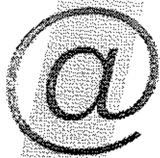
Consiglio Nazionale delle Ricerche

Improving Reliability in the Database Designing Frameworks

Donatella Castelli, Serena Pisani

IEI: B4-11-05-98



Improving reliability in the database design frameworks

Donatella Castelli and Serena Pisani
Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche
Via S. Maria, 46
Pisa, Italy
e-mail: {castelli,serena}@iei.pi.cnr.it
fax: +39 50 554342

Abstract

This paper extends a database schema transformation language, called Schema Refinement Language, with a composition operator and a rule for deriving the conditions under which a composed transformations is guaranteed to produce a correct design. The framework that results from this extension can be exploited for improving the reliability of the database schema design also when other design frameworks are used.

Categories and Subject Descriptions: D.2.4 [**Software Engineering**]: Software/Program Verification - *Correctness proofs*; D.2.4 [**Software Engineering**]: Software/Program Verification - *Reliability*; D.2.4 [**Software Engineering**]: Software/Program Verification - *Validation*; H.2.1 [**Database Management**]: Logical Design - *Data models*

1 Introduction

The reliability of a schema design is usually obtained by reducing the set of the operators that can be used to carry out the design to a fixed set [1, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14]. Only the given operators can be used to carry out the design. Each operator is provided with the conditions under which it is guaranteed to produce a correct design. Usually, these are conditions that can be proved by simply checking the schema structure. A drawback of these proposals is that the set of given operators are often insufficient to cover the specific needs that occur in everyday practice. For some applications, the set of chosen transformations may be too low-level, whereas, for others, this set may be too specialised.

This paper proposes an approach for supporting a correct design which overcomes the above drawback. In this case, the transformational operators can be built dynamically following the designers' needs.

The approach proposed relies upon a design language called Schema Refinement Language (SRL) [15]. This language consists of a set of primitives for schema transformations. Each primitive has its applicability conditions associated with it. The proof of these conditions ensures the correctness of the design step. The paper proposes a composition operator for this language that can be used to express all the schema transformations. Moreover, it introduces a rule for deriving the applicability conditions of a composed transformation from the applicability conditions of the component transformations.

The built framework permits the definition of a personalised set of schema refinement operators. The conditions that guarantee a safe application of these operators are derived automatically by applying the given rule.

Because of its generality, the framework presented can also be exploited to derive the correctness conditions of schema transformations that are specified in other languages. This can be obtained by first expressing the semantics of the selected transformation as the difference between the initial and the final schema. Once this difference is given, it is very easy to construct the analogous SRL transformation as a composition of SRL primitives. At this point, the correctness conditions required are generated automatically.

The SLR framework is described in the next section. Section 3 introduces the composition operator. Section 4 presents the rule for deriving the applicability conditions of a composed schema transformation. In particular, it discusses how, by exploiting this rule, it is possible also to discover situations in which the definition of a transformation is incorrect. Section 5 shows how the results presented can be exploited to derive the applicability conditions in different transformational frameworks. To illustrate this point, a few examples are presented, taken from well known transformational frameworks. Section 6 contains concluding remarks. The algorithm for the generation of the applicability conditions is given in the Appendix.

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, allows to model the database structure and behavior into a single module, called *Database Schema* (DBS) [9]. This module encloses classes (*vein*, *visual_aspect*, *material*, *marble* and *stone*), attributes (*name*, *colour* and *type*), is-a relationships (*marble* **is-a** *material* and *stone* **is-a** *material*), integrity constraints ($\mathsf{dom}(has_aspect;has_vein) = marble$)[1] and operations ($vt \leftarrow marble_veins_types$)(see Figure 4(a) for a pictorial representation of the structural part of this schema).

The notation of Database Schema is formalized in terms of a formal model introduced within the B-Method [2]. This formalization allows to exploit the B theory and tools for proving expected properties of the DBS schemas.

A set of primitive operators that transforms DBS schemas is given. Table 1 shows these operators. Note that each transformation has a further parameter, which has been omitted in the table for simplicity, that specifies the DBS schema it is applied to. The name of each operator and parameter gives an intuitive understanding of the operator semantics. 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.

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:

Definition 1 (DBS schema refinement relation)
A DBS schema S_1 refines a DBS schema S_2 if:

- S_1 and S_2 have the same signature;

[1]; is the relation composition operator.

```
database schema
    Materials

classes
    class vein of VEIN
        with (type:string)
    class visual_aspect of VISUAL_ASPECT
        with (colour:string,
              has_vein:vein)
    class material of MATERIAL
        with (name:string,
              has_aspect:visual_aspect)
    class marble is-a material
        with ()
    class stone is-a material
        with ()

constraints
    dom(has_aspect;has_vein) = marble

initialization
    material,name,has_aspect,visual_aspect,colour,has_vein,vein,
    type,marble,stone:= empty

operations
    vt ←marble_veins_types =
        vt:={t | ∃ m∈marble · has_aspect;has_vein(m)=v ∧ type(v)=t}
end
```

Figure 1: A Database Schema

- there exists a one to one correspondence between the states modelled by S_1 and S_2;
- 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 formal definition of this criterion is given in [15].

The applicability conditions consists of the conjunction of simple conditions, called *applicability predicates*. Two kinds of applicability predicates can be distinguished:

1. Applicability predicates that prevent from the application of meaningless refinements. These, can be partitioned into:

SRL
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*)

Table 1: SRL language

- predicates requiring that an element be/be not in the input schema (these will be indicated in the rest of the paper with, respectively, x$\in S$ and x $\notin S$). For example, an attribute to be removed has to be in the input schema, a class to be added has not to be in the input schema;
- predicates requiring that the expression in a transformation call be given in terms of state variables ($\{x_1, \cdots, x_n\} \subseteq S$, where $x_1, \cdots, x_n$ are the free variable of the expression). For example, the expression that defines how a new class can be derived from the rest of the schema components must contain only state variables of the input schema.

