Generating Significant Examples for Conceptual Schema Validation

Confidential

Asymetrix Report 94-3

H.A. Proper
Asymetrix Research Laboratory
Department of Computer Science
University of Queensland
Australia 4072
E.Proper@acm.org

Version of June 23, 2004 at 10:29

PUBLISHED AS:

H.A. Proper. Generating significant examples for conceptual schema validation. Asymetrix Research Report 94-3, Asymetrix Research Laboratory, University of Queensland, Brisbane, Australia, 1994.

Abstract

This report bases itself on the idea of using concrete examples to verify conceptual schemas, and in particular cardinality constraints. When novice ORM modellers model domains, the selection of proper cardinality constraints for relationship types is quite often prone to errors. In this report we propose a mechanism for the generation of significant examples for selected subschemas. The generated examples are significant in the sense that they illustrate the possible combinations of instances that are allowed with respect to the cardinality constraints on the involved relationship types.

In this report we firstly provide a brief informal discussion of the basic idea. Then we present a syntactic mechanism to select the subschema for which example instances are to be generated. This is followed by the actual example generation algorithm itself. We will also present, as a *spin-off*, an algorithm that allows us

to detect possible flaws in the conceptual schema by calculating the number of instances that can be used to populate the types in the schema.

1 Introduction

A key aspect in the conceptual design procedure ([Hal95]) is the use of examples to derive the initial design of conceptual schemas. A further use of the examples is the validation of parts of the final conceptual schema. Example populations of relationship types can be used to validate the correctness of the information structure, and even more importantly, for the validation of constraints. In this report we propose a mechanism to generate significant example populations for subsets of conceptual schemas. This idea has been described informally in [Har94]. In general, generating a significant example population is hardly possible. Therefore, we limit ourselves to cardinality constraints and significance of the examples with respect to single relationship types only.

The sample populations are used to visualise for the users what the effects are of adding or removing cardinality constraints on relationship types, and to a lesser extent that of changing the information structure itself. An informal discussion of this idea can be found in [Har94]. This proposed mechanism will initially be used in DBCreate and is expected to migrate to InfoModeler in a later stage; after the idea has been tested by the user community.

The aim of DBCreate is to enable (semi!) laymen to design their own database. The example generation tool fits quite well into that idea. Note that it is only obvious that these semi laymen are still presumed to have some basic knowledge of conceptual (and preferably ORM) modelling. No matter how user friendly a CAD/CAM system may become, an architect desiging a house is still required to have a basic working knowledge of the design of houses.

As stated before, in general it is nearly impossible to construct sample populations that are truely significant with respect to all constraints ([BHW91]). In this report we therefore define a significant population of a relationship type to be a population that shows all allowed combinations of instances with respect to the cardinality constraints defined on that relationship type. Setting a less limited goal can easily lead to a combinatoric explosion. The restrictions we made, however, are not an unreasonable limitation in the context of our aims. The most commonly used (and thus mis-used) constraints are the cardinality constraints (totality and uniqueness). For the other constraint types there is of course still the possibility of verbalising them in a semi-natural language format.

The example generation tool itself consists of two basic elements. Firstly, the user must be able to select parts of a conceptual schema, and put them on the screen in an orderly way. These parts together form a tree[1]. As an example, consider figure 1. This screen

[1] One could argue that this should actually be a sequence of trees, however, due to the limited size of PC Screens and user's capabilities to deal with large amount of information at the same time, it is probably wiser to limit ourselves to a single tree of limited size.

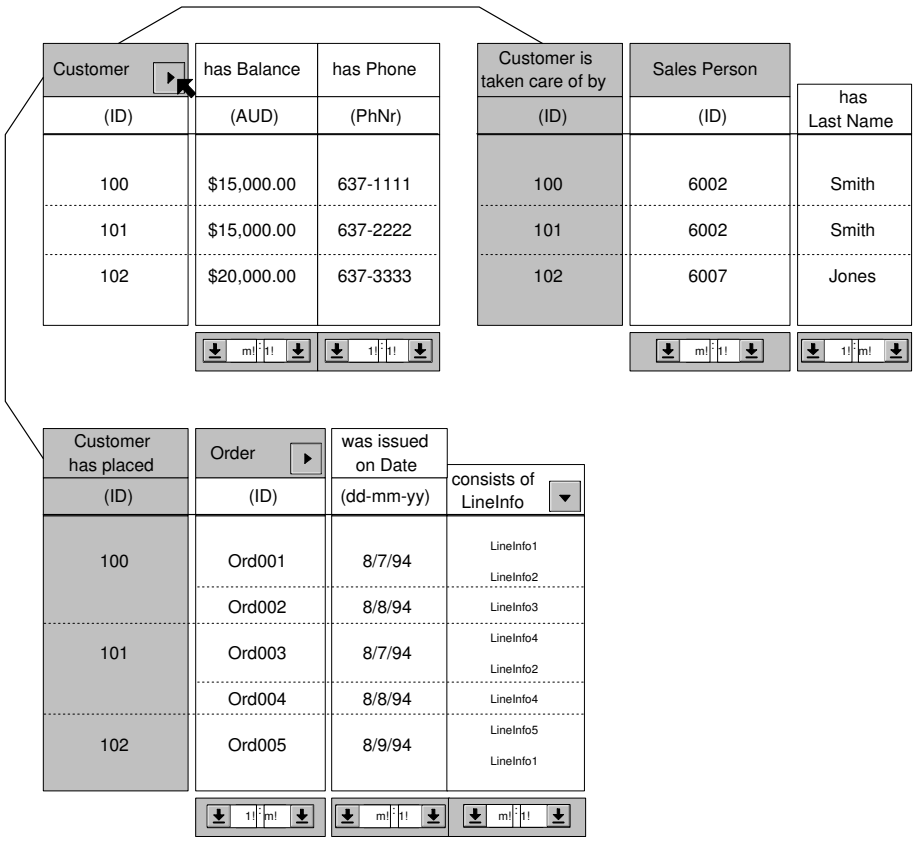

Figure 1: An Example Grid

depicts four interconnected tables, and is based on the schema depicted in figure 2.

In [Har94], similar examples can be found that are discussed in more detail than we do here. The inter-predicate (inter relationship type) uniqueness constraint shown in figure 2 can not be handled as such by the example generation algorithm, as the algorithm focusses on each relationship type separately during the generation process. However, a natural solution appears when realising that when enforcing an inter-predicate uniqueness constraint, this constraint is actually enforced on a derived relationship. In figure 3 the inter-predicate uniqueness constraint from figure 2 is converted to an intra-predicate uniqueness constraint on a derived relationship. In [Hof93] and [WHB92] an algorithm is provided to actually construct this derived relationhip type as part of the semantics of inter-predicate uniqueness constraints.

An optional feature helping the user in better understanding the structure of the displayed examples is illustrated in figure 4 and 5. These figures illustrate a possible

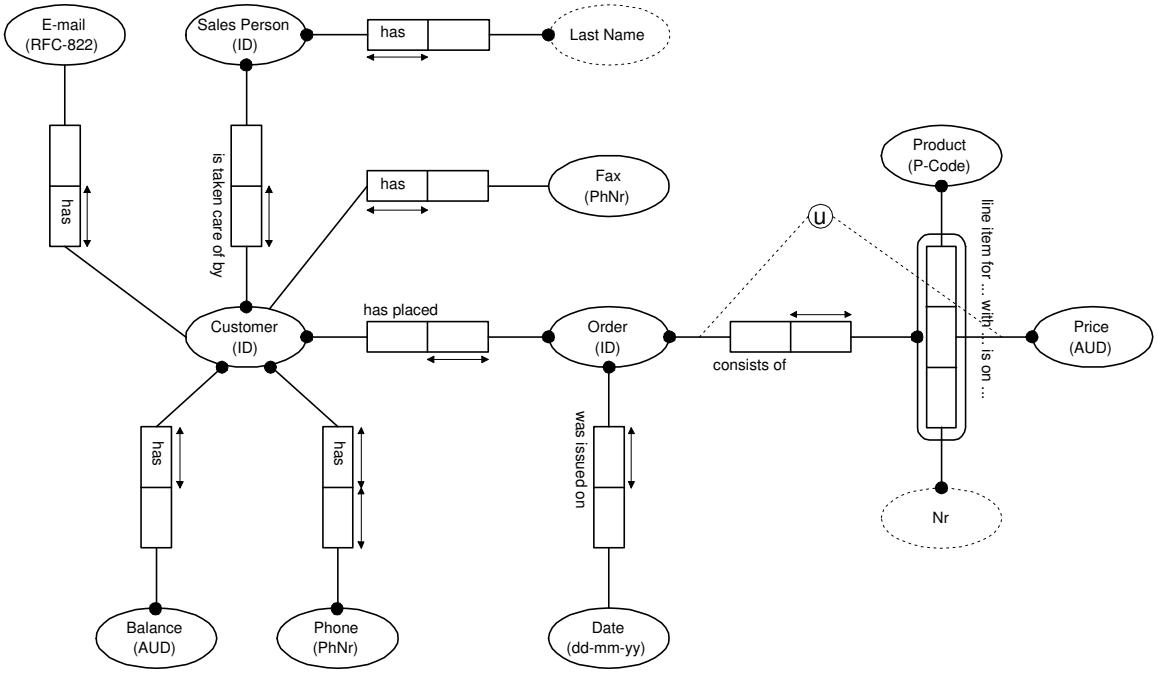

Figure 2: Example Schema

mouse over effect. When the mouse cursor is over one of the values in a table, arrows are shown illustrating the connections between the instances in the table.

