [\mathsf{Base}(x) = \mathsf{Fact}(p)]$ **then** ProcessObjectifications($\mathsf{Fact}(p), n$);
endproc ProcessFactType

Finally, we have to handle the objectifications of the fact type f under consideration. For each fact type g, which contains an objectification of f, we have to choose a predicator for unnesting. This unnesting is performed by the addition of edges with label f. The fact type g then is processed by recursively calling procedure ProcessFactType:

proc ProcessObjectifications (f : FactType; n : Node);
  for $m \in \mathcal{U}$ with $\mathsf{Base}(m) = f$ **do**
    $E := E \cup \{\langle m, n\rangle\}$;
    $\ell(\langle m, n\rangle) := f$;
    ProcessFactType(m, m)
  od
endproc ProcessObjectifications

Next we show that the loop in procedure GenerateForest is governed by the following invariant:

Lemma 4.2 Invariant

Forest$(\mathcal{T}, \mathcal{I})$ **and** CanExtend(m, n)

  ProcessFactType(m, n)

Forest$(\mathcal{T}, \mathcal{I})$

Proof: Suppose $\mathsf{Forest}(\mathcal{T}, \mathcal{I})$ and $\mathsf{CanExtend}(m, n)$. Let f be the fact type implicitly specified by m. After the execution of ProcessFactType, condition t_1 is satisfied as a result of the checks in $\mathsf{CanExtend}(m, n)$.

Each predicator in $f - m$ is the source of a new edge with destination n and label f. From this we conclude that the conditions t_2 and t_5 are satisfied.

Furthermore, the recursive call in the processing of objectifications guarantees the conditions t_3 and t_4. As a consequence, the resulting forest $\mathcal{T}$ is an incomplete tree representation of $\mathcal{I}$.

□

The previous lemmas guarantee the correctness of the entire generation:

Lemma 4.3 Postcondition

$\#$ $\mathcal{I}$ is an information structure $\#$

$\mathcal{T}$:= GenerateForest($\mathcal{I}$)

$\#$ $\mathcal{U} = \emptyset$ $\Rightarrow$ CompleteForest($\mathcal{T}, \mathcal{I}$) $\#$

Proof: Let $\mathcal{I}$ be an information structure. The result of the initialisation is an incomplete tree representation (see lemma 4.1). In each iteration the procedure ProcessFactType yields an incomplete tree representation (see lemma 4.2). If GenerateForest terminates with $\mathcal{U} = \emptyset$ each predicator has been classified, and thus $\mathcal{T}$ is a complete tree representation according to condition t_6.

□

4.2 Further restriction

The generation process described in the previous section results in a correct tree representation, according to the definition in section 3.2. However, in many cases the property of correctness is not strong enough, e.g. when the search space in schema transformation is to be reduced. Then, database structures to be generated must be characterised in advance, in order to exclude candidates with undesirable properties from the generation process.

This characterisation can be embedded within the generation framework as follows. In each iteration the procedure CanExtend selects two nodes m and n as a base for further processing. This selection leaves us many opportunities to influence the nature of the resulting tree representation, simply by restricting the number of possible combinations of m and n. We will discuss some restrictions in this section. In section 5 and 6 the algorithm is illustrated in several detailed examples.

Let $m = \{p\} \in \mathcal{U}$ be an unprocessed isolated node and let n be either a processed node ($n \in \mathcal{N} - \mathcal{U}$) or a new root ($n = m$), such that their bases belong to the same subtype hierarchy. This forms the starting-point for further restriction (see figure 9).

proc CanExtend (**var** m, n : Node) : Boolean;
take random $m = \{p\} \in \mathcal{U}$, $n \in (\mathcal{N} - \mathcal{U}) \cup \{m\}$ with $\sqcap(\mathsf{Base}(m)) = \mathsf{Base}(n)$
such that: $n \in \mathcal{R} \vee \mathsf{total}(\mathsf{Anchor}(n))$
and if NoNesting **then** $\mathsf{unique}(p)$
and if NoRedundancy **then** $n \in \mathcal{R} \vee \mathsf{unique}(\mathsf{Anchor}(n))$
and if NoOptionals **then**
if $n \in \mathcal{R}$
then $\forall_{x \in m \cup n}\,[\mathsf{total}(x)] \vee \forall_{x \in m \cup n}\,[\neg\,\mathsf{total}(x)]$
else $\mathsf{total}(p)$
and if SizePreference **then** $|\mathsf{Facts}(n)| < \gamma$
and if DepthPreference **then** $\mathsf{Depth}(n) < \delta$
endproc CanExtend