2. Applicability predicates that prevent from breaking the correctness of the design. These, can be partitioned into:

   - predicates requiring that only redundant elements be removed (Constr $\Rightarrow$ x=E). For example, a class cannot be removed if the equality condition specified in the call is not implicit in the schema constraints;
   - predicates requiring that the DBS inherent constraints be preserved (Constr $\Rightarrow$ Inh). This means that each time the input is a DBS schema also the result must be a DBS schema. For example, it is not possible to remove a class if there is an attribute that refers this class, or it is not possible to add an is-a relationship if this operation produces an is-a loop within the schema;
   - predicates requiring that the body of an operation be substituted with an algorithmic refinement of the previous one (Body$'$ $\sqsubseteq$ Body) (see [2] for the definition of $\sqsubseteq$).

Let us outline that the main concerns in defining this framework have been simplicity and generality. These qualities are achieved defining both the model and the schema refinement language provided with very primitive mechanisms.

SRL, as presented above, however, is not still sufficiently general to be used to interpret other schema design frameworks. The schema transformations are usually more complex of those listed above. In order to overcome this limitation, in the next section SRL is extended. The extension renders it complete, i.e. able to express all the possible DBS schema transformations.

3 Composition operator

This section introduces a composition operator for SRL. Note that, from now on, for notational convenience, SRL extended with the composition operator will be simply named SRL.

The following preliminary definition is needed before introducing the composition operator.

Definition 2 (Consistent operation modification)
A set of SRL schema transformations $t_1, t_2, \ldots, t_n$ specifies *consistent operation modifications* if, for each pair of transformations (t_k, t_j), with $1 \leq k,j \leq n$, and $k \neq j$, that modify the same operation *op*, at last one of the conditions below holds:

$$body_k \sqsubseteq body_j$$
$$body_j \sqsubseteq body_k$$

where $body_k$ and $body_j$ are the new behaviour of *op*, specified by, respectively, t_k and t_j.

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.

Given the above definition, the SRL composition operator can be defined as follows.

Definition 3 (Composition operator "$\circ$")
Let $t_1, t_2, \ldots, t_n$ be a set of SRL schema transformations that specify consistent operation modifications. Let *<Cl,Attr,IsA,Constr,Op>* be a DBS schema where: *Cl* is a set of classes, *Attr* is a set of attributes, *IsA* is a set of is-a relationships, *Constr* is a set of integrity constraints, and *Op* is a set of 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 $\cup$ *AddedCl - RemovedCl,*
Attr $\cup$ *AddedAttr - RemovedAttr,*
IsA $\cup$ *AddedIsA - RemovedIsA,*
[RemSubst](Constr* $\wedge$ *AddedConstr),*
[RemSubst]Op'>*

where *AddedCl/RemovedCl*, *AddedAttr/RemovedAttr* and *AddedIsA/RemovedIsA* are sets formed, respectively, by the set of classes, attributes and is-a relationships that are added/removed by t_1, $t_2, \ldots, t_n$. *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 in terms of the new schema components. *AddedConstr* are the constraints that are associated to the new elements of the schema. They comprise 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 has the following property:

Property 1 (Completeness)
SRL is a complete DBS schema refinement language.

This property ensures that the SRL is powerful enough to express every DBS schema transformation. As a consequence of this property, the designer can progressively enrich the set of schema refinement transformations following his/her needs.

The following example illustrates how the composition operator can be used to build new transformations.

Example

Let us suppose that the transformation illustrated in Figure 2 is required. This transformation adds a direct relationship between two classes that were related by an indirect link. Moreover, it removes the relationship a_3 since, in the new situation, it is derivable from the others.

This transformation can be built as composition of simple SRL transformations in the following way:

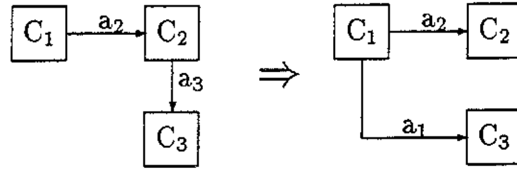


Figure 2: path_replacement

path_replacement$(C_1,\ a_1,\ C_2,\ a_2,\ a_3) =$

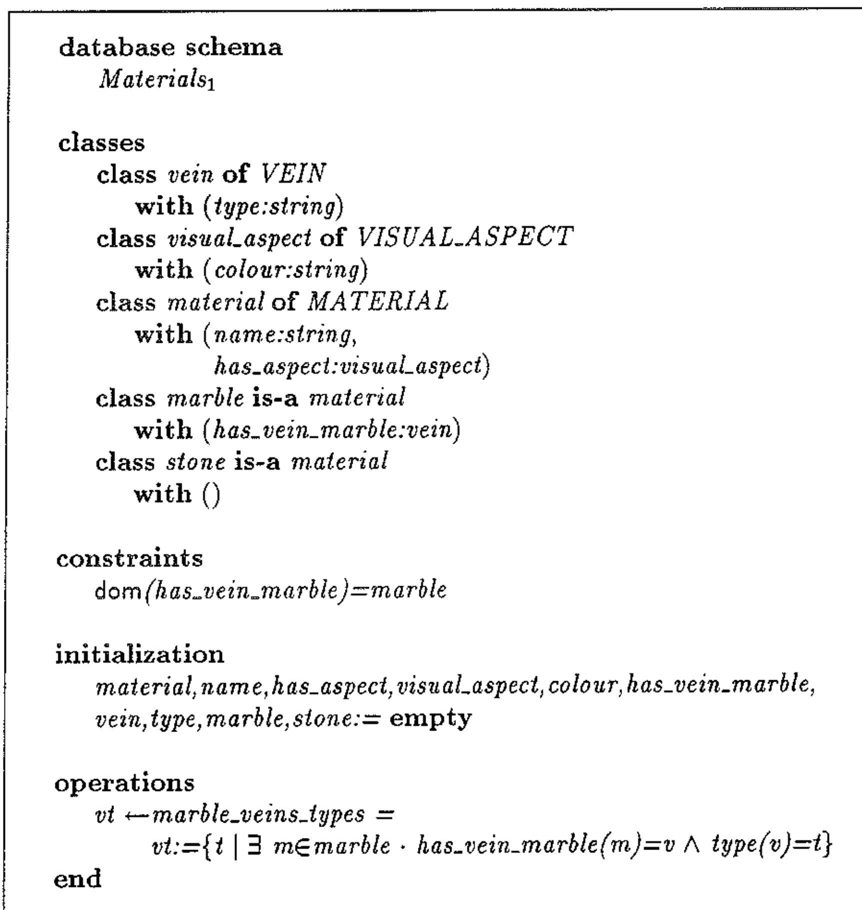
$\quad$ *add.attr*$(a_1,\ C_1,\ a_1{=}a_2;a_3) \circ$