Two further aspects of the example in figure 1 that are noteworthy, as we elaborate them further in the remainder of the report, are the right-button following Customer and the down-button following Line Info. The first button is used to indicate that more facts about customers are stored than currently shown on the screen, i.e. the tree could be breathened. This is even better illustrated in figure 6. The second button indicates that Line Info is a compositely identified object type (nested relationship types are regarded as compositly identified object types as well). Clicking on this button leads to the screen depicted in figure 7. In this screen the Line Info column is split according to the reference schema of object type Line Info. Note that the down-button is now replaced by an up-button to indicate that the details can be hidden again if so desired.

As stated before, an example grid can be seen as a tree. In figure 8 we have depicted the tree that can be associated to this sample grid. The tree underlying a sample grid is constructed by repeatedly selecting items from the conceptual schema and adding them to the existing tree. Initially, there is no order provided in which the nodes of the tree should be put on the screen. However, the items in the header can be shifted around by users at will. Alternatively, one could let the system re-shuffle the entire tree using

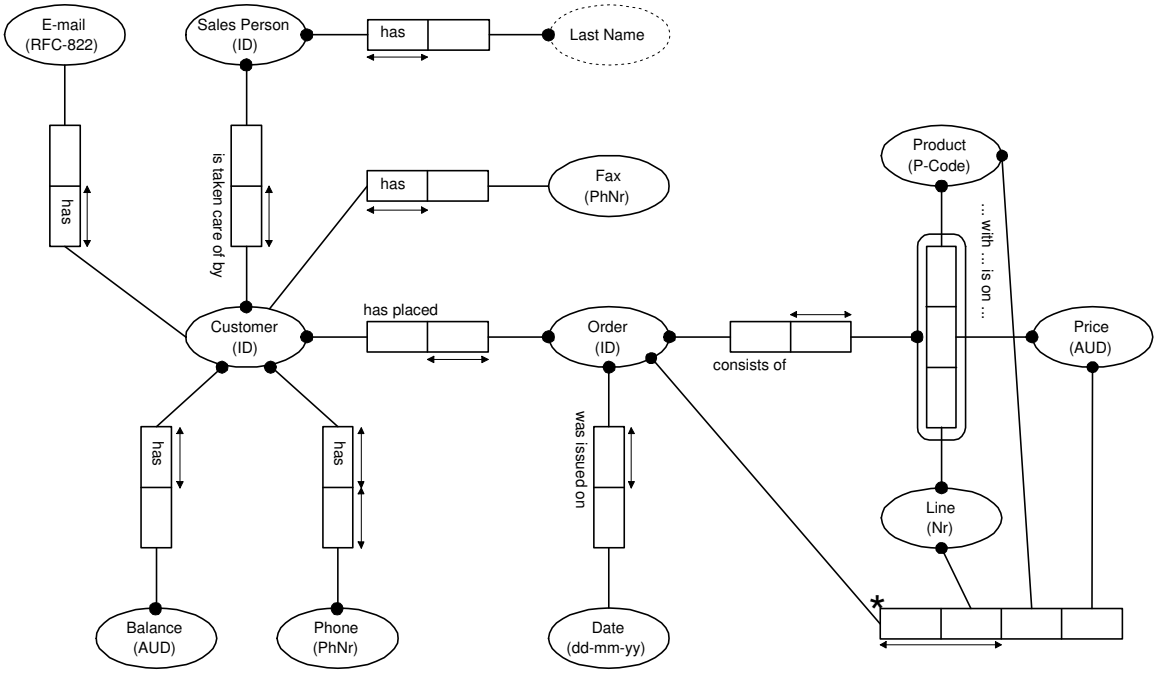

Figure 3: Dealing with Inter-Predicate Constraints

the conceptual relevance ([CH94]) of the object types involved as an ordering criterion. Furthermore, the spider query mechanism ([Pro94a]) could be used to quickly put complete trees on the screen.

The second element of the example generation tool is the generator of the examples itself. Given a tree as depicted in figure 8, a significant population for the relationship types contained (in effect the edges) in the tree must be constructed. As stated before, the significance in this first setup is limited to the generation of all valid combinations for the relationship types in the tree. We limit ourselves to the intra predicate uniqueness and totality constraints only.

The structure of this report is as follows, in section 2 we define the syntax of the trees underlying the sample grids, and provide a brief formalisation of the required concepts of Object-Role Modelling. In section 3 we discuss a negotiation mechanism for the size of the population of the edges in the trees. The generation of the actual sample population is covered in section 4. Finally, section 5 concludes the report. For the reader who is unfamiliar with the notation style used in this report, it is advisable to first read [Pro94b].

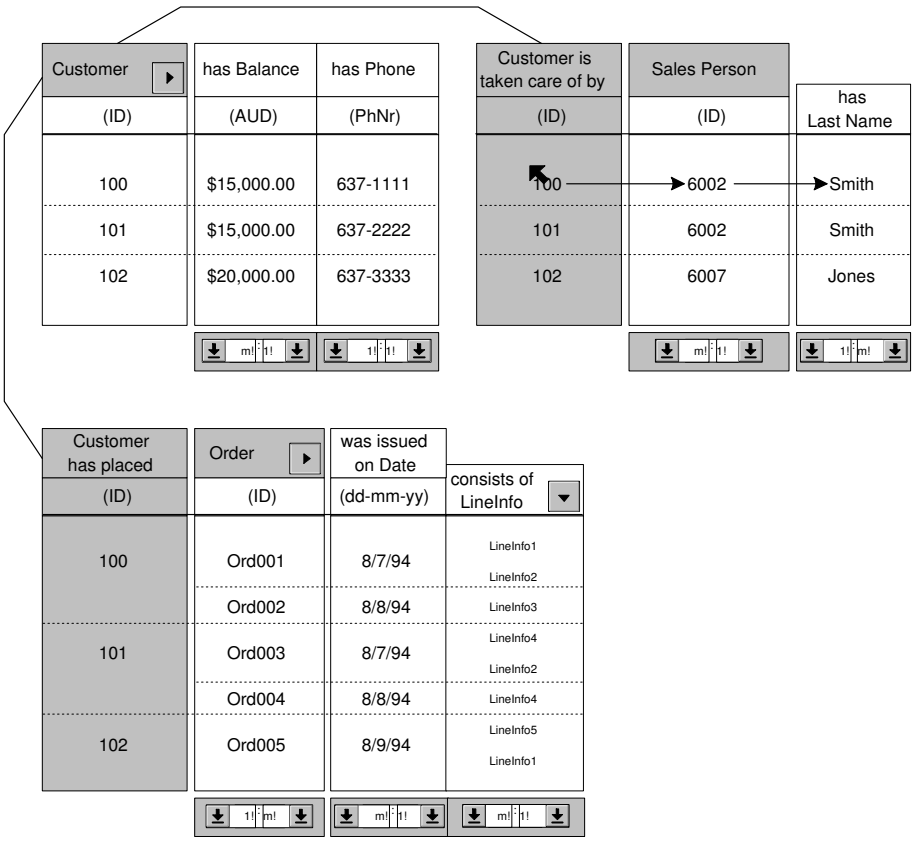

Figure 4: An Example Mouse Over

2 Creating a Forrest

This section discusses the syntax of a forrest for the example grid and also provides a brief formalisation of the ORM concepts needed. Although the example generation tool will initially be used in the context of DBCreate (value types, entity types and binary relationship types), we already allow for ORM schemas as used by InfoModeler. We start out from a formalisation of ORM based on the one used in ([HP95]). However, since only a limited part of the formalisation is needed, we do not cover the formalisation in full detail.

