A Flexible Framework for a Correct Database Design

Donatella Castelli, Serena Pisani

Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche
Via S. Maria, 46 Pisa, Italy
{castelli,serena}@iei.pi.cnr.it

Abstract. This paper presents a flexible database schema transformational framework. Flexibility is achieved by adopting a generic model for describing database schemas and a transformational language able to represent all the correctness preserving schema transformations. This framework, originally defined for schema design, is also applicable for supporting other activities related to the database life-cycle. As an illustrative example, this paper shows how it can be used to support a database reverse engineering process.

1 Introduction

The correctness of a database design is often obtained by fixing the set of schema transformational operators that can be used for carrying out the design and by associating a set of applicability conditions to each of these operators. These conditions must be checked when the operators are applied. If they are satisfied, then the design step is guaranteed to be correct. A drawback of this solution is that the set of schema transformational operators is often too rigid: the operators work for particular models and they can only execute particular refinement steps.

This paper proposes a correctness preserving schema transformational framework that, unlike to the previous proposals [2,3,4,5,12], is also adaptable to different situations. This framework relies on a database schema model, called *Database Schema* (DBS) [6] and a design language, called *Schema Refinement Language* (SRL). SRL consists of a set of primitives (with associated the set of their applicability conditions) for transforming DBS schemas and a composition operator. A rule is given for deriving the applicability conditions of any transformations from the applicability conditions of its component transformations. The composition operator renders the language complete, i.e., able to express all the schema transformations.

The genericity of the model employed and the completeness of the transformational language render the framework suitable for different kinds of database applications. This paper shows how the SRL framework can be exploited to point out incorrect database reverse engineering processes.

The next three sections of this paper introduce the design framework. In particular, Section 2 presents DBS and the primitive operators of SRL. Section

3 introduces the composition operator. Section 4 introduces the rule for deriving the applicability conditions of a composed schema transformation. This rule is given in detail in the Appendix. Section 5 shows how the framework introduced can be exploited in a database reverse engineering process. Finally, Section 6 contains concluding remarks.

2 Schema Refinement Language

The Schema Refinement Language (SRL) assumes that the whole design relies on a single notation which is sufficiently general to represent semantic and object models. This notation, illustrated briefly through the example in Figure 1[1], allows to model the database structure and behavior into a single module, called *Database Schema* (DBS). This module encloses classes, attributes, is-a relations, integrity constraints and operations specifications. A graphical representation of the schema *StoneDB* is given in Figure 4(a).

database schema *StoneDB*
class *exterior* **of** *MARBLE* **with** (*e_name:NAME; e_tech_char:TECH_CHAR*)
class *interior* **of** *MARBLE* **with** (*i_name:NAME; i_tech_char: TECH_CHAR*)
constraints
  $\forall\ m \cdot m \in (interior \cup exterior) \Rightarrow \exists\ n \cdot n = (e\_name \cup i\_name)(m)$
initialisation
  exterior, interior, e_tech_char, i_tech_char, e_name, i_name := **empty**
operations
  assign_tech_char(n,t) = **pre** $n \in \mathsf{ran}(e\_name \cup i\_name) \wedge t \in TECH\_CHAR$
   then $i\_tech\_char := i\_tech\_char \mathbin{<\!\!\!+} \{(x,t) \mid i\_name(x)=n\} \parallel$
     $e\_tech\_char := e\_tech\_char \mathbin{<\!\!\!+} \{(x,t) \mid e\_name(x)=n\}$

Fig. 1. A Database Schema

The notation of Database Schema is formalised in terms of a formal model introduced within the B-Method [1]. This formalisation allows to exploit the B theory and tools for proving expected properties of the DBS schemas.

The SRL primitive operators that transforms DBS schemas are given in Table 1. The equality conditions that appear as a parameter in the add/rem transformations specify how the new/removed element can be derived from the already existing/remaining ones. These conditions are required since only redundant components can be added and removed in a refinement step. The language does not permit to add or remove schema operations. It only permits to change the way in which an operation is defined. Note that the operation definitions are also automatically modified as a side effect of the transformations that add and remove schema components. In particular, these automatic modifications add appropriate updates for each of the new schema components, cancel the occurrences of the removed components and apply the proper variable substitutions.