$\quad$ *rem.attr*$(a_3,\ C_2,\ a_3{=}a_2^{-1};a_1)$

The transformation *path_replacement* can be used as any other SRL transformation. For example, it can be applied to the database schema *Materials* of Figure 1 as follows:

path_replacement(marble, has_vein_marble, visual_aspect, has_aspect, has_vein)(Materials)

Figure 3 presents the DBS schema that is produced by this transformation. Figure 4 illustrates graphically the effect of the transformation on the static part of the schema.



database schema
Materials$_1$

classes
class *vein* **of** *VEIN*
with (*type:string*)
class *visual_aspect* **of** *VISUAL_ASPECT*
with (*colour:string*)
class *material* **of** *MATERIAL*
with (*name:string,*
has_aspect:visual_aspect)
class *marble* **is-a** *material*
with (*has_vein_marble:vein*)
class *stone* **is-a** *material*
with ()

constraints
dom*(has_vein_marble)=marble*

initialization
material, name, has_aspect, visual_aspect, colour, has_vein_marble,
vein, type, marble, stone:= **empty**

operations
vt ←*marble_veins_types* =
$vt{:=}\{t \mid \exists\ m{\in}marble \cdot has_vein_marble(m){=}v \wedge type(v){=}t\}$
end

Figure 3: DBS Materials$_1$

This section has illustrated as the DBS schema refinement transformations can be dynamically built. The next section shows as, in this dynamic context, it is still possible to support the designer in carrying out a correct design process.

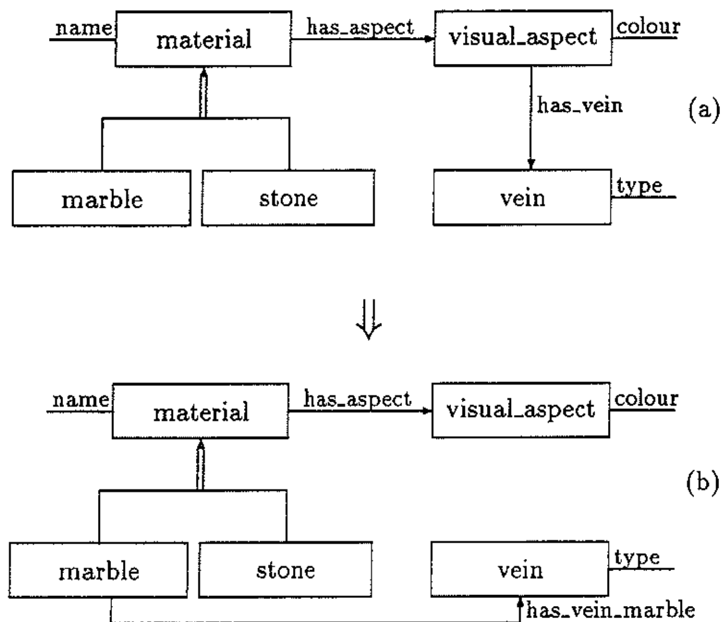


Figure 4: path_replacement

4 Applicability conditions

The generation of the applicability conditions of a composed transformation has a double purpose. First, it permits to highlight some mistakes in the definition of the transformation. Second, it provides a set of sufficient conditions for ensuring that the application of the transformation results in a correct design. What follows describes how these conditions are generated. Then, the two uses of the applicability conditions are discussed.

The applicability conditions of a composed transformation are generated constructively [17]. The construction is done by an algorithm, called *Applicability Condition Generating Algorithm* (ACGA), given in Appendix. This algorthm takes as input the set of SRL transformations and their applicability conditions and returns a set of applicability predicates. The algorithm generates the applicability conditions by considering the schema structure and the modifications brought by the component transformations. The applicability conditions of the composed transformation are given by the conjunction of the predicates that are returned by the algorithm.

As far as the applicability conditions of composed transformations, the following property holds [17]:

Property 2 (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 SRL database design.

4.1 Applicability of a transformation

As there are no constraints on how the transformations should be composed, it may happen that a defined transformation results to be never applicable, i.e. its applicability conditions are never verified. If t is a composed transformation, with n parameters and applicability conditions $appl_t$, the proof of the following condition permits to exclude such wrong definition

$$\exists p_1, \cdots, p_n, S \cdot appl_{t(p_1,\cdots,p_n)}(S)$$

where p_1, $\cdots$, p_n are the parameters of t and S is a DBS schema.