2.1 ORM Basis

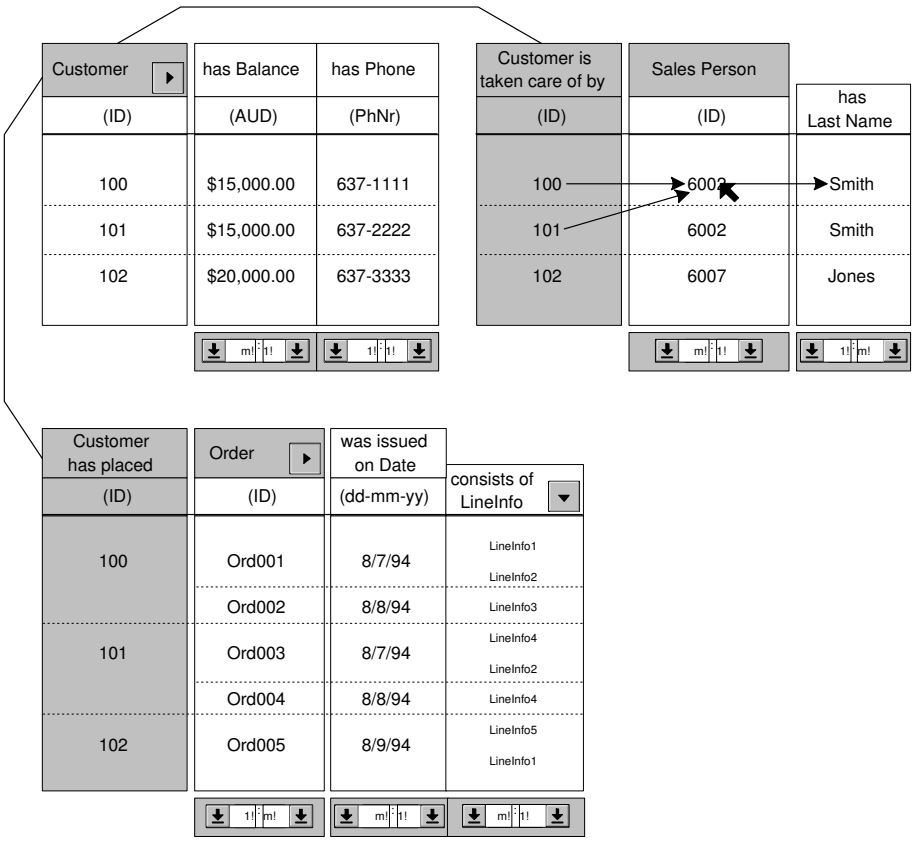

Figure 5: Another Mouse Over

A conceptual schema is presumed to consist of a set of types $\mathcal{TP}$. Within this set of types two subsets can be distinguished: the relationship types $\mathcal{RL}$, and the object types $\mathcal{OB}$. Furthermore, let $\mathcal{RO}$ be the set of roles in the conceptual schema. The fabric of the conceptual schema is then captured by two functions and two predicates. The set of roles associated to a relationship type are provided by the partition: $\mathsf{Roles} : \mathcal{RL} \rightarrow \wp(\mathcal{RO})$. Using this partition, we can define the function Rel which returns for each role the relationship type in which it is involved: $\mathsf{Rel}(r) = f \iff r \in \mathsf{Roles}(f)$. Every role has an object type at its base called the player of the role. This player is formally provided by the function: $\mathsf{Player} : \mathcal{RO} \rightarrow \mathcal{TP}$. Subtyping of object types is captured by the predicates $\mathsf{SpecOf} \subseteq \mathcal{OB} \times \mathcal{OB}$. Using SpecOf we can define the notion of type relatedness: $x \sim y$ for object types x and y. This notion captures the intuition that two object types may share instances. This relation is defined by the following four derivation rules:

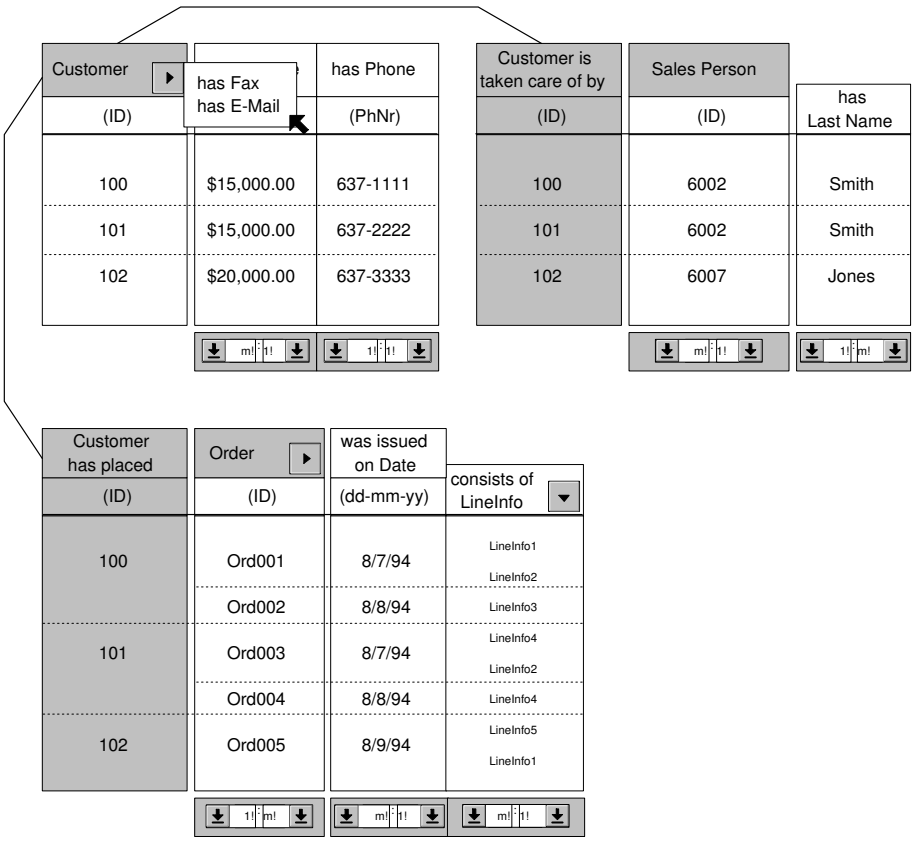

Figure 6: Extendable Column

1. $x \in \mathcal{TP} \vdash x \sim x$
2. $x \,\mathsf{SpecOf}\, y \vdash x \sim y$
3. $x \sim y \vdash y \sim x$
4. $x \sim y \sim z \vdash x \sim z$

Note that when using ORM with the advanced concepts ([HP95]) such as polymorphism, sequence types, set types, etc., the definition of $\sim$ needs to be refined.

Instances of all non-value types must be identified in terms of instances of other object types. This identification is usually provided by a so called *reference schema*. If $\mathcal{VL}$ denotes the set of value types, then the (direct!) identification relationship between types is presumed to be captured by the function:

$$\mathsf{RefSch} : (\mathcal{TP} - \mathcal{VL}) \rightarrow \mathcal{TP} \cup \mathcal{RO}^+ \cup (\mathcal{RO} \times \mathcal{RO})^+$$