[1] $\mathbin{<\!\!\!+}$ stands for the overriding between relations; $\parallel$ represents the parallel assignment.

Table 1. SRL language

add.class (*class.name*, *class.name =expr*)
rem.class (*class.name*, *class.name =expr*)
add.attr(*attr.name*, *class.name*, *attr.name =expr*)
rem.attr (*attr.name*, *class.name*, *attr.name =expr*)
add.isa(*class.name1*, *class.name2*)
rem.isa (*class.name1*, *class.name2*)
mod.op (*op.name*, *body*)

A transformation can be applied when its *applicability conditions* are verified. These are sufficient conditions, to be verified before the execution of the transformation, that prevent from applying meaningless and correctness breaking schema design. The criterion for the correctness of schema design is based on the following definition (for a formal definition see [7]):

Definition (DBS schema refinement relation) A DBS schema S_1 refines a DBS schema S_2 if:
(a) S_1 and S_2 have the same signature;
(b) there exists a 1:1 correspondence between the states modelled by S_1 and S_2;
(c) the database B_1 and B_2, modelled by S_1 and S_2, when initialised and submitted to the same sequence of updates, are such that each possible query on B_1 returns one of the results expected by evaluating the same query on B_2. □

The applicability conditions consist of the conjunction of simple conditions. In that follows, these conditions will be called *applicability predicates*.

Let us outline that the main objective in defining this framework has been the flexibility. In order to fulfill this objective the model and the schema refinement language have been provided with very primitive mechanisms.

SRL, as presented above, is not still suitable enough to be used in the real applications. Indeed, the applied schema transformations are usually more complex of those listed above. In order to overcome this limitation, a composition operator for SRL is introduced in the next section.

3 Composition Operator

The composition operator permits to define complex transformations from simpler one. The following preliminary definition is needed before introducing the composition operator (the corresponding formal definition is given in [7]).

Definition (Consistent operation modification) A set of SRL schema transformations specifies *consistent operation modifications* if, for each operation that is modified by more than one transformation, the replacing bodies can be totally ordered with respect to the refinement relation. □

Intuitively, this definition means that all the bodies that are specified for the same operations by different transformations must describe the same general behaviour. They can only differ for being more or less refined.

The SRL composition operator can be now defined as follows. **Definition (Composition operator "$\circ$")** Let t_1, t_2, ..., t_n be a set of SRL schema transformations that specify consistent operation modifications. Let *<Cl,Attr,IsA,Constr,Op>* be a DBS schema where: *Cl*, *Attr*, *IsA*, *Constr* and *Op* are, respectively, the set of classes, attributes, is-a relationships, integrity constraints and schema operations. *Op* always contains an operation *Init* that specifies the schema initialisation. The SRL schema transformation composition operator is defined as follows:

$$t_1 \circ t_2 \circ \ldots \circ t_n \ (<Cl,Attr,IsA,Constr,Op>)=$$
$$<Cl{\cup}ACl\text{-}RCl, Attr{\cup}AAttr\text{-}RAttr, IsA{\cup}AIsA\text{-}RIsA,$$
$$/\text{RemSubst*}/(Constr{\wedge}AConstr),[\text{RemSubst*}/Op'>$$

where *ACl/RCl*, *AAttr/RAttr* and *AIsA/RIsA* are sets formed, respectively, by the set of classes, attributes and is-a relationships that are added/removed by t_1, t_2, ..., t_n. *RemSubst** is the transitive closure of the variable substitutions *x:=E* dictated by the conditions that are specified when an element is removed. If we have, for example, *rem.class(c, c=E)* $\circ$ *rem.class(d, d=f(c))* $\circ$ *rem.class(e, e=F)* then *RemSubst** is the parallel composition of the substitutions *c:=E*, *d:=f(E)* and *e:=F*. [*RemSubst**]X is the expression that is obtained by applying the substitution *RemSubst** to X. For example, [*x:=E*]*R(x)* is *R(E)*. This substitution permits to rephrase integrity constraints and operation definitions by removing the cancelled schema components. *AConstr* are the conjunction of the inherent constraints associated with the new schema components and the conditions that specify how an added element relates to the remaining ones. Finally, Op' is the new set of operation definitions. These result from the modifications that are required explicitly and from the automatic adjustments caused by the addition and removal of schema components. When more than one of the component transformations modifies an operation, the more specialised behavior is selected.□

SRL is a complete DBS schema refinement language. This property ensures that SRL is powerful enough to express every DBS schema transformation. The following example illustrates how new transformations can be build.