The transformation is applicable if at least an instance of the parameters and a schema that verifies the applicability conditions exist. Otherwise, there is something wrong in the definition of t.

To illustrate the last point, let us examine the following example[2]:

$$\begin{aligned} &specialisation(C_1,\ C_3,\ C_3,\ a,\ v) = \\ &\qquad add.is\text{-}a(C_3,\ C_2) \circ \\ &\qquad rem.is\text{-}a(C_2,\ C_1) \circ \\ &\qquad rem.class(C_2,\ C_2 = \mathsf{dom}(a \rhd \{v\})) \end{aligned}$$

The applicability conditions of this transformation are given in Table 2:

Applicability conditions
$C_3 \in Cl$ $C_2 \in Cl$ $(C_3, C_2) \notin IsA$ $(C_2, C_1) \in (IsA \cup \{(C_3, C_2)\})$
$C_2 \notin \{a, v\}$
$(Constr \wedge AddedConstr) \Rightarrow \neg(C_2 \text{ is-a-reach } C_3)$ $(Constr \wedge (AddedConstr - \{C_3 \text{is-a} C_2\})) \Rightarrow C_3 \subseteq C_2$ $(Constr \wedge AddedConstr) \Rightarrow C_2 = \mathsf{dom}(a \rhd \{v\})$
$\neg\exists a \in Attr(C_2)$ $\neg\exists C \in Cl \cdot ((C,C_2) \in ((IsA - \{(C_2, C_1)\}) \cup \{(C_3,C_2)\}) \vee (C_2,C) \in ((IsA - \{(C_2,C_1)\}) \cup \{(C_3,C_2)\}))$

Table 2: Applicability conditions of "specialisation"

Carrying out the verification of the above applicability conditions we discover that the last condition will be never verified, since C can be instantiated with "C_3". This discovery gives us an indication of the problems in the definition of the transformation. The transformation removes the class C_2 that is is-a related to C_3. Indipendently from the values that will be given to C_2 and C_3, when the transformation will be instantiated it will always produce a dangling is-a relationship , as illustrated in Figure 5.

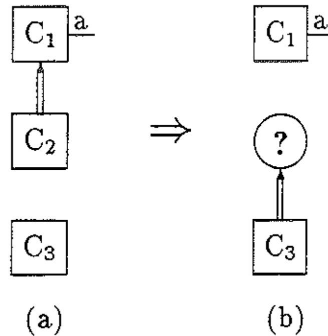


Figure 5: Specialisation

The proof of the applicability conditions of an SRL transformation permits thus to rule out wrong definition, as the one given in this example.

[2] In that follows, $\rhd$ stands for the range restriction, 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 and $C_3$is-aC_2 stands for the inherent constraint "C_3 is a subclass of C_2".

4.2 Correctness of a design step

At the beginning of Section 4, we have seen that the applicability conditions of a composed transformation are generated dynamically from the definition of the transformation.

The applicability conditions so generated are parametric with respect to the parameters of the composed transformation. By reasoning on these conditions, 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, 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. 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 the component transformations and the values of the parameters, several applicability predicates are discharged. The ACGA algoritm, reported in the Appendix, actually implement a mix between the generation of the applicability conditions and the second pruning, The result is the set of applicability predicates that the designer has to prove for a particular application of the transformation. Notice that this set 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.

For now on, for "applicability conditions" it will be meant the conditions returned by the application of the ACGA algorithm.

As example of dynamic generation of the applicability conditions of a composed transformation, let us see which are the applicability conditions of the transformation *path_replacement*, as invoked in the example of Section 3. The following abbreviations are used:

$NewConstr_1 = Constr \wedge Inh \wedge has_vein{=}has_aspect^{-1};has_vein_marble$
and
$NewConstr_2 = Constr \wedge Inh \wedge has_vein_marble{=}has_aspect;has_vein$

where *Constr* stands for the constraints of the initial schema and *Inh* stands for the inherent constraints that are implicitly added by the transformation.

The applicability conditions of *path_replacement* are:

- $\mathrm{NewConstr}_1 \Rightarrow \mathsf{dom}(has_aspect;has_vein) \subseteq marble$
- $\mathrm{NewConstr}_2 \Rightarrow has_vein = has_aspect^{-1};has_vein_marble$

The first condition requires that the added relationship, defined as composition of *has_aspect* and *has_vein*, be defined on the class *marble*. The second condition requires that the removed relationship be derivable as the sequential composition of the remaining ones. Those listed above are the only applicability conditions that are returned to the designer. The others are checked and discharged by the ACGA algorithm automatically.

As a concluding remark of this section, note that 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.

5 Exploiting SRL in other design frameworks

Having a very primitive level of basic transformations and a powerful mean for composing them permits to transfer our approach for a more reliable design also to other frameworks.

As a matter of fact, in all the design frameworks that can be interpreted as a special case of the one that has been described, the applicability conditions of any transformation can be simply generated. It is sufficient, first, to define the refinement transformation as the difference between the initial and the final schemas and, then, to express this difference as composition of SRL primitives. At this point, the applicability conditions follow automatically.

This approach to the derivation of the applicability conditions of a refinement transformation can be useful in all those design contexts in which the preconditions for a correct design are not given. These include also those contexts in which there are no established transformations but the design is done by writing down the logical schema directly. In this case the transformation is implicit and, of course, there are no preconditions for guaranteeing its correctness. This approach may be useful also when there are applicability conditions, but they are only given informally. In these cases, assistance tools for the verification of these preconditions cannot be built. Using the suggested approach we can take advantage of those provided for SRL.

The approach illustrated above has an inherent limitation: it can be applied only if its premises agree with those of SRL. In particular, the employed model must be a submodel of DBS and the DBS schema refinement relation must conform to the one that has been given in Section 2.