Figure 9: Guidance conditions for the extension of the tree representation

First we deal with lossless tree representations. To prevent loss of information in the result, the anchor of node n should have a total role constraint:

$$n \in \mathcal{R} \vee \mathsf{total}(\mathsf{Anchor}(n))$$

Absence of this constraint allows $\mathsf{Fact}(p)$ to have instances that cannot be related to instances of $\mathsf{Fact}(\mathsf{Anchor}(n))$ higher up in the tree. These instances of $\mathsf{Fact}(p)$ can not be reached via node n, which results in loss of information.

Next we deal with several other structural properties, such as absence of nesting, redundancy or optionals, and preferences with respect to the size or depth of the tree. A certain combination of these properties can be specified via the guidance parameters NoNesting, NoRedundancy, etcetera (see figure 9).

The meaning of these parameters is expressed in so-called guidance conditions. For example, for the construction of flat relational structures (parameter NoNesting), a uniqueness constraint $\mathsf{unique}(p)$ is required. Freedom from redundancy (parameter NoRedundancy) is obtained when $\mathsf{Anchor}(n)$ has a uniqueness constraint, provided n is not a root:

$$n \in \mathcal{R} \vee \mathsf{unique}(\mathsf{Anchor}(n))$$

A structure will be free from optionals (parameter NoOptionals) when the equality of $\mathsf{Pop}(m)$ and $\mathsf{Pop}(n)$ can be guaranteed. This is expressed in terms of total role constraints. Furthermore, a maximal number (γ) of fact types per tree can be specified (parameter SizePreference) and a maximal depth δ of the tree can be specified (parameter DepthPreference). In the following sections these conditions will be illustrated in examples.

5 An elaborated example

In this section the behaviour of the generation algorithm is illustrated. This is done by showing how an example information structure is transformed into tree structures, and how different settings of the guidance parameters influence the result.

Consider the information structure shown in figure 10. We assume that this structure has already been defoliated. The population in figure 11 will be used to illustrate the characteristics of some generated trees at the level of instances.

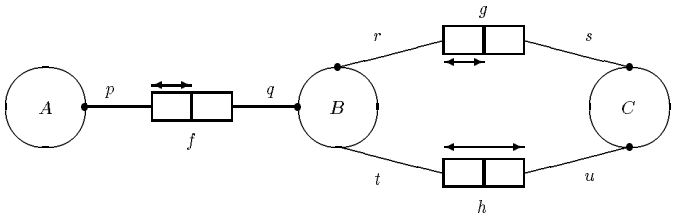

Figure 10: Example schema

f	
A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

g	
B	C
b_1	c_1
b_2	c_1
b_3	c_2

h	
B	C
b_1	c_1
b_2	c_1
b_1	c_2

Figure 11: Example population

The selection of a parameter setting will on the one hand be determined by quantitative aspects of the conceptual schema under consideration. On the other hand it will be determined by aspects of the target environment. For example, if the application is to be stored on CD-ROM, the parameter NoRedundancy can be switched off. In a relational target environment, the parameter NoNesting obviously should be set to true. For more details, see [4].

5.1 Size preference without redundancy

Suppose we aim at a tree structure where the maximal number of fact types per tree is 2 and redundancy can not occur. This can be obtained by the following setting of parameters:

NoRedundancy = True
SizePreference = True
$\gamma = 2$

In the structure in figure 10 we have $\mathcal{P}_d(\mathcal{I}) = \{p, q, r, s, t, u\}$ and therefore both $\mathcal{N}$ and $\mathcal{U}$ initially consist of 6 nodes, each containing one predicator. There are no edges yet: $\mathcal{E} = \emptyset$ and $\ell = \emptyset$.