Example. Let us define the transformation illustrated graphically in Figure 2. This transformation adds a new class, *C*, as superclass of *n* already existing classes, C_1, $\cdots$, C_n, and moves a set of attributes shared with the subclasses to the new class. The designer can built this transformation as composition of simple SRL transformations[2]:

$$add.superclass(C,\ (C_1,\ \cdots,\ C_n),\ \{((a_1,\ \cdots,\ a_n),\ a),\cdots,\ ((b_1,\ \cdots,\ b_n),\ b)\}) =$$
$$add.class(C,\ C{=}C_1 \cup \cdots \cup\ C_n) \circ add.isa(C_1,\ C) \circ \cdots \circ add.isa(C_n,\ C) \circ$$
$$add.attr(a,\ C,\ a{=}a_1 \cup \cdots \cup\ a_n) \circ$$
$$rem.attr(a_1,\ C_1,\ a_1{=}C_1{\lhd}\ a) \circ \cdots \circ rem.attr(a_n,\ C_n,\ a_n{=}C_n{\lhd}\ a)) \circ \cdots \circ$$
$$add.attr(b,\ C,\ b{=}b_1 \cup \cdots \cup\ b_n) \circ$$

[2] $\lhd$ indicates the domain restriction.

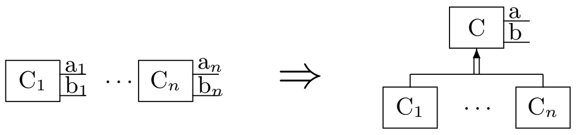

Fig. 2. add_superclass

$$rem.attr(b_1,\ C_1,\ b_1{=}C_1\lhd\ b) \circ\cdots\circ\ rem.attr(b_n,\ C_n,\ b_n{=}C_n\lhd\ b))$$

The transformation *add.superclass* can be used as any other SRL transformation. For example, it can be applied to the database schema *StoneDB* of Figure 1, generating the DBS schema of Figure 3, as follows:

$$add.superclass(marble,(exterior,interior),\{((e\_name,i\_name),name)\})(StoneDB)$$

The change brought to the static part of the schema are shown in Figure 4.

database schema *StoneDB_1*
class *marble* **of** *MARBLE* **with** (*name:NAME*)
class *exterior* **is-a** *marble* **with**(*e_tech_char:TECH_CHAR*)
class *interior* **is-a** *marble* **with**(*i_tech_char:TECH_CHAR*)
constraints
$\forall\ m \cdot m{\in}(interior{\cup}exterior) \Rightarrow \exists\ n \cdot n{=}name(m)$
initialisation
marble, name, exterior, interior, e_tech_char, i_tech_char := **empty**
operations
assign_tech_char(n,t) = **pre** $n{\in}\mathsf{ran}(\ name) \wedge t{\in}TECH\_CHAR$
then $i\_tech\_char := i\_tech\_char \lhd\!\!\!+ \ \{(x,t) \mid (name{\lhd}interior)(x){=}n\}\ \|$
$e\_tech\_char := e\_tech\_char \lhd\!\!\!+ \ \{(x,t) \mid (name{\lhd}exterior)(x){=}n\}$

Fig. 3. StoneDB_1

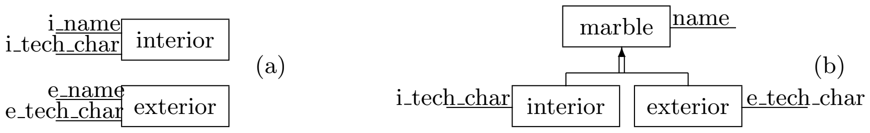

Fig. 4. *StoneDB* schema

The next section presents the mechanisms that allow to dynamically associate, to each of the new defined transformations, its applicability conditions.