In order to illustrate how the approach proposed can be useful in other frameworks, we present two examples of derivation of applicability conditions. The examples consider two refinement transformations taken from two different well known languages.

Example 1

The first example considers a schema transformation that belongs to the set proposed in [3]. The transformations within this set do not change the information content of the schema. Moreover, they conform to SRL. The transformation chosen is: *elimination of dangling subentities in generalisation hierarchies.* For brevity, below it will be named *elimination.*

The transformation *elimination* removes n non overlapping subclasses and reduces them to a superclass. The elements of the superclass are partitioned in n groups by the value of added attribute. Figure 6 shows this transformation.

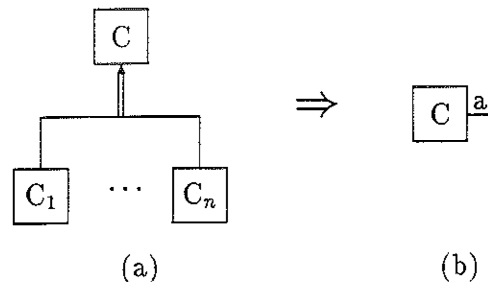


Figure 6: elimination

The schema in Figure 6(b) differs from the schema in Figure 6(a) since in the second one there is a new attribute a and the classes $C_1, \cdots, C_n$ and their is-a relationships are missing. This difference can be easily expressed as composition of SRL primitives:

$elimination(a,\ C,\ (v_1, \cdots, v_n),\ (C_1, \cdots, C_n)) =$
$\quad add.attr(a,\ C,\ a=\{(x,y) \mid x \in (C_1 \cup \cdots \cup C_n) \land$
$\qquad (x \in C_1 \rightarrow y=v_1) \land \cdots \land (x \in C_n \rightarrow y=v_n)\}) \circ$
$\quad rem.class(C_1, C_1 = \mathrm{dom}(a \rhd \{v_1\})) \circ \cdots \circ rem.class(C_n, C_n = \mathrm{dom}(a \rhd \{v_n\})) \circ$
$\quad rem.isa(C_1, C) \circ \cdots \circ rem.isa(C_n, C)$

The transformation *elimination* can be applied to the schema S in Figure 7(a) to obtain the schema of Figure 7(b):

elimination(type, Material, (marble, stone), (Marble, Stone))(S)

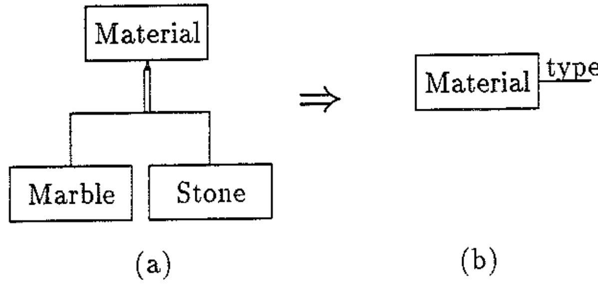


Figure 7: elimination

The applicability conditions associated to the above instantiation of the *elimination* are the following:

- NewConstr $\Rightarrow$ $(\mathrm{dom}(\{(m,t) \mid m \in (Marble \cup Stone) \land$
  $(m \in Marble \rightarrow t = marble \land m \in Stone \rightarrow t = stone)\}) \subseteq Material)$
- NewConstr $\Rightarrow$ $Marble = \mathrm{dom}(type \rhd \{marble\})$
- NewConstr $\Rightarrow$ $Stone = \mathrm{dom}(type \rhd \{stone\})$

where *NewConstr* stands for the conjunction of the constraints of the initial schema and a subset of those added. This subset consists of all the constraints added by the component transformations that differ from the transformation that has generated the condition.

Example 2

The second example is taken from [11]. Here, a set of ER schema transformations to support the designer during the schema development is proposed. Among all the given transformations there is a group of them that are semantics-preserving, i.e. they do not change the information content of the schema. One of these transformations is *disaggregation a compound attribute.* For brevity, it will be called *disaggregation.*

Let us see how the applicability conditions of this transformation can be derived.

The transformation *disaggregation* replaces the compound attribute with its component fields as shown in Figure 8.

The schema in Figure 8(a) differs from the schema in Figure 8(b) since in the second one there are n new attributes, $a_a_1, \cdots, a_a_n$, and the attribute a is missing.

This difference can be expressed as composition of SRL primitives as follows[3]:

[3] In that follows, $\mathrm{prj}_i(z)$ stands for the projection of z on the i-th element.

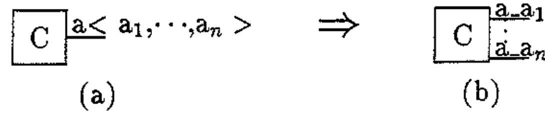


Figure 8: disaggregation

$disaggregation((a_1, \cdots, a_n), C, a) =$
$add.attr(a_a_1, C, a_a_1 = \{(x,y) \mid x \in C \wedge y = \mathrm{prj}_1(a(x))\}) \circ \cdots \circ$
$add.attr(a_a_n, C, a_a_n = \{(x,y) \mid x \in C \wedge y = \mathrm{prj}_n(a(x))\}) \circ$
$rem.attr(a, C, a = \{(x,y) \mid x \in C \wedge y = <a_a_1(x), \cdots, a_a_n(x)>\})$

Let us, for example, apply the transformation *disaggregation* to the schema S in Figure 9(a).

disaggregation((colour, vein), Material, aspect)(S)

This instantiation produces the schema in Figure 9(b).