We will show how the generation proceeds step by step, in each step choosing two nodes m and n that fulfil the guidance criteria, followed by the processing of the fact type implicitly specified by m. During this processing the set of unprocessed nodes $\mathcal{U}$ will be reduced. Furthermore, nodes in $\mathcal{N}$ will be joined and edges between these nodes will be constructed.

1. Let the first selection of nodes consist of $m = \{r\}$ and $n = \{r\}$. Note that the choice $m = n$ will lead to the creation of a new root (see section 4.1). This choice passes all checks in CanExtend. Procedure ProcessFactType reacts as follows: an edge labelled with g from $\{s\}$ to $\{r\}$ is created, and $\{s\}$ and $\{r\}$ are removed from $\mathcal{U}$. $\mathcal{N}$ does not change, because no nodes have to be joined. The resulting situation is shown in figure 12.

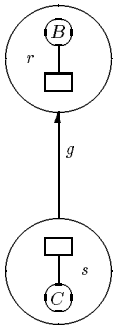

Figure 12: Situation after one step

2. Now $\mathcal{U}$ contains the unprocessed isolated nodes $\{p\}, \{q\}, \{t\}$ and $\{u\}$. Since the maximal grouping rate $\gamma = 2$ has not yet been reached, the next step may either extend the tree in figure 12 or create a new root. Suppose the existing tree is extended. There are three possibilities to do this:

   (a) Choose $m = \{u\}$ and $n = \{s\}$, and process fact type h.

   (b) Choose $m = \{t\}$ and $n = \{r\}$, and process fact type h.

   (c) Choose $m = \{q\}$ and $n = \{r\}$, and process fact type f.

   Consider the first alternative. It attempts to join the nodes $\{u\}$ and $\{s\}$, and to add an edge from node $\{t\}$ to node $\{u, s\}$. However, Anchor($\{u, s\}$) =

s and predicator s is not unique. Therefore, the guidance condition of parameter NoRedundancy is not fulfilled and as a consequence, this possibility is rejected. In section 5.2 the meaning of this guidance condition will be illustrated at the instance level.

Next we consider the second alternative. ProcessFactType reacts as follows: after joining $\{t\}$ and $\{r\}$, an edge labelled with h from $\{u\}$ to $\{r,t\}$ is added. Furthermore, the nodes $\{u\}$ and $\{t\}$ are deleted from $\mathcal{U}$. The resulting tree is shown in figure 13.

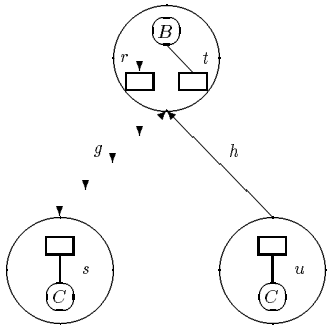

Figure 13: Situation after two steps

In case the third alternative is selected, the result is an edge labelled f from $\{p\}$ to $\{q,r\}$. We do not consider this alternative and proceed with the tree shown in figure 13.

3. At this moment $\mathcal{U}$ contains two unprocessed nodes $\{p\}$ and $\{q\}$. This leaves three possibilities for the processing of fact type f in the last step:

   (a) Join $\{q\}$ and $\{r,t\}$.

   (b) Create a new root $\{q\}$.

   (c) Create a new root $\{p\}$.

   Consider the first alternative. Observing the guidance parameters (especially $\gamma = 2$), we see that this alternative is rejected, as it would result in a tree containing three fact types.

   Next we consider the second alternative. In this alternative q is chosen to become the root of a new tree. In the reaction of ProcessFactType, $\mathcal{N}$ does not change, $\mathcal{U}$ is made empty and an edge from node $\{p\}$ to node $\{q\}$ is added. The label of this new edge is f. This alternative leads to the final situation shown in figure 14.

   If the third alternative is chosen the new root is $\{p\}$. Then an edge from $\{q\}$ to $\{p\}$ is added.