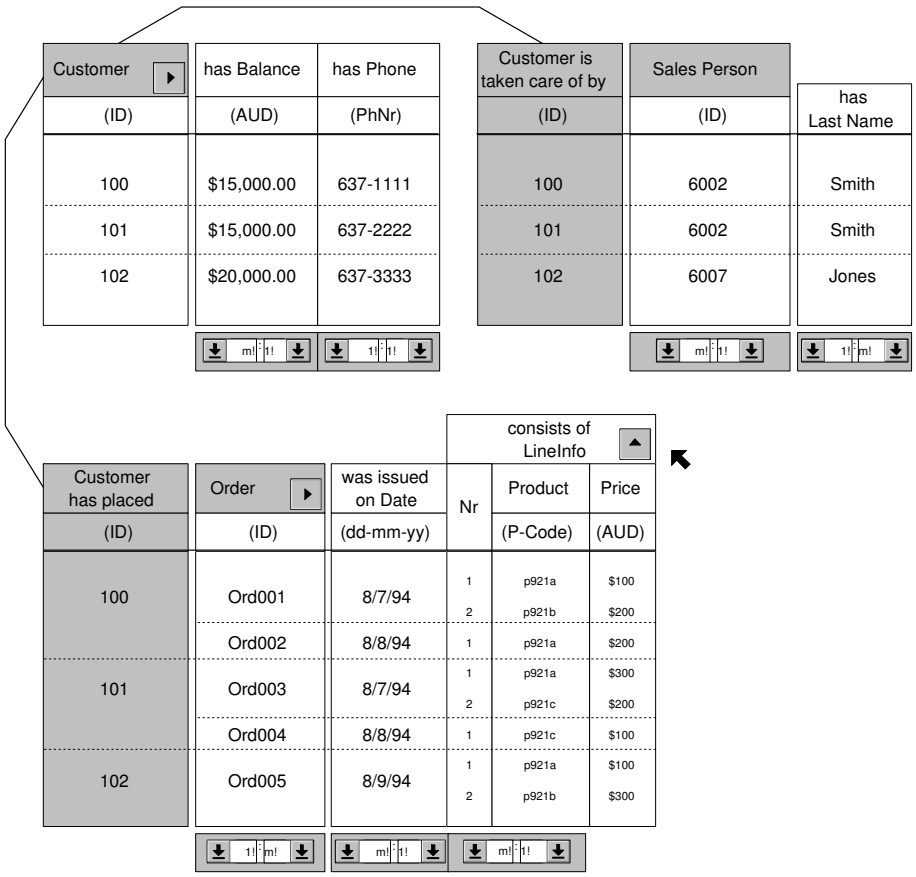

Figure 7: Exploded View on Line Items

Each non-value type is either identified by a type (a super type), or a sequence of roles (a relationship type), or a sequenc of role pairs (compositely identified object types). Note that in this report we do not concern ourselves with well-formedness rules on reference schemas. A function that is derived from RefSch and which is needed in the remainder is IdfObjs : $\mathcal{OB} \rightarrow \mathcal{OB}^+$. This function returns the sequence of object types needed to directly identify a given object type. Its definition is provided as:

$$\mathsf{IdfObjs}(x) \triangleq \begin{cases} [y] & \text{if } \mathsf{RefSch}(x) = [y] \\ [y_1, \ldots, y_l] & \text{if } \mathsf{RefSch}(x) = [\langle p_1, y_1 \rangle, \ldots, \langle p_l, y_l \rangle] \\ [y_1, \ldots, y_l] & \text{if } \mathsf{RefSch}(x) = [\langle p_1, q_1, y_1 \rangle, \ldots, \langle p_l, q_l, y_l \rangle] \end{cases}$$

Note that we presume the existance of an implicit coercion function between sequences and sets. So, for example, if S is a set of sequences we allow ourselves to write $\cup S$ for

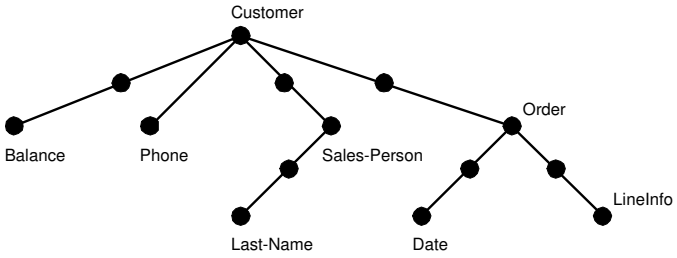

Figure 8: An Example Tree

the set of all elements occurring in the sequences in S.

A lot of the decissions made by the example generation algorithm are based on the maximum size of the populations of the object types. Later on, an algorithm to calculate these sizes is presented. However, the maximum sizes of the populations of the value types should be given by the modeller. For instance, a value type representing the gender of persons will usually contain at most two instances. To accommodate this, we presume the existance of the function $\mathsf{DomSize} : \mathcal{VL} \rightarrow \mathbb{N}$. One obvious requirement for this function is: $x, y \in \mathcal{VL} \wedge x\ \mathsf{SpecOf}\ y \Rightarrow \mathsf{DomSize}(x) \leq \mathsf{DomSize}(y)$.

Finally, in this report an important role is played by the uniqueness and tolality constraints. For that purpose, we presume the predicates $\mathsf{Unique} \subseteq \wp(\mathcal{RO})$ and $\mathsf{Total} \subseteq \wp(\mathcal{RO})$ to provide all uniqueness and totality constraints.

In this report we thus only use the following components of an ORM schema:

$$\langle \mathcal{TP}, \mathcal{RL}, \mathcal{VL}, \mathcal{OB}, \mathcal{RO}, \mathsf{SpecOf}, \mathsf{Roles}, \mathsf{Player}, \mathsf{RefSch}, \mathsf{DomSize}, \mathsf{Unique}, \mathsf{Total} \rangle$$

When implementing the algorithms and ideas presented in this article, these components are the *interface* between the example generator tool and the meta model from the fact base.

2.2 Forrests

A tree of an example grid is built from a set of nodes. Let in the remainder $\mathbb{N}$ be a set of all such nodes. Formally, a tree can now be defined by two functions: EOut and Obj. As one object type can be represented by more than one node, the object type represented by a node is provided by the function: $\mathsf{Obj} : \mathbb{N} \rightarrowtail \mathcal{OB}$. The edges of the tree are provided by the function $\mathsf{EOut} : \mathbb{N} \rightarrow \wp(\mathbb{L} \times \mathbb{N})$. This function provides for each node the set of outgoing edges. Using the EOut function the following "inverse" function can be derived, which returns for each node the set of incomming edges:

$$\mathsf{EIn} : \mathbb{N} \rightarrow \wp(\mathbb{L} \times \mathbb{N})$$

$$\mathsf{EIn}(n) \triangleq \left\{ \langle l, m \rangle \;\middle|\; \langle l, n \rangle \in \mathsf{EOut}(m) \right\}$$

The edges of the tree are labelled with link information. The set of links $\mathbb{L}$ is defined by: $\mathbb{L} \triangleq \mathcal{RO} \cup \mathcal{RP}$, where $\mathcal{RP} \triangleq \left\{ p^{\leftarrow} \;\middle|\; p \in \mathcal{RO} \right\}$ is the set of *reversed* roles. The reversed roles can be used to connect a node representing a relationship type to a node representing one of the participating object types. Each link has a starting point and an ending point. To access these points (object types) uniformly, we introduce the following two generic functions:

$$\mathsf{Start}, \mathsf{End} : \mathbb{L} \rightarrow \mathcal{TP}$$

$$\mathsf{Start}(x) \triangleq \begin{cases} \mathsf{Player}(x) & \text{if } x \in \mathcal{RO} \\ \mathsf{Rel}(x) & \text{otherwise} \end{cases}$$

$$\mathsf{End}(x) \triangleq \begin{cases} \mathsf{Rel}(x) & \text{if } x \in \mathcal{RO} \\ \mathsf{Player}(x) & \text{otherwise} \end{cases}$$

Note: from now on we presume Rel and Player to be generalised to elements from $\mathcal{RP}$ in the obvious way.

In order for EOut and Obj to span a tree, they must adhere to certain properties. Each edge in the tree must be a connection between the source and destination of the edge via the role. This is expressed by the following axiom:

[T1] For each $l \in \mathbb{L}$:

$$\langle l, m \rangle \in \mathsf{EOut}(n) \Rightarrow \mathsf{Start}(l) \sim \mathsf{Obj}(n) \wedge \mathsf{Obj}(m) = \mathsf{End}(l)$$

The function EOut must indeed define a tree, so the graph spanned by EOut has to be a connected, acyclic graph with a unique root.

[T2] $|\pi_2(\cup \mathsf{ran}(\mathsf{EOut}))| = |\mathsf{ran}(\mathsf{EOut})|$

[T3] $\exists!_{x \in \mathsf{dom}(\mathsf{EOut})} [x \notin \pi_2(\cup \mathsf{ran}(\mathsf{EOut}))]$

Note: if a directed graph has a unique root then it is automatically connected. All used nodes must have an object type associated:

[T4] $\mathsf{dom}(\mathsf{EOut}) \cup \pi_2(\cup \mathsf{ran}(\mathsf{EOut})) \subseteq \mathsf{dom}(\mathsf{Obj})$

Furthermore, a link can be used only once for an outgoing edge of a node. This is formally enforced by:

[T5] $\langle l, n_1 \rangle \in \mathsf{EOut}(m) \wedge \langle l, n_2 \rangle \in \mathsf{EOut}(m) \Rightarrow n_1 = n_2$

One of the items that can be shown when displaying nodes on screen is the fact that a given node can be extended with more edges. In figure 1, the right-button behind the Customer indicated that more relationship types are available for this object type. A node can be extended with an extra edge iff a link exists that can form a proper edge, and is not already used by on another edge of this node. This property can be expressed formally as:

$$\mathsf{CanExtend}(n) \iff \left\{ l \in \mathcal{RO} \;\middle|\; \mathsf{Start}(l) \sim \mathsf{Obj}(n) \wedge l \notin \pi_1\,\mathsf{EOut}(n) \right\} \neq \varnothing$$

The set in the righthand side of the above definition can actually be used to fill the listbox displayed in figure 6.