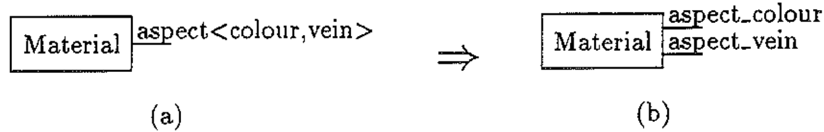


Figure 9: disaggregation

The only applicability condition of *disaggregation* is:

- NewConstr $\Rightarrow$ $aspect = \{(m,a) \mid m \in Material \rightarrow a = <aspect_colour, aspect_vein>\}$

where *NewConstr* are defined as in the previous example.

6 Conclusions

This paper has proposed a solution to the problem of offering a flexible instrument for supporting a correct database design. Flexibility is achieved by employing a model that can represent both object and semantic models and by providing a means for defining refinement transformations dynamically. Flexibility, in this case, does not degrade correctness since the applicability conditions can still be associated to the dynamically built transformations. The paper has shown also as, by exploiting the flexibility of SRL, it is possible to employ it for generating the applicability conditions of refinement transformations proposed in other contexts. This use of the illustrated framework permits to improve the reliability of the database design process also in frameworks where no specific instrument for such purpose exists.

We have experimented this particular application of the framework proposed in carrying out the design of two databases. The first is the multimedia database for supporting the MIAOW system[18]. 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 second is a database that maintains multimedia data about the history of Computer Science in Italy[19]. This database was designed as part of the Italian National Project *Museo Virtuale della Storia dell'Informatica.* The design of these two database was carried out by generating a sequence of OMT-like schemas[16]. Each schema in the sequence was produced by ad-hoc transformations. For each of such transformations, the corresponding applicability conditions were

generated. These design experiences, on the one hand, confirmed the utility of the approach illustrated in discovering specific design mistakes and, on the other, suggested us improvements to our framework. In particular, it highlighted situations in which an automatic discharging was preferable.

Let us conclude by outlining that the particular type of formalisation that has been chosen for the presented framework allows to built tools that assist in the generation and discharge of the applicability conditions. In particular, the same B tools that can be used for proving properties of DBS schema can also be exploited for supporting the discharge of the SRL applicability preconditions. Moreover, they can also be used to implement the more sophisticated version of the ACGA algorithm.

References

[1] Petia Assenova, Paul Johannssen. Improving Quality in Conceptual Modelling by the Use of Schema Transformation. In *Lecture Notes in Computer Science*, n.1157, pp.277-291, Springer-Verlag, 1996.

[2] J.R.Abrial. *The B-Book.* Cambridge University Press, 1996.

[3] C. Batini, S. Ceri and Navathe S. B.. *Conceptual Database Design.* Redwood City, CA, The Benjamin/Cummings Publishing Company, Inc., 1992.

[4] 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.

[5] P.L.Bergstein. Object-Preserving Class Transformations. in Proc. *Object-Oriented Programming Systems, Languages and Applications Conference*, Phoenix, Arizona, October 1991. In Special Issue of SIGPLAN Notice, 26(11), pp.299-313, 1991.

[6] P.L.Bergstein and W.L.Hürsch. Maintaining Behavioral Consistency during Schema Evolution. In *First JSSST International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, Lecture Notes in Computer Science, Springer-Verlag, pp.176-193, November 1993.

[7] Peter Mc. Brien, Alexandra Poulovassilis. A Formal Framework for ER Schema Transformation. In *Lecture Notes in Computer Science*, n.1331, pp.408-421, Springer-Verlag, 1997.

[8] P.van Bommel. Database design by computer-aided schema transformations. *Software Engineering Journal*, pp.125-132, July 1995.

[9] D.Castelli and E.Locuratolo. ASSO: A Formal Database Design Methodology, in *Information Modelling and Knowledge Bases VI*, H. Jaakkaola et al.eds., IOS-Press, 1994.

[10] A.D'Atri and D.Saccà. Equivalence and Mapping of Database Schemas. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pp.187-195, Singapore, August 1984.

[11] J.L.Hainaut. Transformation-based Database Engineering, *Tutorial of the Very Large Data Bases Conference*, Zurigh, Switzerland, September 1995.

[12] J.L.Hainaut, V.Englebert, J. Henrard, J.M.Hick and D.Roland. Evolution of Database Applications: the DB-MAIN Approach. In *Proceedings of the 13th International Conference on ER Approach*, Manchester, Springer-Verlag, 1994.

[13] I.Kobayashi. Losslessness and Semantic Correctness of Database Schema Transformations: Another Look of Schema Equivalence. *Information Systems*, 11(1), pp.41-59, 1986.

[14] K.J.Lieberherr, W.L.Hürsch and C.Xiao. Object-Extending Class Transformations.II *Formal Aspects of Computing*, 6, pp.391-416, 1994.

[15] S.Pisani and R.Occhipinti, *Definizione di un insieme di trasformazioni di schemi per la progettazione di basi di dati*, Tesi di Laurea, Computer Science Dept. University of Pisa, 1996.

[16] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[17] Castelli D. and Pisani S. *A Transformational Approach to Database Design*, IEI-CNR Technical Report, 1998.

[18] *MIAOW Multimedia Database: Revised Design and Implementation*, MIAOW-CNR-REP-001-007. 1996.

[19] De Marco G. and Pisani S. *Disegno e Realizzazione della Base di Bati Multimediale di Supporto al Progetto "Museo Virtuale della Storia dell'Informatica in Italia"*, IEI-CNR Internal Note B4-25, 1997.