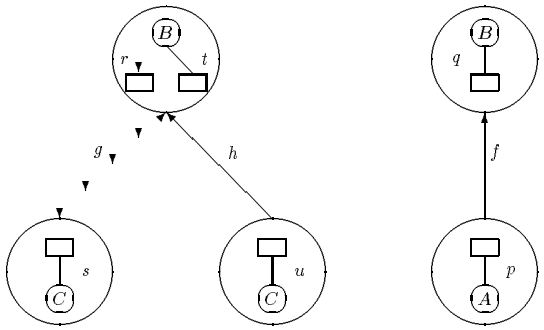

Figure 14: Final situation

The forest in figure 14 corresponds to the following relation types:

- $\left[\overline{\{r,t\}}, \{s\}, \{u\}\ \textsf{op rep}\right]$
- $\left[\overline{\{q\}}, \{p\}\ \textsf{rep}\right]$

There are two relation-valued attribute types $\{u\}$ and $\{p\}$. $\{u\}$ is an optional relation-valued attribute type for instances of C. It is the result of $\neg\,\textsf{unique}(t)$ and $\neg\,\textsf{total}(t)$ (see figure 10).

The example population from figure 11 is represented in the generated structure in figure 15.

$\overline{\{r,t\}}$	$\{s\}$	$\{u\}$ op rep	$\overline{\{q\}}$	$\{p\}$ rep
b_1	c_1	c_1	b_1	a_1
		c_2		a_2
b_2	c_1	c_1	b_2	a_3
b_3	c_2	ε	b_3	a_4

Figure 15: Example population in generated structure

We have used parameter NoRedundancy for the generated structure. Note that this structure is indeed free from redundancy, as every fact instance is mentioned only once. Note furthermore that empty values (denoted by ε) are allowed for attribute type $\{u\}$.

The scenario discussed in this section shows the strong influence of the first step of the generation on the final result. The choice for node $\{r\}$ in the first step (in combination with parameter NoRedundancy) fully determined the height of the tree(s) in the resulting structure.

5.2 Maximal grouping without optionals

In case a maximal grouping of fact types is preferred, such that optional attribute types cannot occur, the parameters should be set as follows:

NoOptionals = True
SizePreference = True
$\gamma = |\mathcal{F}_d| = 3$

The ultimate result of maximal grouping is a forest consisting of a single tree. However, since γ expresses the *maximal* number of fact types per tree, the setting $\gamma = |\mathcal{F}_d|$ will also allow representations consisting of more than one tree.

Initially both $\mathcal{U}$ and $\mathcal{N}$ consist of 6 nodes, each containing one predicator. Furthermore $\mathcal{E} = \emptyset$ and $\ell = \emptyset$.

1. Suppose the first step chooses $m = \{q\}$ and $n = \{q\}$. This choice passes all checks in CanExtend. Then ProcessFactType reacts as follows: an edge labelled with f is constructed from $\{p\}$ to $\{q\}$, and both processed nodes are removed from $\mathcal{U}$. $\mathcal{N}$ does not change. The resulting situation is shown in figure 16.

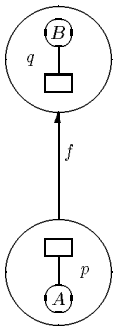

Figure 16: Situation after one step

2. Now $\mathcal{U}$ consists of the nodes $\{r\}, \{s\}, \{t\}$ and $\{u\}$. There are six possibilities for the next step:

   (a) Choose $m = \{t\}$ and $n = \{q\}$, and process fact type h.

   (b) Choose $m = \{r\}$ and $n = \{q\}$, and process fact type g.

   (c) Create a new root for either $\{r\}, \{s\}, \{t\}$ or $\{u\}$.

   Consider the first alternative. It attempts to join node $\{t\}$ and node $\{q\}$, adding an edge from node $\{u\}$ to node $\{q,t\}$. However, the root $\{q,t\}$ causes an optional attribute type $\{u\}$, since predicator q is total and t

is not. Therefore the guidance condition of parameter NoOptionals is not fulfilled and CanExtend rejects this possibility.

Next we consider the second alternative. It selects unprocessed node $\{r\}$ in combination with processed node $\{q\}$. Procedure ProcessFactType will react as follows: after joining $\{r\}$ and $\{q\}$, an edge labelled with g from $\{s\}$ to $\{q,r\}$ is added. Furthermore, the nodes $\{r\}$ and $\{s\}$ are deleted from $\mathcal{U}$. The situation is shown in figure 17.