The order in which the nodes themselves are displayed on the screen is recorded by the function $\mathsf{Order} : \mathbb{N} \rightarrow \mathbb{N}$. As this function must provide a total order of the nodes, we should have:

$$\mathsf{Order}(x_1) = \mathsf{Order}(x_2) \Rightarrow x_1 = x_2$$

One additional option of a system using an example generator is to have the system re-order the tree based on the conceptual relevance of the object types represented by the nodes. If $\mathsf{CWeight} : \mathcal{TP} \rightarrow \mathbb{N}$ is a function returning the conceptual weight of types, then the system is able to order the nodes such that:

$$\mathsf{Order}(n_1) < \mathsf{Order}(n_2) \Rightarrow \mathsf{CWeight}(\mathsf{Obj}(n_1)) \leq \mathsf{CWeight}(\mathsf{Obj}(n_2))$$

Note: more than one order may exist for the same CWeight values since differing object types may have the same conceptual weight. Not all nodes need to be displayed on the screen. Some nodes can be left implicit. For instance, a node representing a non-objectified binary relationship does not have to be shown on the screen. The set of implicit nodes in a tree is identified by:

$$\mathbb{I} \triangleq \left\{ n \in \mathbb{N} \;\middle|\; \pi_1\,\mathsf{EIn}(n) \cap \mathcal{RP} = \varnothing \wedge \pi_1\,\mathsf{EOut}(n) \cap \mathcal{RO} = \varnothing \wedge \mathsf{EIn}(n) \neq \varnothing \wedge \mathsf{EOut}(n) \neq \varnothing \right\}$$

From this definition immediately follows that two neighbouring nodes cannot both be implicit. This leads to the following lemma:

Lemma 2.1 (*no implicit neighbours*) $n \in \mathbb{N} \wedge m \in \pi_2\,\mathsf{EOut}(n) \Rightarrow \{n, m\} \not\subseteq \mathbb{I}$

Proof:

Let $n \in \mathbb{N} \wedge m \in \pi_2\,\mathsf{EOut}(n)$ such that $\{n, m\} \subseteq \mathbb{I}$.

Since $m \in \pi_2\,\mathsf{EOut}(n)$ it immediately follows from the definition of EIn that:

$$\pi_1\,\mathsf{EIn}(m) = \pi_1\,\mathsf{EOut}(n)$$

As we presumed that $n, m \in \mathbb{I}$ we in particular have:

$$\pi_1\,\mathsf{EOut}(n) \cap \mathcal{RO} = \varnothing \text{ and } \pi_1\,\mathsf{EIn}(m) \cap \mathcal{RP} = \varnothing$$

Since $\pi_1\,\mathsf{EIn}(m) = \pi_1\,\mathsf{EOut}(n)$ we have:

$$\pi_1\,\mathsf{EOut}(n) \cap \mathcal{RO} = \varnothing \text{ and } \pi_1\,\mathsf{EOut}(n) \cap \mathcal{RP} = \varnothing$$

Which implies that $\mathsf{EOut}(n)$ which is a contradiction since $m \in \pi_2\,\mathsf{EOut}(n)$.

Therefore we can not have $\{n, m\} \subseteq \mathbb{I}$ if $n \in \mathbb{N} \wedge m \in \pi_2\,\mathsf{EOut}(n)$. □

As an example, consider the tree depicted on the right side of figure 9 in the context of the schema shown there as well. The open circles represent nodes that do not have to be shown on the screen when this tree is shown to the user.

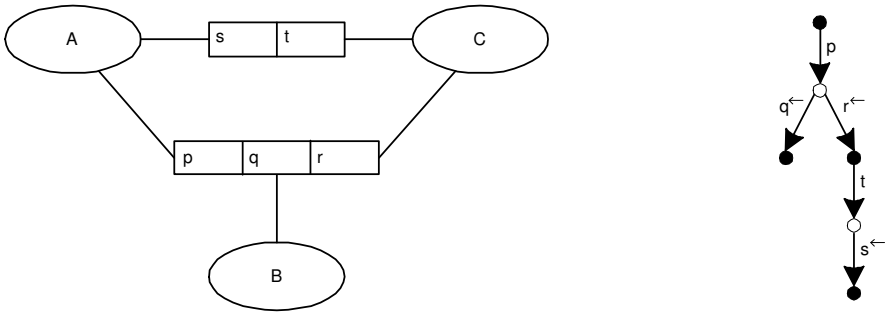

Figure 9: Examples of Implicit Nodes

A further result is the following lemma stating that each implicit node must correspond to a relationship type, and that the predicators used to label the incoming and outgoing edges are all of the same relationship type:

Lemma 2.2 (*single relationship type involvement*) $n \in \mathbb{I} \Rightarrow \forall_{l \in \pi_1(\mathsf{EOut}(n) \cup \mathsf{EIn}(n))}\,[\mathsf{Rel}(l) = \mathsf{Obj}(n)]$

Proof:

If $n \in \mathbb{I}$ then we must, due to the definition of $\mathbb{I}$ have $l \in \pi_1\,\mathsf{EIn}(n) \Rightarrow l \in \mathcal{RO}$. This implies that $\mathsf{Rel}(l) = \mathsf{Obj}(n)$. So $\forall_{l \in \pi_1\,\mathsf{EIn}(n)}\,[\mathsf{Rel}(l) = \mathsf{Obj}(n)]$

Furthermore, if $l \in \pi_1\,\mathsf{EOut}(n)$, it follows from the definition of $\mathbb{I}$ that $l \in \mathcal{RP}$. From axiom T1 follows that $\mathsf{Start}(l) \sim \mathsf{Obj}(n)$, which can be reformulated as $\mathsf{Rel}(l) \sim \mathsf{Obj}(n)$. Since two relationship types can, in ORM, only be type related if they are the same, we therefore have: $\mathsf{Rel}(l) = \mathsf{Obj}(n)$. As a result: $\forall_{l \in \pi_1\,\mathsf{EOut}(n)}\,[\mathsf{Rel}(l) = \mathsf{Obj}(n)]$. □

To cater for identification, and in particular complex identification, nodes can have associated a number of identifying nodes. This is captured by the function

$$\mathsf{NRefSch} : \mathbb{N} \rightarrowtail \mathbb{N} \cup (\mathcal{RO} \times \mathbb{N})^+ \cup (\mathcal{RO} \times \mathcal{RO} \times \mathbb{N})^+$$

which can defines an (possibly empty) identification tree for each node in the tree provided by EOut. This function must always behave conform the identification given in the schema:

[T6] For each $n \in \mathsf{dom}(\mathsf{NRefSch})$ we should have:

1. $\mathsf{NRefSch}(n) = [m] \Rightarrow \mathsf{RefSch}(\mathsf{Obj}(n)) = [\mathsf{Obj}(m)]$
2. $\mathsf{NRefSch}(n) = [\langle p_1, m_1 \rangle, \ldots, \langle p_l, m_l \rangle] \Rightarrow$
   $\mathsf{RefSch}(\mathsf{Obj}(n)) = [p_1, \ldots, p_l] \wedge \forall_{1 \leq i \leq l} [\mathsf{Player}(p_i) = \mathsf{Obj}(m_i)]$
3. $\mathsf{NRefSch}(n) = [\langle p_1, q_1, m_1 \rangle, \ldots, \langle p_l, q_l, m_l \rangle] \Rightarrow$
   $\mathsf{RefSch}(\mathsf{Obj}(n)) = [\langle p_1, q_1 \rangle, \ldots, \langle p_l, q_l \rangle] \wedge \forall_{1 \leq i \leq l} [\mathsf{Player}(q_i) = \mathsf{Obj}(m_i)]$

where $m, m_1, \ldots, m_l \in \mathbb{N}$ and $p_1, \ldots, p_l, q_1, \ldots, q_l \in \mathcal{RO}$.

The above axiom can thus be seen as an invariance requirement on the algorithm that builds the tree of the example grid.

Not all nodes used by EOut must necessarily have an identification tree associated. For instance, in the first example grid of the running example, Line Info instances are not denoted by means of their full identification. We used textual representations (surogates) of abstract instances such as: LineInfo1, LineInfo2, etc.