4 Applicability Conditions

The applicability conditions of a composed transformation are sufficient conditions for ensuring that the application of the transformation to a schema results in a correct design. These conditions are generated constructively by an algorithm, called *Applicability Condition Generating Algorithm* (ACGA), given in Appendix. ACGA generates the applicability conditions by considering the schema structure and the changes brought by the component transformations.

As far as the applicability conditions of composed transformations, the following property holds [7]:

Property (SRL is a refinement language) Let t_1, t_2, ..., t_n be SRL schema transformations and S be a DBS schema. The application of the transformation $t_1 \circ t_2 \circ \ldots \circ t_n(S)$, when its applicability conditions are verified, produces a refinement of S.□

This property ensures the correctness of any database SRL design.

By reasoning on the applicability conditions generated by the ACGA algorithm, it turns out that some of them can be solved without instantiating the parameters; others can be discharged by simply comparing the values of the parameters. This suggest us to automatically prune these predicates and associate to the instance of a transformation only the simplified set of applicability predicates. The pruning is done at different stages. When the transformation is defined, with parameters $p_1, \ldots, p_n$, the set of applicability predicates is scanned and, for each predicate P_{ij} of the set, the proof of $\forall p_1, \ldots, p_n, S \cdot P_{ij}$ is attempted, where S is a DBS schema. If the proof is successful, P_{ij} is inserted in the set of the applicability predicates that have not to be proved anymore. The second kind of pruning, is executed when the transformation is instantiated. By reasoning on the structure of any transformations and the values of the parameters, several applicability predicates are discharged. The ACGA algorithm, reported in the Appendix, actually implements a mix between the generation of the applicability conditions and the second pruning. The result of the second pruning is the set of applicability predicates that the designer has to prove for a particular application of the transformation. Notice that no much effort is usually required for this proof. This is because the set of applicability predicates returned by the ACGA algorithm is often very small. Moreover, since the SRL framework and its application conditions are formalised, an automatic, or at least guided, discharge of the applicability conditions that are generated is possible.

Let us see an example of dynamic generation of the applicability conditions of a composed transformation. Table 2 lists the applicability conditions of the transformation *add.superclass*, as invoked in the example of Section 3[3].

Those above are the only applicability conditions that are returned to the designer. The others are checked and discharged automatically either when the *add.superclass* is defined or when the transformation is applied.

[3] *NewConstr* stands for the constraints generated by the ACGA algorithm for each condition to be proved; C_2 is-a-reach C_1 is a predicate that indicates, if verified, the existence of an is-a path between C_1 and C_2.

Table 2. Applicability conditions of the add.superclass

$NewConstr \Rightarrow \neg$(marble $\mathsf{is\text{-}a\text{-}reach}$ exterior)
$NewConstr \Rightarrow \neg$(marble $\mathsf{is\text{-}a\text{-}reach}$ interior)
$NewConstr \Rightarrow \mathsf{dom}$(e_name$\cup$i_name) $\subseteq$ marble
$NewConstr \Rightarrow$ e_name=(exterior$\lhd$name)
$NewConstr \Rightarrow$ i_name=(interior$\lhd$name)

5 Exploiting SRL in a Reverse Engineering Process

The genericity of the employed model, the completeness of the SRL language and the definition of the ACGA algorithm render the transformational framework a general instrument for supporting a correct database design. The designer can thus build a personalised set of transformations and use them as primitive operators. This can be done without giving up the support for correctness.

The above three characteristics provides also the ground for a wider use of the framework. As an example of possible use, below it is illustrated how the presented framework can be profitably employed for supporting a reverse engineering process.

Several approaches, either systematic or informal, are available for carrying out database reverse engineering processes [10,14]. They permit to produce a plausible high level model of an existing logical schema by employing different heuristics and rules. These approaches exhibit different levels of automatic treatments. Generally, the more automatic they are, the more limitations they impose on the original form of the logical schema. Actually, most authors consider impracticable a rigid compiling approach. They recommend, instead, informal approaches driven by frequent interactions with the designer that is responsible of taking semantic decisions. These informal approaches, if more applicable in practice, are, however, unable of ensuring that the semantic interpretation embedded in the reverse engineering process is correct. This limitation regards both the static part of the schema, and, more remarkably, the behavioural part.