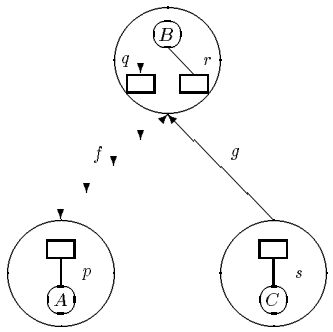

Figure 17: Situation after two steps

In the third alternative a new root is created. This possibility will not be worked out further.

3. At this moment $\mathcal{U}$ contains the unprocessed nodes $\{t\}$ and $\{u\}$. This leaves four possibilities for the processing of fact type h in the last step:

   (a) Choose $m = \{t\}$ and $n = \{q,r\}$.

   (b) Choose $m = \{u\}$ and $n = \{s\}$.

   (c) Create a new root for either $\{t\}$ or $\{u\}$.

   Consider the first alternative. It attempts to join $\{t\}$ and $\{q,r\}$. As a result of parameter NoOptionals this alternative will be rejected (see also the previous step).

   Consider the second alternative. It selects $\{u\}$ and $\{s\}$ to be joined. In the reaction of ProcessFactType the set $\mathcal{N}$ is adapted by joining $\{u\}$ and $\{s\}$, and $\mathcal{U}$ is made empty. Furthermore an edge labelled with h from node $\{t\}$ to node $\{u,s\}$ is added. This leads to the final situation shown in figure 18.

The tree in figure 18 corresponds to the following relation type:

$$\left[\overline{\{q,r\}}\,,\{p\}\ \mathsf{rep},\{s,u\}\,,\{t\}\ \mathsf{rep}\right]$$

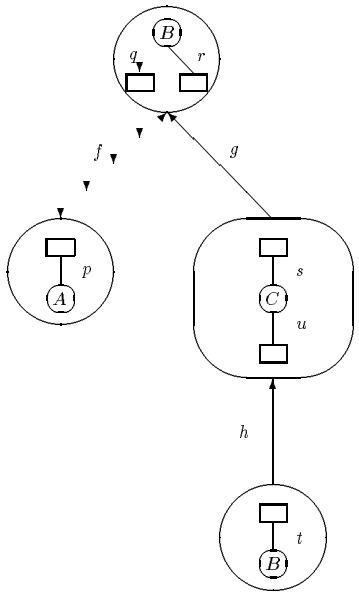

Figure 18: Final situation

There are two relation-valued attribute types $\{p\}$ and $\{t\}$, for instances of A and B respectively. The nesting of these attribute types is the result of $\neg$ unique(q) and $\neg$ unique(u) (see figure 10).

The population from figure 11 is represented in the generated relation type in figure 19. Note that this structure does not have optional attribute types, as was specified by the guidance parameters.

$\overline{\{q,r\}}$	$\{p\}$ rep	$\{s,u\}$	$\{t\}$ rep
b_1	a_1	c_1	b_1
	a_2		b_2
b_2	a_3	c_1	b_1
			b_2
b_3	a_4	c_2	b_1

Figure 19: Example population in generated structure

The tree representation shown in figure 18 is essential for the meaning of figure 19. There are three tuples, the first one of which is read as follows: b_1 *is related to* a_1 *and* a_2 *(via* f*) and to* c_1 *(via* g*).* c_1 *is related to* b_1 *and* b_2 *(via* h*).* An incorrect interpretation could be: a_1 *is related to* b_1*, and* a_2 *is related to* b_2.

We see that the generated structure allows for redundancy: the instances $\{c_1, b_1\}$ and $\{c_1, b_2\}$ of fact type h are mentioned twice. A closer inspection of the original schema in figure 10 and the tree representation in figure 18 leads to the cause of this redundancy. In the tree structure we see that an instance of fact type h containing object x of type C is mentioned as often as x is involved in g. Since predicator s is not unique, some instances of h may be mentioned more than once.

5.3 Loss of information

At the end of the previous section we considered the cause of redundancy. In this section we will show that the point that was made there is closely related to the question of loss of information.