For the remaining axioms on identification trees in the example grids, we need one more derived function. The function $\mathsf{IdfNodes} : \mathbb{N} \rightarrow \mathbb{N}^+$ which (analogously to IdfObjs) determines the set of nodes needed to directly identify a given node. Its definition is provided as:

$$\mathsf{IdfNodes}(x) \triangleq \begin{cases} [y] & \text{if } \mathsf{NRefSch}(x) = [y] \\ [y_1, \ldots, y_l] & \text{if } \mathsf{NRefSch}(x) = [\langle p_1, y_1 \rangle, \ldots, \langle p_l, y_l \rangle] \\ [y_1, \ldots, y_l] & \text{if } \mathsf{NRefSch}(x) = [\langle p_1, q_1, y_1 \rangle, \ldots, \langle p_l, q_l, y_l \rangle] \end{cases}$$

Note again that we presume the existance of an implicit coercion function between sequences and sets.

We can now require each identification tree to be a tree indeed.

[T7] (*acyclic*) If $x \in \mathsf{dom}(\mathsf{NRefSch})$, then we have $\mathsf{NonCyclic}(x, \{x\})$, where:

$$\mathsf{NonCyclic}(x, X) \iff X \cap \mathsf{IdfNodes}(x) = \varnothing \wedge \forall_{y \in \mathsf{IdfNodes}(x)} [\mathsf{NonCyclic}(y, X \cup \{y\})]$$

Note that for simple ORM schemas the fact that NRefSch is a tree for each object type in the tree spanned by EOut, follows directly from axiom T6 and the acyclicity of the identification trees spanned by RefSch. However, when using the polymorphism concept in more advanced ORM models, in particular when defining recursive data

structures, RefSch does not necessarily have to span trees anymore (but NRefSch is still required to do so).

For obvious reasons, all nodes used in the identification trees have an object type associated:

[T8] $\cup \mathsf{ran}(\mathsf{IdfNodes}) \subseteq \mathsf{dom}(\mathsf{Obj})$

The example grid tree and the identification trees should not be intermixed (except for the roots of the identification trees):

[T9] $\cup \mathsf{ran}(\mathsf{IdfNodes}) \cap (\mathsf{dom}(\mathsf{EOut}) \cup \mathsf{dom}(\mathsf{EIn})) = \varnothing$

The root nodes of the identification trees should indeed be part of the tree for the example grid:

[T10] $(\mathsf{dom}(\mathsf{IdfNodes}) - \cup \mathsf{ran}(\mathsf{IdfNodes})) \subseteq (\mathsf{dom}(\mathsf{EOut}) \cup \mathsf{dom}(\mathsf{EIn}))$

Finally, one good default rule is to automatically add simple identifications. So the following rule should be an invariant when manipulating the trees (e.g. when adding a simply identified object type):

[T11] $|\mathsf{RefSch}(\mathsf{Obj}(x))| = 1 \Rightarrow \mathsf{NRefSch}{\downarrow}x$

A tree for the example grid, in the context of an ORM schema, is now completely determined by the following five components:

$$\langle \mathbb{N}, \mathsf{EOut}, \mathsf{Obj}, \mathsf{Order}, \mathsf{NRefSch} \rangle$$

3 Negotiating the Size of the Example Space

As the title of this section suggests, in this section we concern ourselves with a negotiation mechanism to determine the number of examples that are to be used in the example grid. Such a negotation phase is needed because some object types may have a limited set of instances. In particular value types such as Gender which will generally have two or three instances only.

The first question we need to answer is the maximum number of instances that a given type may have. Due to relationships between types and the constraint patterns associated to the relationships; limiting the number of instances of a value type may propagate through the conceptual schema. As an example consider the schema shown in figure 10. Object type B has a maximum number of instances of 2. As each instance of object type A must play relationship f with a unique instance of B, there can be only

two instances of A. As a result, there can be only two instances of relationship f, so if f would be objectified this could lead to another propagation of a size limitation. In the context of larger schemas, these propagations may cause "ripple" effects through the entire schema. Note that if A is connected to B through a series of other relationship types, then it could even be the case that the maximum size of B needs to be reduced further!

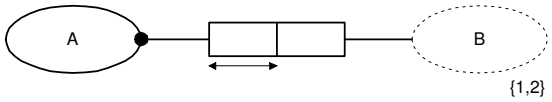

Figure 10: Propagating Maximum Size of Populations

Initially, object types are presumed to have a potentially infinite maximum population. For this purpose we need to introduce the notion of *infinity* as an explicit element in our calculations. Let ∞ denote infinity, then our population size calculations take place within the set: $\mathbb{N}^{\infty} \triangleq \mathbb{N} \cup \{\infty\}$. For $\mathbb{N}^{\infty}$ we inherit the $+$, $\times$ and $<$ operations from $\mathbb{N}$ with the following additional cases:

if $n \in \mathbb{N}$, then:

$$n \leq \infty$$
$$n + \infty = \infty + n = \infty$$
$$n \times \infty = \infty \times n = \infty$$

As the minimum $\mathsf{min}(n, m)$ of two natural numbers is defined in terms of $<$, we obviously have: $n \in \mathbb{N} \Rightarrow \mathsf{min}(n, \infty) = \mathsf{min}(\infty, n) = n$.

3.1 Generating Patterns

In determining the maximum sizes of populations, we need to generate for each relationship type the significant combinations of instances. As stated before, we consider a combination of instances to be significant if it shows all allowed combinations with respect to the cardinality constraints defined on that relationship type. The patterns are generated by the algorithm given below[2]. This algorithm takes as input the relationship type to be populated and the maximum sizes of the types (determined so far); in particular the players of the roles in the relationship type and the maximum size of the relationship type itself. The latter size is relevant in the case of objectified relationship types. Suppose in the example of figure 10, A is actually an objectified relationship type. In such a case we can only generate two instances for relationship type A. The algorithm itself is given as:

[2] I would like to thank L. Campbell for providing me with the first informal draft version of this algorithm.

GenPattern : $\mathcal{RL} \times (\mathcal{TP} \to \mathbb{N}^{\infty}) \to (\mathcal{TP} \rightarrowtail \mathbb{N}) \times \wp(\mathcal{RO} \to \mathbb{N})$

GenPattern$(Rel, Size) \triangleq$

VAR

Pattern:	$\wp(\mathcal{RO} \to \mathbb{N})$;
FreshTuple:	$\mathcal{RO} \to \mathbb{N}$;
WorkTuple:	$\mathcal{RO} \to \mathbb{N}$;
Used:	$\mathcal{TP} \rightarrowtail \mathbb{N}^{\infty}$;
p:	$\mathcal{RO}$;

MACROS

$Extendable(P : \wp(\mathcal{RO})) \equiv$

$\forall_{p \in P} [Used(\mathsf{Player}(p)) < Size(\mathsf{Player}(p))] \wedge$
$Used(\mathsf{Rel}(p)) + |P| \leq Size(\mathsf{Rel}(p))$;

$IncrUsed(p : \mathcal{RO}) \equiv$

BEGIN
$Used(\mathsf{Player}(p)) +:= 1$;
$Used(\mathsf{Rel}(p)) +:= 1$;
END;

BEGIN
# Initialise variables
Pattern := $\varnothing$;
Used(*Rel*) := 0;
FOR EACH $p \in \mathsf{Roles}(Rel)$ DO
Used(Player(p)) := 0;
END FOR;

WHILE $Extendable(\mathsf{Roles}(p))$ DO
# Generate fresh tuple
FOR EACH $p \in \mathsf{Roles}(Rel)$ DO
$IncrUsed(p)$;
$FreshTuple(p) := Used(\mathsf{Player}(p))$;
END FOR;

Probe uniqueness

$Pattern +:= \{FreshTuple\}$;
FOR EACH $p \in \mathsf{Roles}(Rel)$ DO
WorkTuple := *FreshTuple*;

# Try to mutate tuple
IF $\neg\exists_{\tau \subseteq \mathsf{Roles}(Rel)}\, [\mathsf{Unique}(\tau) \wedge p \notin \tau] \wedge Extendable(\{p\})$ THEN
$IncrUsed(p)$;
$WorkTuple(p) := Used(\mathsf{Player}(p))$;
$Pattern +:= \{WorkTuple\}$;
END IF;
END FOR;

# Generate nil tuple
FOR EACH $p \in \mathsf{Roles}(Rel)$ DO
$IncrUsed(p)$;
$FreshTuple(p) := 0$;
END FOR;

# Probe totality
FOR EACH $p \in \mathsf{Roles}(Rel)$ DO
WorkTuple := *FreshTuple*;

# Try to mutate tuple
IF $\neg\, \mathsf{Total}(p) \wedge Extendable(\{p\})$ THEN
$IncrUsed(p)$;
$WorkTuple(p) := Used(\mathsf{Player}(p))$;
$Pattern +:= \{WorkTuple\}$;
END IF;
END FOR;
END WHILE;

RETURN $\langle Used, Pattern \rangle$;
END.