Appendix: The Applicability Condition Generating Algorithm

The *Applicability Condition Generating Algorithm* (ACGA) is described below. This algorithm 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*> and 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:

- *AddedCl/Attr* and *RemovedCl/Attr* indicate, respectively, the set of classes and attributes that are added and removed;
- *Attr*(C), *AddedAttr*(C) and *RemovedAttr*(C) indicate, respectively, the set of attributes that are defined on the class C in the initial schema, added to C and removed from C by the composed transformation;
- *AddedConstr* are the constraints that are added by the component transformations;
- *RemSubst* are the variable substitutions induced by all the removals of schema components;
- C_1 is-a C_2 stands for the inherent constraint "C_1 is a subclass of C_2";
- *a* attribute-of C is the inherent constraint "*a* is an attribute of the class C";
- *ver_appl* is the set of applicability conditions that are proved to be verified when the composed transformation is defined.

Applicability Condition Generating Algorithm

The algorithm consists of four steps. 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 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 1
appl$_\circ$:= $\emptyset$

Step 2
% The conditions generated by this step require that multiple additions or removals of a component with the same name be equivalent.

for all x, E, F· x∈*AddedCl* ∧ x=E∈*AddedConstr* ∧ x=F∈*AddedConstr* ∧ $E \neq F$
do appl$_\circ$:= appl$_\circ$∪ {(*Constr*∧*AddedConstr* - {x=E, x=F}) ⇒ (x=E ⇔ x=F)};
% Two component transformations can add the same class only if the expressions that define the class are equivalent.

for all x, E, F· x∈(*AddedCl* ∩ *RemovedCl*) ∧ x=E∈*AddedConstr* ∧ x=:F∈*RemSubst* ∧ $E \neq F$
do appl$_\circ$:= appl$_\circ$∪ {(*Constr*∧*AddedConstr*) ⇒ (x=E ⇔ x=F)};
% Two component transformations can add and remove the same class only if the expressions that define the class are equivalent.

for all x, E, $F\cdot$ x$\in Cl \wedge$ x=:$E\in RemSubst \wedge$ x=:$F\in RemSubst \wedge E\neq F$
do $\text{appl}_o := \text{appl}_o \cup \{(Constr \wedge AddedConstr) \Rightarrow (x=E \Leftrightarrow x=F)\}$;
% Two component transformations can remove the same class only if the expressions that define the class are equivalent.

for all x, E, $F\cdot$ x$\in AddedAttr \wedge$ x=$E\in AddedConstr \wedge$ x=$F\in AddedConstr \wedge E\neq F$
do $\text{appl}_o := \text{appl}_o \cup \{(Constr \wedge AddedConstr - \{x=E, x=F\}) \Rightarrow (x=E \Leftrightarrow x=F)\}$;
% Two component transformations can add the same attribute only if the expressions that define the attribute are equivalent.

for all x, E, $F\cdot$ x$\in (AddedAttr \cap RemovedAttr) \wedge$ x=$E\in AddedConstr \wedge$ x=:$F\in RemSubst \wedge E\neq$
do $\text{appl}_o := \text{appl}_o \cup \{(Constr \wedge AddedConstr) \Rightarrow (x=E \Leftrightarrow x=F)\}$;
% Two component transformations can add and remove the same attribute only if the expressions that define the attribute are equivalent

for all x, E, $F\cdot$ x$\in Attr \wedge$ x=:$E\in RemSubst \wedge$ x=:$F\in RemSubst \wedge E\neq F$
do $\text{appl}_o := \text{appl}_o \cup \{(Constr \wedge AddedConstr) \Rightarrow (x=E \Leftrightarrow x=F)\}$;
% Two component transformations can remove the same attribute only if the expressio that define the attribute are equivalent.

Step 3

% If the equivalence stated above holds, then the composition of all the additions and removal of the same component has no effect. As a consequence, many of the applicability predicates associated to these transformations do not need to be proved. The algorithm, then, adds to th set *appl* only the applicability predicates that have necessarily to be proved.

appl := $\emptyset$;
for all x $\cdot$ x$\in AddedCl \wedge$ x$\notin RemovedCl$ **do**
any E **where** x=$E\in AddedConstr$ **then**
appl := appl $\cup$ {x$\notin Cl$, x$\notin$**Free**(E), **Free**(E)$\subseteq$($Cl \cup Attr$)}

for all x $\cdot$ x$\in RemovedCl \wedge$ x$\notin AddedCl$ **do**
any E **where** x:=$E\in RemSubst$ **then**
appl := appl $\cup$ {x$\in Cl$, x$\notin$**Free**(E), $Constr \Rightarrow$ x=E,
$\neg\exists$a$\in Attr$(x), $\neg\exists$C$\in Cl \cdot$ ((C,x)$\in IsA \vee$ (x,C)$\in IsA$)}

for all x $\cdot$ x$\in AddedAttr \wedge$ x$\notin RemovedAttr$ **do**
any E **where** x=$E\in AddedConstr$ **then**
if $\neg\exists$C$\in$($Cl \cup AddedCl$) $\cdot$ x$\in AddedAttr$(C) **then** appl_o := **false**
else appl := appl $\cup$ {C$\in Cl$, x$\notin$**Free**(E), **Free**(E)$\subseteq$($Cl \cup Attr$), x$\notin Attr$,
$Constr \Rightarrow$ dom(E)$\subseteq$C |
$\exists$C$\in$($Cl \cup AddedCl$) $\cdot$ x$\in AddedAttr$(C)}

for all x $\cdot$ x$\in RemovedAttr \wedge$ x$\notin AddedAttr$ **do**
any E **where** x:=$E\in RemSubst$ **then**
if $\neg\exists$C$\in$($Cl \cup AddedCl$) $\cdot$ x$\in AddedAttr$(C) **then** appl_o := **false**
else appl := appl $\cup$ {C$\in Cl$, x$\notin$**Free**(E), x$\in Attr$(C),
$Constr \Rightarrow$ x=E | $\exists$C$\in$($Cl \cup AddedCl$) $\cdot$ x$\in Attr$(C)}