In what follow we illustrate, by an example, how the schema transformational framework, that has been presented in the previous sections, can be used to overcome the above limitation.

Consider the schema depicted in Figure 5(a). Imagine that this is the logical schema of a particular database and that the following operation is associated to this schema.

$$\begin{aligned}\textit{assign\_tech\_char(n,t)} = \ & \textbf{if } n{\in}\mathsf{ran}\textit{(e\_name)}\\ & \textbf{then } \textit{e\_tech\_char:=e\_tech\_char} \lhd\!\!\!+ \{\textit{(x,t)}\mid \textit{e\_name(x)=n}\}\end{aligned}$$

Imagine, now, that a reverse engineering approach is applied to this schema and that the conceptual schema returned after this application is that given in Figure 5(b).

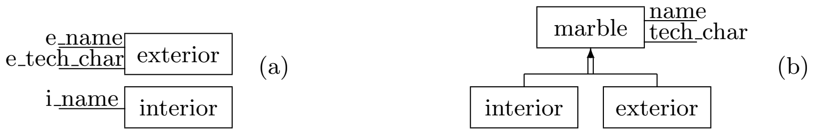

Fig. 5. Reverse engineering

By observing both the data and the operations during the reverse engineering process the existence of a superclass of *exterior* and *interior* has been extrapolated. The attributes of these two classes have been moved to the superclass[4].

We now employ the SRL transformational framework to verify the correctness of the reverse engineering process. To do this, we move forward from the reverse process, i.e., we derive from the two schemas a transformation that, when applied to the schema in Figure 5(b), produces the logical schema and the operation defined on it. If such transformation can be safely applied, then we can conclude that the logical schema is a correct implementation of the conceptual schema, and, vice versa, that the chosen conceptual schema is a correct specification of the logical schema. This transformation can be easily defined by considering the difference between the set of components in the two schemas and the assumptions on the semantics of the logical schema that have driven the reverse engineering process:

$$\begin{array}{l}
\text{add.attr}(i\_name,\ interior,\ i\_name{=}name{\lhd}interior) \circ \\
\text{add.attr}(e\_name,\ exterior,\ e\_name{=}name{\lhd}exterior) \circ \\
\text{rem.attr}(name,\ marble,\ name{=}(e\_name{\cup}i\_name)) \circ \\
\text{add.attr}(e\_tech\_char, exterior,\ e\_tech\_char{=}tech\_char) \circ \\
\text{rem.attr}(tech\_char,\ marble,\ tech\_char{=}e\_tech\_char) \circ \\
\text{rem.isa}(interior,\ marble) \circ \text{rem.isa}(exterior,\ marble) \circ \\
\text{rem.class}(marble,\ marble{=}(interior{\cup}exterior)) \circ \\
\text{mod.op}\ (assign\_tech\_char,\ \textbf{if}\ n \in \mathsf{ran}\, e\_(name)\ \textbf{then} \\
\qquad tech\_char := tech\_char \mathrel{\lhd\!\!+} \{(x,t) |\ e\_name(x){=}n\})
\end{array}$$

Notice that assumptions on the semantics of the application have been used to define the relations between the two schemas. For example, the above transformation specifies that the new class *marble* is defined as the union of the classes *interior* and *exterior*.

The applicability conditions of this transformation are obtained by applying the ACGA algorithm[5]:

[4] In this example, we make the assumption that the both the logical schema and the conceptual schema are given as DBS schemas. As outlined above, this not a too restrictive assumption since, in case other models are used, then these can be easily mapped in terms of the DBS model.

[5] *NewConstr* is the conjunction of all the initial schema constraints and of all those added by the composed transformation; a attribute-of C stands for the inherent constraint "a is an attribute of C".