3.2 Schema Plausibility Check

An interesting spin-off of the pattern generation algorithm is that when using this algorithm to determine the maximum size of all types in a conceptual schema, the result can be used to do a plausibility on the conceptual schema. If some type has a number maximum number of instances of 0, it is highly likely that there is an error in the constraint patterns of the conceptual schema. Since the pattern generation algorithm only uses a limited class of patterns, this kind of plausibility check may be a bit "oversensitive".

When determining the maximum sizes of all types in a conceptual schema, the results of the pattern generation algorithm can be used to refine, the current maximum sizes of the object types can be resized. This resizing is done by the algorithm below. It (re-)calculates the maximum sizes of the object types involved in each relationship type for which one of the involved types already has a maximum size that is not infinite.

$\mathsf{ReSize} : (\mathcal{TP} \to \mathbb{N}^\infty) \to (\mathcal{TP} \to \mathbb{N}^\infty)$
$\mathsf{ReSize}(Size) \triangleq$
VAR
Pattern: $\wp(\mathcal{RO} \to \mathbb{N})$;
Used: $\mathcal{TP} \rightarrowtail \mathbb{N}^\infty$;
ToDo: $\wp(\mathcal{RL})$;
x, r: $\mathcal{TP}$;
p: $\mathcal{RO}$;

BEGIN
# We should only take relationship types with at least one finite
# size for the types involved into consideration. Otherwise, GenPattern
# would not terminate.
$ToDo := \{\mathsf{Rel}(p) \mid Size(\mathsf{Player}(p)) \neq \infty \vee Size(\mathsf{Rel}(r)) \neq \infty\}$;
One way to optimise this further is to restrict the *ToDo* set
to those relationships for which one of the involved sizes has
changed during the last (or initial!) iteration.

# Recalculate maximum sizes
FOR EACH $r \in ToDo$ DO
$\langle Used, Pattern\rangle := \mathsf{GenPattern}(r, Size)$
FOR EACH $x \in \{\mathsf{Player}(p) \mid p \in \mathsf{Roles}(r)\} \cup \{r\}$ DO
$Size(x) := \mathsf{min}(Size(x), Used(x))$;
END FOR;
END FOR;

```
    # Propagate maximums in subtype hierarchy
    FOR EACH x, y ∈ OB DO
      IF x SpecOf y THEN
        Size(x) := min(Size(y), Size(x));
      END IF;
    END FOR;

    RETURN Size;
  END.
```

The idea is now to constantly call the above resize algorithm until the eventual maximum sizes have been stabalised. We do this by means of the following fixed-point calculation:

```
CalcSizes : →(TP → IN^∞)
CalcSizes() ≜
  VAR
    Size:     TP → IN^∞;
    NewSize:  TP → IN^∞;

  BEGIN
    NewSize := MaxSize;

    # Do fixed point calculation
    REPEAT
      Size := NewSize;
      NewSize := ReSize(Size);
    UNTIL NewSize = Size;
  END.
```

Note that this algorithm terminates since the number of type classes is finite and when resizing the maximum sizes we always choose the minimum value. In this last algorithm, MaxSize is the initial setting of the maximum sizes. This initial setting completely depends on the number of instances (or generatable) for the value types in the conceptual schema. In subsection 2.1 we introduced the function DomSize : $\mathcal{VL} \rightarrow \mathbb{N}$ as the function that provides these sizes. From this we can derive the initial maximum size for any object type as follows:

$$\mathsf{MaxSize} : \mathcal{TP} \to \mathbb{N}^{\infty}$$
$$\mathsf{MaxSize}(x) \triangleq \begin{cases} \mathsf{DomSize}(x) & \text{if } \exists_x [\mathsf{DomSize}{\downarrow}x] \\ \infty & \text{otherwise} \end{cases}$$

We are now in a position to interpret the results from CalcSizes. Let $\mathsf{Size}(x)$ be the size of type x resulting after CalcSizes. If $\mathsf{Size}(x) = 0$ for some type x, then it is highly likely that there is a problem with the constraint patterns. In such a case, each relationship type with a player y such that $\mathsf{Size}(y) = 0$ and a player z such that $\mathsf{Size}(z) > 0$ should be examined.

Furthermore, if there is a type x such that $\mathsf{Size}(x) < \mathsf{MaxSize}(x)$, there is a limited likelyhood that there is a problem with the constraint patterns. In this case, each relationship with a player y such that $\mathsf{Size}(y) < \mathsf{MaxSize}(y)$ and a player z such that $\mathsf{Size}(z) = \mathsf{MaxSize}(z)$, should be examined.

3.3 The Example Space

When generating examples for an example grid, we do this for each "umbrella" in the tree, i.e. a node and its direct descendants. The umbrella associated to a node n is defined formally by:

$$\mathsf{Umbrella} : \mathbb{N} \to \wp(\mathbb{N})$$
$$\mathsf{Umbrella}(n) \triangleq \pi_2\, \mathsf{EOut}(n) \cup \bigcup_{y \in \pi_2\, \mathsf{EOut}(n) \cap \mathbb{I}} \mathsf{Umbrella}(y)$$

From lemma 2.1 follows that an umbrella can be at most three nodes deep: the root, one layer of explicit or implicit nodes, and the layer containing the off-spring of the implicit nodes of layer 2. An example of two umbrellas is shown in figure 11. These two umbrellas correspond to two tables of the example grid shown in figure 1. When generating the examples for an entire tree, we only have to consider all umbrellas of the nodes which are not implicit, since the implicit nodes are already contained in these umbrellas.

The set of relationships involved in an umbrella can now simply be derived as follows:

$$\mathsf{RelSet} : \mathbb{N} \to \wp(\mathcal{RL})$$
$$\mathsf{RelSet}(n) \triangleq \left\{ \mathsf{Rel}(l) \;\middle|\; l \in \pi_1\, \mathsf{EOut}(n) \right\}$$

where Rel is presumed to be generalised to elements of $\mathcal{RP}$. Note that we only need to look at the set of outgoing edges from root of the umbrella as the outgoing edges from the nodes of the second layer are all implicit nodes, which allows us to apply lemma 2.2.

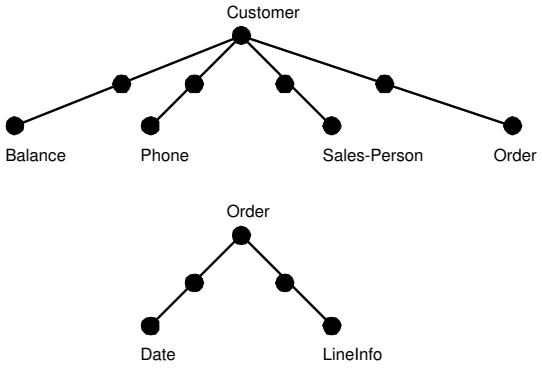

Figure 11: Two example umbrellas

A crucial first step in the generation of the examples is the negotiation of the number of instances in the root of each umbrella. In this negotiation we disregard the context of each umbrella, so we can not simply use the results of the CalcSizes function. The reason to ignore the context lies in the fact that when validating the constraint pattern of one relationship type, one does not want to have a negative influence on the number of examples by a (possible) problem in another part of the schema. In the definition, again an initial maximum size of type populations is needed. From this we can derive the maximum size for any object type using the following recursive function:

$$\mathsf{MaxSize} : \mathcal{OB} \rightarrow \mathbb{IN}$$

$$\mathsf{MaxSize}(x) \triangleq \begin{cases} \mathsf{DomSize}(x) & \text{if } x \in \mathcal{VL} \\ \Pi_{y \in \mathsf{IdfObjs}(x)}\, \mathsf{MaxSize}(y) & \text{otherwise} \end{cases}$$

Using this function the actual 'negotiation' function can be defined, determining the proper size of the root of an umbrella:

$$\mathsf{NodeSize} : \mathbb{N} \rightarrow \mathbb{IN}$$

$$\mathsf{NodeSize}(n) \triangleq \mathsf{min}(\mathit{MaxUserSizePref}, \mathsf{min}_{r \in \mathsf{RelSet}(n)}\ \mathit{Usage}(r, n))$$

where Usage is defined as:

$$\mathit{Usage} : \mathcal{RL} \times \mathbb{N} \rightarrow \mathbb{IN}$$

$$\mathit{Usage}(r, n) \triangleq \begin{cases} \mathit{Used}(\mathsf{Obj}(n)) & \text{if } \mathsf{MaxSize}(n) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

such that: $\langle \mathit{Used}, \mathit{Pattern} \rangle = \mathsf{GenPattern}(r, \mathsf{MaxSize})$

and ***MaxUserSizePref*** is a user defined constant providing the preferred maximum size of root nodes. Note: we presume $\mathsf{max}_{x \in X}\, f(x)$ returns 0 if $X = \varnothing$.