for all (x,y) $\cdot$ (x,y)$\in AddedIsA \wedge$ x$\notin RemovedIsA$ **do**
appl := appl $\cup$ {x$\in Cl$, y$\in Cl$, (x,y)$\notin IsA$, $Constr \Rightarrow \neg$(y is-a-reach x), $Constr \Rightarrow$ x$\subseteq$y}

for all $(x,y) \cdot (x,y) \in \mathit{RemovedIsA} \wedge x \notin \mathit{AddedIsA}$ **do**
appl := appl $\cup$ $\{(x,y) \in \mathit{IsA}\}$

for all $x \cdot x \in (\mathit{AddedCl} \cap \mathit{RemovedCl})$ **do**
appl := appl $\cup$ $\{\neg\exists a \in \mathit{Attr}(x), \neg\exists y \in \mathit{Cl} \cdot ((x,y) \in \mathit{IsA} \vee (y,x) \in \mathit{IsA})\}$
%This is the only case that requires that some applicability predicates be verified for the component transformations that add and remove the same schema component

Step 4
appl := appl - *ver_appl*;
% This assignment removes the applicability conditions that have been verified when the composed transformation has been defined.

repeat
p := **extract**(appl);
appl := appl - p;
case type(p) **of**

$x \in \mathit{Cl}$
then if not($x \in (\mathit{AddedCl} \cup \mathit{Cl})$) **then** $appl_o$:= **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 $x \notin \mathit{Cl}$
then if not($x \notin \mathit{Cl}$) **then** $appl_o$:= **false**
% This predicate is false if the class x is in the initial schema. It is true otherwise.

or $x \in \mathit{Attr}(C)$
then if not($x \in \mathit{Attr}(C)$) **then** $appl_o$:= **false**
% This predicate is false if the x is not an attribute of C in the initial schema. It is true otherwise.

or $x \notin \mathit{Attr}$
% This condition guarantees that all the names of the schema attributes are distinct.
then if not($x \notin \mathit{Attr}$) **then** $appl_o$:= **false**
% This condition is false if the attribute x is in the initial schema. It is true otherwise.

or $x \in \mathit{IsA}$
then if not($x \in \mathit{IsA}$) **then** $appl_o$:= **false**
% This predicate is false if the is-a relationship x is not in the initial schema. It is true otherwise.

or $x \notin \mathit{IsA}$
then if not($x \notin \mathit{IsA}$) **then** $appl_o$:= **false**
% This predicate is false if the is-a relationship x is in the initial schema. It is true otherwise.

or $\mathbf{Free}(E) \subseteq (\mathit{Cl} \cup \mathit{Attr})$
then if not($\mathbf{Free}(E) - (\mathit{AddedCl} \cup \mathit{AddedAttr}) \subseteq \mathit{Cl} \cup \mathit{Attr}$) **then** $appl_o$:= **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** $appl_o$:= **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 $C_1 \in (Cl \cap AddedCl \cap RemovedCl)$ **then do nothing**

else if $(\exists C \in (Cl \cup AddedCl) \cdot ((C_1, C) \in (AddedIsA - RemovedIsA) \vee$
$(C, C_1) \in (AddedIsA - RemovedIsA))) \vee$
$(\exists C \in Cl \cdot ((C, C_1) \in (IsA - RemovedIsA) \vee$
$(C_1, C) \in (IsA - RemovedIsA))) \vee$
$(\exists C \in Cl \cdot ((C,C_1) \in (IsA \cap AddedIsA \cap RemovedIsA) \vee$
$(C_1,C) \in (IsA \cap AddedIsA \cap RemovedIsA)))$

then $appl_o$:= **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 $C_1 \in (Cl \cap AddedCl \cap RemovedCl)$ **then do nothing**

else if $(\exists a \in (AddedAttr(C_1) - RemovedAttr(C_1))) \vee$
$(\exists a \in (Attr(C_1) - RemovedAttr(C_1))) \vee$
$(\exists a \in (Attr(C_1) \cap AddedAttr(C_1) \cap RemovedAttr(C_1)))$

then $appl_o$:= **false**

% The predicate has not to be checked if the class C_1, that is removed, belongs to the init 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 $\exists\ C_3, \cdots, C_n \in (Cl \cup AddedCl) \cdot (C_1 = C_2 \cup C_3 \cup \cdots \cup C_n) \in AddedConstr$.

then do nothing

else $appl_o := appl_o \cup \{(Constr \wedge (AddedConstr - \{C_2 \text{ is-a } C_1\})) \Rightarrow C_2 \subseteq C_1\}$;

% The predicate is true if there is a transformation in the composition that adds the class and defines it as the union C_2 and other classes.

or $Constr \Rightarrow \neg(C_1 \text{ is-a-reach } C_2)$

then $appl_o := appl_o \cup \{(Constr \wedge AddedConstr) \Rightarrow \neg(C_1 \text{ is-a-reach } C_2)\}$;

or $Constr \Rightarrow x=E$

then if $x=E \in AddedConstr$ **then do nothing**

else $appl_o := appl_o \cup \{(Constr \wedge AddedConstr) \Rightarrow x=$]

or $Constr \Rightarrow \text{dom}(F) \subseteq C$

then $appl_o := appl_o \cup \{(Constr \wedge (AddedConstr - \{a \text{ attribute-of } C \mid$
$\exists a \cdot a=F \in AddedConstr\})) \Rightarrow \text{dom}(F) \subseteq C$

or $body_1 \sqsubseteq body_2$

then $appl_o := appl_o \cup \{[RemSubst^*]\ body_1 \sqsubseteq [RemSubst^*]\ body_2\}$

% The variable substitutions must be taken into account when evaluating the refinement relation.

```
until appl = ∅ ∨ appl_o = false;
return appl_o;
```