1. $(NewConstr\text{-}\{i\_name\ \mathsf{attribute\text{-}of}\ interior\}) \Rightarrow \mathsf{dom}(name \lhd interior) \subseteq interior$
2. $(NewConstr\text{-}\{e\_name\ \mathsf{attribute\text{-}of}\ exterior\}) \Rightarrow \mathsf{dom}(name \lhd exterior) \subseteq exterior$
3. $NewConstr \Rightarrow name = e\_name \cup i\_name$
4. $(NewConstr\text{-}\{e\_tech\_char\ \mathsf{attribute\text{-}of}\ exterior\}) \Rightarrow \mathsf{dom}(tech\_char) \subseteq exterior$
5. $NewConstr \Rightarrow tech\_char = e\_tech\_char$
6. $NewConstr \Rightarrow marble = interior \cup for\_exterior\_m$
7. **pre** $n \in \mathsf{ran}(name) \wedge t \in TECH\_CHAR$ **then**
   $tech\_char := tech\_char \mathbin{\lhd\!\!\!+} \{(x,t) | name(x)=n\}$
   $\sqsubseteq$
   if $n \in \mathsf{ran}(e\_name)$ **then** $e\_tech\_char := e\_tech\_char \mathbin{\lhd\!\!\!+} \{(x,t) | e\_name(x)=n\}$

If we try to prove these conditions, we discover that the conditions 4 and 7 are false. The condition 4 is false since the domain of the attribute *tech_char* is greater than the set described by the class *exterior*. The condition 7 is false since the higher level operation specifies state transformations that are not implemented by the lower level one. In particular, the overriding set in the former operation can contain a pair whose first element is any marble object, whereas, in the latter operation, the first element of the same pair can only be an object of the class *exterior*. By reasoning on the failed proofs, we can try to repair the mistake originated by the reverse engineering process. In this case, for example, the two false conditions suggest to restrict the domain of the attribute *tech_char*. This can be done by introducing in the conceptual schema the constraint $\mathsf{dom}(tech\_char) = exterior$.

As a consequence, the two applicability predicates that were false on the previous conceptual schema are now trivially verified.

6 Conclusions

This paper has presented a framework for supporting a correct database design. Differently form other proposals, this framework is not based on a fixed set of schema transformational operators stated at priori, but these operators can be built dynamically according the designer needs. This characteristic and the availability of a model that can be used to interpret several other database models, are the elements that mainly contribute to the flexibility of the framework. The paper has shown an example of how this framework can be exploited, not only for the design but also for supporting different stages of the database life-cycle. Other significant uses are possible [8,9]. We have, for example, experimented this framework for supporting the maintenance of the multimedia database for the MIAOW system [13]. This database, designed as part of the Marble Industry Advertising over the World ESPRIT Project (n. 3990), maintains information about stones and stone actors. The original design of this database consisted of a sequence of OMT-like schemas [15]. Each schema in the sequence is generated by ad-hoc transformations. The final schema can be directly mapped into an Illustra schema [11]. We first interpreted the original design of this database in terms of our framework. This permitted us to verify the correctness of the original design.

Then, by interpreting the changes to the conceptual schema, originated by the change of the requirements, in terms of particular schema transformations, we were able to predict, for each change, the minimal changes to be operated on the schemas that documented the design of the MIAOW database. This experience suggested us improvements to our framework and confirmed its versatility.