4 Filling an Example Grid

In this section we do the final step and actually populate the example grids. Thus far the only result from the GenPattern algorithm we used was the usage of instances. In this section, the patterns generated by GenPattern are finally utilised to generate the real examples.

4.1 Filling Object Types

When generating instances for the edges of an umbrella, we need to generate instances for the nodes (object types) first. Let $\mathsf{GenerateInst} : \mathcal{OB} \times \mathbb{N} \rightarrow \Omega$ be a given function that generates instances for object types. This function need the natural number to generate unique (and yet deterministic) instances.

When applied for a value type x, the result of $\mathsf{GenerateInst}(x, n)$ is one of the instances of the domain for values of x. Suppose that the user has given 8/7/94, 8/8/94, 8/9/94 as examples for the Date value, then $\mathsf{GenerateInst}(\mathsf{Date}, 2)$ would result in 8/8/94. When the $\mathsf{GenerateInst}(x, n)$ is used for a non-value type, a surrogate instance is generated. For instance, $\mathsf{GenerateInst}(\mathsf{LineInfo}, 1)$ could result in LineInfo1. The set Ω is used in this report as a general domain for simple instances from value types and composed instances from compositely identified types. Using this primitive function, the following more refined generation function can be defined, which also takes the required details of identifications (IdfNodes) into account:

$$\mathsf{GetInst} : \mathcal{OB} \times \mathbb{N} \rightarrow \Omega$$

$$\mathsf{GetInst}(x, n) \triangleq \begin{cases} nil & \text{if } n = 0 \\ \mathsf{Compose}(\mathsf{IdfNodes}(x), n-1) & \text{if } \mathsf{IdfNodes}{\downarrow}x \\ [\mathsf{GenerateInst}(x, n)] & \text{otherwise} \end{cases}$$

Note that the $n = 0$ case is a special case. Generating a nil value for $n = 0$ allows us to treat patterns resulting from optional roles uniform to the other patterns. The Compose function is used to construct the actual instance in the case of an object type with an identification provided by IdfNodes.

$$\mathsf{Compose} : \mathcal{OB}^{+} \times \mathbb{N} \rightarrow \Omega$$

$$\mathsf{Compose}([x_1, \ldots, x_n], m) \triangleq \begin{cases} [\mathsf{GetInst}(x_1, \gamma(m)+1)] & \text{if } n = 1 \\ [\mathsf{Compose}([x_1], \gamma(m)\ \mathsf{DIV}\ \mathsf{MaxSize}(x_1))] +\!\!+ & \\ \quad [\mathsf{Compose}([x_2, \ldots, x_n], \gamma(m)\ \mathsf{MOD}\ \mathsf{MaxSize}(x_1))] & \text{otherwise} \end{cases}$$

$$\text{where:} \quad \gamma(m) = \begin{cases} x/2 & \text{if IsEven(x)} \\ \Pi_{1 \leq i \leq n}\, \mathsf{MaxSize}(x_i) - (m+1)/2 & \text{otherwise} \end{cases}$$

Note that the use of the γ function causes a "natural" spread in the use of the examples.

4.2 Filling an Example Grid

For filling an umbrella with instances all that remains to be done is to call GenPattern for each relationship type contained in the umbrella with the negotiated nodesizes given in NodeSize, generate the concrete instances using GetInst, and finally order the result. All this is done by the GenPop algorithm. The algorithm takes the root node of the current umbrella as its input parameter and results in a population of the relationship types contained in the umbrella, together with an *ordered* population of the object types playing a role in these relationship types. The results can than be used (together with the earlier discussed Order for the nodes) to present the examples in an ordered way. The algorithm is defined as:

$\mathsf{GenPop} : \mathbb{N} \rightarrow (\mathcal{RL} \rightarrowtail \wp(\mathcal{RO} \rightarrowtail \Omega)) \times (\mathcal{OB} \rightarrowtail \Omega^+)$
$\mathsf{GenPop}(n) \triangleq$
VAR
$RPop$: $\mathcal{RL} \rightarrowtail \wp(\mathcal{RO} \rightarrowtail \Omega)$
$OPop$: $\mathcal{OB} \rightarrowtail \Omega^+$
Rel : $\mathcal{RL}$
$Tuple$: $\mathcal{RO} \rightarrowtail \Omega$
$NewTuple$: $\mathcal{RO} \rightarrowtail \mathbb{N}$
$Role$: $\mathcal{RO}$

BEGIN
FOR EACH $Rel \in \mathsf{RelSet}(n)$ DO
$\langle Used, Pattern \rangle := \mathsf{GenerateInst}(Rel, \mathsf{NodeSize})$;
FOR EACH $Tuple \in Pattern$ DO
# We presume that $\mathbb{N} \subseteq \Omega$
NewTuple := Tuple

FOR EACH $Role \in \mathsf{Roles}(Rel)$ DO
$NewTuple(Role) := \mathsf{GetInst}(\mathsf{Player}(Role), Tuple(Role))$;
$OPop(\mathsf{Player}(Role)) +\!\!+:= [NewTuple(Role))]$;
END FOR;
$RPop(Rel) +:= \{NewTuple\}$;
END FOR;
END FOR;

$OPop := \mathsf{ReOrder}(RPop, OPop, \mathsf{Obj}(n))$;
RETURN $\langle RPop, OPop \rangle$;

END.

The $\mathsf{ReOrder} : (\mathcal{RL} \rightarrowtail \wp(\mathcal{RO} \rightarrowtail \Omega)) \times (\mathcal{OB} \rightarrowtail \Omega^+) \times \mathcal{TP} \rightarrow (\mathcal{OB} \rightarrowtail \Omega^+)$ function is used to order the instances in the result. The order is based on the number of tuples in the relationship examples that use the instances. We do not provide an ordering algorithm for this function and leave that to the choice of the programmers. However, the following ordering condition must be met. If $P = \mathsf{ReOrder}(RPop, OPop, x)$, then:

$$\forall_{i,j \in \mathsf{dom}(P)} [\ i < j \Rightarrow |\mathsf{Tuples}(RPop, P[i], x)| \geq |\mathsf{Tuples}(RPop, P[j], x)|\ ]$$

where $\mathsf{Tuples}(RPop, v, x) = \{t \mid \exists_{p:\mathsf{Player}(p)=x} [t \in RPop(\mathsf{Rel}(p)) \wedge t(p) = v]\}$.

5 Conclusions

In this report we have presented an algorithm to generate examples for a selection of relationship types, such that the examples are significant (to a certain degree). These examples can be used to fill example grids, allowing users to validate cardinality constraints by looking at example patterns.

As a spin-off, we also discussed the option of calculating the maximum sizes of all types in a conceptual schema. This would allow us to detect possible problems with cardinality constraints.

The next step is to integrate these ideas into DBCreate and in a later stage InfoModeler itself. The definitions in this report are already suited for the ORM schemas as they can be specified in InfoModeler. They are not yet suited for the extended ORM concepts as introduced in [HP95]. However, the modification of the here presented algorithms to cover these extensions is not expected to be hard. asy

References

[BHW91] 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):471–495, October 1991.

[CH94] L.J. Campbell and T.A. Halpin. Abstraction Techniques for Conceptual Schemas. In R. Sacks-Davis, editor, *Proceedings of the 5th Australasian Database Conference*, volume 16, pages 374–388, Christchurch, New Zealand, January 1994. Global Publications Services.

[Hal95] T.A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[Har94] J. Harding. Examples View: Seeing is Understanding. Asymetrix Product Design, Database Division, Asymetrix Corp, Seattle, WA, 1994.

[Hof93] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[HP95] T.A. Halpin and H.A. Proper. Subtyping and Polymorphism in Object-Role Modelling. *Data & Knowledge Engineering*, 15:251–281, 1995.

[Pro94a] H.A. Proper. Interactive query formulation using spider queries. Asymetrix Research Report 94-2, Asymetrix Research Laboratory, University of Queensland, Brisbane, Australia, 1994.

[Pro94b] H.A. Proper. Introduction to formal notations. Asymetrix Research Report 94-0, Asymetrix Research Laboratory, University of Queensland, Brisbane, Australia, 1994.

[WHB92] Th.P. van der Weide, A.H.M. ter Hofstede, and P. van Bommel. Uniquest: Determining the Semantics of Complex Uniqueness Constraints. *The Computer Journal*, 35(2):148–156, April 1992.