Consider the schema in figure 10 and the tree representation in figure 18 again. Assume the absence of constraint $\mathsf{total}(s)$. This assumption does not affect the relation type expressed by the tree in figure 18:

$$\left[\, \overline{\{q, r\}}\,, \{p\}\,\mathsf{rep}, \{s, u\}\,, \{t\}\,\mathsf{rep} \right]$$

Let c_3 be an object of type C which is not involved in fact type g. If c_3 is involved in an instance of fact type h, a problem arises: this instance of h cannot be represented in a tree structure according to figure 18. This problem does not occur if the anchor of non-root and non-leave nodes is required to be total (see section 4.2).

6 Complex identification

Very often the identification of an entity type can be done by a single label type. However, it may be necessary to use a combination of label types as identifier. In this section we discuss the effect of such complex naming references on the resulting relation types. We take the schema from example 2.6 as input for the generation process.

6.1 Maximal grouping

Maximal grouping is obtained by the following setting of parameters:

SizePreference = True
$\gamma = |\mathcal{F}_d|$

Consider the tree representation in figure 20. It is easily checked that this tree represents the defoliated information structure correctly and that no guiding conditions are violated.

The corresponding relation type will contain two optional relation-valued attribute types $\{p_{14}\}$ and $\{p_8\}$, since predicators p_{13} and p_7 are neither total

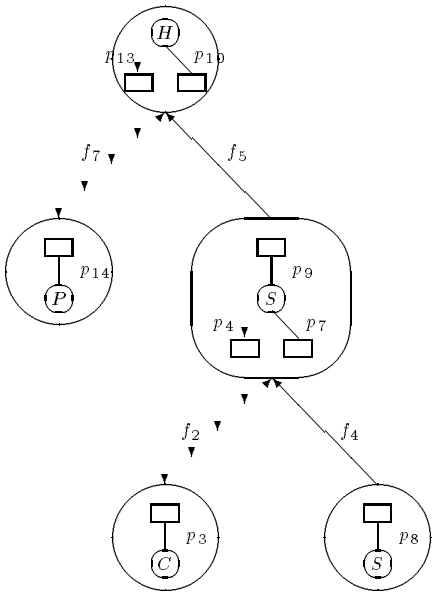

Figure 20: Example tree

nor unique (see example 2.6). Furthermore, redundancy is allowed for fact types f_2 and f_4. This redundancy is caused as follows: the anchor of node $\{p_4, p_7, p_9\}$ is predicator p_9, which is not unique.

Usually different scenarios can result in the same tree representation. The tree in figure 20 may be the result of a generation starting with predicator p_{13}:

> Start with node $\{p_{13}\}$ and process fact type f_7. Then join $\{p_{10}\}$ and $\{p_{13}\}$ and add an edge labelled by fact type f_5. Proceed with $\{p_4\}$ and $\{p_7\}$.

Another scenario that generates this tree starts with predicator p_{10}:

> Start with node $\{p_{10}\}$ and process fact type f_5. Then join $\{p_7\}$ and $\{p_9\}$ and add an edge labelled by fact type f_4. Proceed with $\{p_4\}$ and $\{p_{13}\}$.

Both scenarios mentioned above contain a permutation of the predicators p_{13}, p_{10}, p_4 and p_7. Not every possible sequence of these predicators will result in the same tree. Firstly, it depends on the choice for a node to be extended. Furthermore, only if both p_4 and p_7 are chosen after p_{10}, the result will be the same.

The tree structure shown in figure 20 corresponds to the following hierarchical relation type:

$$\left[\ \overline{\{p_{13}, p_{10}\}}\ , \{p_{14}\}\ \mathsf{oprep}, \{p_9, p_4, p_7\}\ , \{p_3\}\ , \{p_8\}\ \mathsf{oprep}\right]$$

Each set of predicators can be replaced by the naming reference (identification) of its base. These naming references were discussed in example 2.6. For example, $\{p_{13}, p_{10}\}$ has base H. Entity type H has naming reference $\langle$H-nr, S-name, C-name$\rangle$, or $\langle H, S, C\rangle$ for short. The resulting relation type then is:

$$\left[\ \overline{\langle H, S, C\rangle}\ , \langle P\rangle\ \mathsf{oprep}, \langle S, C\rangle\ , \langle C\rangle\ , \langle S, C\rangle\ \mathsf{oprep}\right]$$

This relation type can be simplified to a large extend. It is dominated by the tuple $\langle H, S, C\rangle$. Naming reference $\langle S, C\rangle$ corresponding to node $\{p_9, p_4, p_7\}$ contains the same S and C values as naming reference $\langle H, S, C\rangle$. This $\langle S, C\rangle$ therefore can be omitted. For the same reason $\langle C\rangle$ is obsolete. Note that $\langle S, C\rangle$ corresponding to node $\{p_8\}$ cannot be treated in the same way, as it is the result of fact type f_4 which is *not* part of the identification of another base.