References

1. J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. P. Assenova and P. Johannssen. Improving Quality in Conceptual Modelling by the Use of Schema Transformation. *Lecture Notes in Computer Science*, n.1157, pp.277-291, Springer-Verlag, 1996.
3. C. Batini, G. Di Battista and G. Santucci. Structuring Primitives for a Dictionary of Entity Relationship Data Schemas. *IEEE Transactions on Software Engineering*, 19(4), April 1993.
4. P. Van Bommel. Database design by computer-aided schema transformations. *Software Engineering Journal*, pp.125-132, July 1995.
5. P. Mc. Brien and A. Poulovassilis. A Formal Framework for ER Schema Transformation. *Lecture Notes in Computer Science*, n.1331, pp.408-421, Springer-Verlag, 1997.
6. D. Castelli and E. Locuratolo. ASSO: A Formal Database Design Methodology. *Information Modelling and Knowledge Bases VI*, H. Jaakkaola et al.eds., IOS-Press, 1995.
7. D. Castelli and S. Pisani. *A Transformational Approach to Database Design.* IEI-CNR Technical Report, 1998.
8. D. Castelli and S. Pisani. Ensuring Correctness of personalised schema refinement transformations. *Proc. International Workshop on Verification, Validation and Integrity Issue in Expert and Database Systems*, 1998, to appear.
9. D. Castelli. A strategy for Reducing the Effort for Database Schema Maintenance. *Proc. Second Euromicro Conf. on Software Maintenance and Reengineering*, pp.29-35, Florence, 1998.
10. S. Ghannouchi, H. Ghezala and F. Kamoun. A Generic Approach for Data Reverse Engineering taking into Account Application Domain Knowledge. *Proc. Second Euromicro Conf. on Software Maintenance and Reengineering*, pp.21-28, Florence, 1998.
11. Illustra Server Release 3.2, 1995.
12. K.J. Lieberherr, W.L. Hürsch and C. Xiao. Object-Extending Class Transformations. *Formal Aspects of Computing*, 6, pp.391-416, 1994.
13. *MIAOW Multimedia Database: Revised Design and Implementation.* MIAOW-CNR-REP-001-007. 1996.
14. W. J. Premerlani and M. R. Blaha. An Approach for Reverse Engineering of Relational Databases. *Communication of the ACM*, 37(5), pp.42-49, 1994.
15. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

Appendix: The Applicability Condition Generating Algorithm

The *Applicability Condition Generating Algorithm* (ACGA) takes as input a set of SRL transformations $\{t_1, t_2, \ldots, t_n\}$, their applicability conditions, the set *ver_appl*, specified below, and a DBS schema *<Cl,Attr,IsA,Constr,Op>*. It returns a set of applicability predicates. The description of the algorithm is given using an informal notation and makes use of the following abbreviations:

- *ACl/Attr/IsA*, *RCl/Attr/IsA*, *AConstr* and *RemSubst** are described in Definition 3;
- *Attr*(C), *AAttr*(C) and *RAttr*(C) indicate the set of attributes that are , respectively, defined on the class C in the initial schema, added to C and removed from C by the composed transformation;
- C_1 is-a C_2 stands for the inherent constraint "C_1 is a subclass of C_2";
- *ver_appl* is the set of applicability conditions that are proved to be verified when the composed transformation is defined.

The algorithm consists of four steps. For sake of brevity, only the step 4 is reported explicitly. The first step initialises the set $appl_\circ$ which maintains the applicability predicates returned by the algorithm. The second step generates a first group of applicability predicates. These require that the component transformations specify consistent modifications. The third step generates a temporary set *appl* of applicability predicates to be proved. This set is scanned in the fourth step. If a predicated of this set is found to be false, then $appl_\circ$ is set to "false" and the algorithm is terminated. Each predicate of the set that cannot be discharged by the checks operated by the algorithm it is inserted in the set $appl_\circ$.

Step 4 of the Applicability Condition Generating Algorithm

```
appl:=appl-ver_appl;
repeat
  p:=extract(appl); appl:=appl-p;
  case type(p) of $x \in Cl$ then if not($x \in (ACl \cup Cl)$) then $appl_\circ$:=false
% This predicate is false if x is not in the initial schema and there is no transformation
in the composition that adds the class x. It is true otherwise.
  or $x \notin Cl$ then if not($x \notin Cl$) then $appl_\circ$:=false
% This predicate is false if the class x is in the initial schema. It is true otherwise.
  or $x \in Attr(C)$ then if not($x \in Attr(C)$) then $appl_\circ$:=false
% This predicate is false if the x is not an attribute of C in the initial schema. It is
true otherwise.
  or $x \notin Attr$ then if not($x \notin Attr$) then $appl_\circ$:=false
% This condition is false if the attribute x is in the initial schema. It is true otherwise.
  or $x \in IsA$ then if not($x \in IsA$) then $appl_\circ$:=false
% This predicate is false if the is-a relationship x is not in the initial schema. It is true
otherwise.
  or $x \notin IsA$ then if not($x \notin IsA$) then $appl_\circ$:=false
% This predicate is false if the is-a relationship x is in the initial schema. It is true
otherwise.
  or Free(E)$\subseteq (Cl \cup Attr)$
```

```
      then if not(Free(E)-$(ACl \cup AAttr)) \subseteq (Cl \cup Attr)$ then $appl_\circ$:=false
% The predicate is false if the free variables in E are not added variables or they do
not belong to the initial schema. It is true otherwise.
   or $x \notin$Free(E) then if not($x \notin$Free(E)) then $appl_\circ$:=false
% The predicate is false if x is a free variable of E. It is true otherwise.
   or $\neg\exists C \in Cl \cdot (C,C_1) \in IsA \vee (C_1,C) \in IsA$
      then if $C_1 \in (Cl \cap ACl \cap RCl)$ then do nothing
             else if $(\exists C \in (Cl \cup ACl) \cdot ((C_1,C) \in (AIsA\text{-}RIsA) \vee (C,C_1) \in (AIsA\text{-}RIsA))) \vee$
                     $(\exists C \in Cl \cdot ((C,C_1) \in (IsA\text{-}RIsA) \vee (C_1,C) \in (IsA\text{-}RIsA))) \vee$
                     $(\exists C \in Cl \cdot ((C,C_1) \in (IsA \cap AIsA \cap RIsA) \vee (C_1,C) \in (IsA \cap AIsA \cap RIsA)))$
                  then $appl_\circ$:=false
% The predicate has not to be checked if the class $C_1$, that is removed, belongs to the
initial schema and it is added by a transformation in the composition. It is false, if
there are not removed is-a relationships that involve $C_1$.
   or $\neg\exists a \in Attr(C_1)$ then if $C_1 \in (Cl \cap ACl \cap RCl)$ then do nothing
        else if $(\exists a \in (AAttr(C_1)\text{-}RAttr(C_1))) \vee$
                $(\exists a \in (Attr(C_1) \cap AAttr(C_1) \cap RAttr(C_1))) \vee$
                $(\exists a \in (Attr(C_1)\text{-}RAttr(C_1)))$ then $appl_\circ$:=false
% The predicate has not to be checked if the class $C_1$, that is removed, belongs to
the initial schema and it is added by a transformation in the composition. It is false if
there are not removed attributes defined on $C_1$.
   or $Constr \Rightarrow C_2 \subseteq C_1$
        then if $\exists C_3,\cdots,C_n \in (Cl \cup ACl) \cdot (C_1 = C_2 \cup C_3 \cup \cdots \cup C_n) \in AConstr$
                then do nothing
                else $appl_\circ := appl_\circ \cup \{(Constr \wedge (AConstr\text{-}\{C_2 \text{ is-a } C_1\})) \Rightarrow C_2 \subseteq C_1\}$;
% The predicate is true if there is a transformation in the composition that adds the
class $C_1$ and defines it as the union $C_2$ and other classes.
   or $Constr \Rightarrow \neg(C_1 \text{ is-a-reach } C_2)$
        then $appl_\circ := appl_\circ \cup \{(Constr \wedge AConstr) \Rightarrow \neg(C_1 \text{ is-a-reach } C_2)\}$;
   or $Constr \Rightarrow x=E$ then if $x=E \in AConstr$ then do nothing
                        else $appl_\circ := appl_\circ \cup \{(Constr \wedge AConstr) \Rightarrow x=E\}$;
   or $Constr \Rightarrow \text{dom}(F) \subseteq C$
      then $appl_\circ := appl_\circ \cup \{(Constr \wedge (AConstr\text{-}\{a \text{ attribute-of } C \mid \exists a \cdot$
             $a=F \in AConstr\})) \Rightarrow \text{dom}(F) \subseteq C\}$;
   or $body_1 \sqsubseteq body_2$
      then $appl_\circ := appl_\circ \cup \{[RemSubst^*]body_1 \sqsubseteq [RemSubst^*]body_2\}$
% The variable substitutions must be taken into account when evaluating the refine-
ment relation.
until $appl = \emptyset \vee appl_\circ$=false;
return $appl_\circ$;
```