Finally, leaving out the sharp brackets, we get:

$$[\ \overline{H, S, C}\ , P\ \mathsf{op\ rep}, [S, C]\mathsf{op\ rep}]$$

6.2 Size / depth preference without optionals

Consider the following setting of parameters:

NoOptionals = True
SizePreference = True
$\gamma = 2$
DepthPreference = True
$\delta = 2$

The result of the generation will be a forest where each tree contains at most 2 fact types and has maximal depth 2, such that optional attribute types do not occur. The tree representation in figure 21 satisfies these conditions.

It can be reached via the following scenario:

1. Start with node $\{p_4\}$ and process fact type f_2.
2. Then join $\{p_4\}$ and $\{p_9\}$ and add an edge labelled by fact type f_5.
3. Proceed with $\{p_{14}\}$. It becomes a new root because the tree does not contain a node with base P yet. Add an edge from $\{p_{13}\}$ to $\{p_{14}\}$ labelled by f_7.
4. Finally, create a new root $\{p_8\}$. For two reasons it is not possible to add p_8 to $\{p_4, p_9\}$.

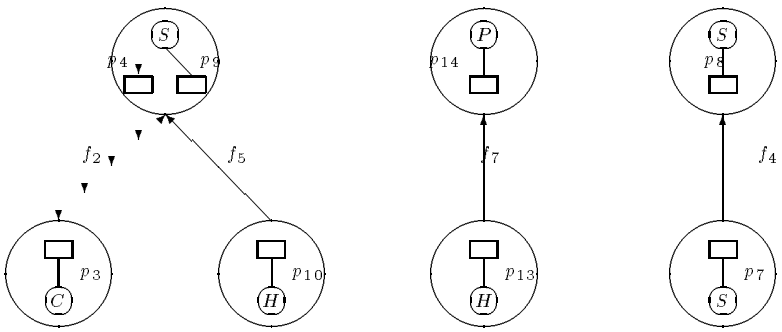

Figure 21: Example forest

Firstly, parameter NoOptionals only allows the extension of node $\{p_4, p_9\}$ with predicators having a total role constraint. Secondly, the tree having $\{p_4, p_9\}$ as root already contains the maximal number of fact types.

The resulting tree representation corresponds to the following relation types:

$\left[\overline{\{p_4, p_9\}}, \{p_3\}, \{p_{10}\} \text{ rep} \right]$

$\left[\overline{\{p_{14}\}}, \{p_{13}\} \right]$

$\left[\overline{\{p_8\}}, \{p_7\} \text{ rep} \right]$

Replacing the sets of predicators by the naming references of their bases, we get:

$\left[\overline{\langle S, C\rangle}, \langle C\rangle, \langle H, S, C\rangle \text{ rep} \right]$

$\left[\overline{\langle P\rangle}, \langle H, S, C\rangle \right]$

$\left[\overline{\langle S, C\rangle}, \langle S, C\rangle \text{ rep} \right]$

In the first relation type the values of $\langle C\rangle$ also occur in $\langle S, C\rangle$, since f_2 is part of the identification of entity type S. Therefore, this $\langle C\rangle$ can be omitted.

7 Conclusions

In this paper we discussed the automatic generation of database structures for a given conceptual schema. A mechanism for the guidance of the generation process was introduced.

For the specification of conceptual schemata a general object-role platform was used. These schemata were hierarchically represented in trees consisting of nodes of predicators, the central notion of the platform. Then, a simple generation algorithm for these tree representations was introduced, leaving many opportunities to influence the nature of the result.

In this way the correctness of the algorithm could be easily guaranteed. Furthermore, it was possible to specify guidance parameters, in order to yield structures having desirable properties, involving e.g. redundancy and table size.

We did not consider more sophisticated search algorithms in this paper. This generation algorithm can be used as a base when searching an optimal representation (subject to some cost function). Another problem that can be addressed is finding suitable candidate keys for the generated internal representation. In a complicated schema, that might be a problem. Finally, future research will consist of: (1) the incorporation of object-orientation and complex objects in terms of object-role models (see also [23] and [9]) and (2) the incorporation of additional background in database storage structures.

Acknowledgement

We would like to thank the anonymous referees for their contributive remarks.

References

[1] S. Abiteboul, P.C. Fischer, and H.J. (Eds.) Schek. *Nested Relations and Complex Objects in Databases.* Springer Verlag, 1987.

[2] A.D. Atri and D. Sacca. Equivalence and mapping of database schemes. In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 187–195, 1984.

[3] P. van Bommel, A.H.M. ter Hofstede, and Th.P. van der Weide. Semantics and verification of object-role models. *Information Systems*, 16(5), October 1991.

[4] P. van Bommel and Th.P. van der Weide. Towards database optimization by evolution. In *Proceedings Computing Science in The Netherlands (CSN) 1991*, pages 109–123, November 1991.

[5] P.P. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[6] L.S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

[7] H.M. Dreizen and S.K. Chang. Imprecise schema: a rationale for relations with embedded subrelations. *ACM Transactions on Database Systems*, 14(4):447–479, December 1989.

[8] A.H.M. ter Hofstede and Th.P. van der Weide. Formalisation of techniques: Chopping down the methodology jungle. Technical report, Department of Information Systems, University of Nijmegen, The Netherlands, dec 1990. To be published.

[9] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in Data Modelling. Technical Report 91/07, SERC, Software Engineering Research Centrum, Utrecht, The Netherlands, July 1991.

[10] Isamu Kobayashi. Classification and transformations of binary relationship relation schemata. *Information Systems*, 11(2):109–122, 1986.

[11] C.M.R. Leung and G.M. Nijssen. Relational database design using the NIAM conceptual schema. *Information Systems*, 13(2):219–227, 1988.

[12] Luqi. Automated prototyping and data translation. *Data and Knowledge Engineering*, 5:167–177, 1990.

[13] P Lyngbaek and V. Vianu. Mapping a semantic database model to the relational model. In *ACM SIGMOD*, pages 132–142, 1987.

[14] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 1987.

[15] G.M. Nijssen and T.A. Halpin. *Conceptual schema and Relational Database Design: A fact oriented approach.* Prentice Hall of Australia Pty Ltd, 1989.

[16] Z.M. Ozsoyoglu and L.Y. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.

[17] N. Prabhakaran. *Generation of Relational Database Schemata and its Applications.* PhD thesis, Department of Computer Science University of Queensland, Australia, 1984.

[18] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[19] H.J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[20] M.H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. Technical report, Fachbereich Informatik, Technische Hochschule Darmstadt, West Germany, 1986. DVSI-1986-T3.

[21] P. Shoval and M. Even-Chaime. ADDS: A system for automatic database schema design based on the binary-relationship model. *Data and Knowledge Engineering*, 3(2):123–144, 1987.

[22] T.J. Teory, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *Computing Surveys*, 18(2):197–222, 1986.

[23] O. de Troyer. The OO-Binary Relationship Model: A Truly Object Oriented Conceptual Model. In R. Andersen, J.A. Bubenko, and A. Sølvberg, editors, *Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering*, pages 561 – 578, Trondheim, Norway, May 1991. Lecture Notes in Computer Science 498.

[24] O.M.F. de Troyer. RIDL*: A Tool for the Computer-Assisted Engineering of Large Databases in the Presence of Integrity Constraints. Technical report, Institute for Language Technology and Artificial Intelligence ITK, Tilburg University, 1989. ITK Research Report No. 3.

[25] S. Twine. Mapping between a NIAM conceptual schema and KEE frames. *Data and Knowledge Engineering*, 4(2):125–155, 1989.

[26] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[27] Th.P. van der Weide, A.H.M. ter Hofstede, and P. van Bommel. The uniquest algorithm: A formal semantics of complex uniqueness constraints. Technical Report 90-19, Department of Information Systems, University of Nijmegen, The Netherlands, October 1990. To be published.