Three Levels of Reuse for Supporting the Evolution of Database Schemas

Donatella Castelli

Istituto di Elaborazione della Informazione-CNR, Pisa, Italy
castelli@iei.pi.cnr.it

Abstract. This paper presents a reuse technique which permits to simplify redesign that may occur when the conceptual schema is changed. This technique is based on the reuse of the refinement steps, the verifications and the schemas of the original design. For each kind of reuse, the technique specifies the conditions under it can be applied and how the reused components should evolve.

1 Introduction

When a conceptual schema evolves the whole design process has to be reexamined to adapt it to the new situation. In the database design practice it is often recommended to manage this evolution by employing methods based on the reuse of previous design [1,3,11,14,4]. The use of these methods, however, is far from being trivial. Often, a significant effort is required to select the appropriate components to be reused and to understand which modifications must be operated. This effort, on the one hand discourages the use of these methods, and on the other reduces the advantages of reuse.

This paper shows how this difficulty can be overcome when the schema design is interpreted through schema transformations. To this aim, it proposes precisely defined database schema design maintenance techniques. These techniques specify the conditions under which reuse can be applied and how the different reused components evolve in response to the changes applied to the conceptual schema.

The techniques illustrated are embedded within the transformational framework described in [10]. Several other frameworks [1,3,2], either implicitly or explicitly transformational, can be rephrased in terms of this one. As a consequence, the results presented in this paper can be easily extended also to all these other frameworks.

The techniques proposed is introduced by progressively increasing the amount of reuse. Section 3, introduces the reuse technique. Section 4 focuses on the reuse the refinement steps. Refinement steps can be reused provided that they still produce a correct design process. Section 4 shows how the reuse of proofs can be exploited to reduce the cost of this check for correctness. The techniques presented are then generalised in Section 5 to include also the reuse of the schemas that document the design. In particular, the modifications to be applied to each

J. Akoka et al. (Eds.): ER'99, LNCS 1728, pp. 1–16, 1999.

of the schemas of the previous design are precisely characterised. Finally, concluding remarks are given in Section 6. The next Section introduces the schema design transformational framework that will be used in the rest of the paper.

2 A Framework for Database Design

The transformational framework that will be used to illustrate the proposed reuse technique is very simple and general [9]. It has been preferred to others since its characteristics permit to derive results that can be easily transferred to other database design frameworks.

In this framework the whole design relies on a single notation [8]. 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). This module encloses classes, attributes, is-a relations, integrity constraints and operation specifications.

database schema *StoneDB*
class *marbles* **of** *MATERIAL*
  with (*close_up_photo:IMAGE*; *long_distance_photo:IMAGE*)
class *for_exterior_m* **is-a** *marble*
  with (*e_colour: COLOUR*)
class *for_interior_m* **is-a** *marble*
  with (*i_colour: COLOUR*)
constraints
  $e\_colour = \{(x,y) \cdot x \in for\_exterior\_m \wedge \exists k (k \in long\_distance\_photo(x) \wedge y = get\_colour(k))\}$
  $i\_colour = \{(x,y) \cdot x \in for\_interior\_m \wedge \exists k \ (k \in close\_up\_photo(x) \wedge y = get\_colour(k))\}$
  $for\_exterior\_m \cup for\_interior\_m = marble$
initialisation
  marbles, close_up_photo, long_distance_photo,
  for_interior_m, for_exterior_m, i_colour, e_colour$:= \emptyset$
operations
  $y \leftarrow return\_colour(x) =$
   pre $x \in marble$ **then if** $x \in$ *(for_interior_m - for_exterior_m)*
                then $y := \{i\_colour(x)\}$
                else if $x \in$ *(for_exterior_m - for_interior_m)*
                    then $y := \{e\_colour(x)\}$ **else** $y := \{i\_colour(x), e\_colour(x)\}$
   end
end

Fig. 1. A Database Schema

The notation of Database Schemas is formalized in terms of a formal model introduced within the B-Method [5]. This formalization allows to exploit the B theory and tools for proving expected properties of the DBS schemas.

Both the construction of the conceptual schema and the refinement of this schema into the chosen DBMS schema are carried out by means of schema trans-

formations. Table 1 and Table 2 show, respectively, the primitives for schema construction, collectively called *Schema Transformation Language*(STL), and those for schema refinement, called *Schema Refinement Language*(SRL). Each transformation has a further parameter, which has been omitted in the tables for simplicity, which specifies the name of the DBS schema that is transformed. The semantics of each operator can intuitively be understood by the name of the operator. Both the languages comprise also a composition operator "$\circ$" which allows to construct more complex transformations [10]. This operator is enough powerful to render the two languages complete. Moreover, an additional composition operator ";" is given to compose transformations sequentially.

add.class (*class.id*,*class.type*,*class.init*)
rem.class (*class.id*)
mod.class(*class.id*,*class.type*,*class.init*)
add.attr(*attr.id*,*class.id*,*attr.type*,*attr.init*)
rem.attr (*attr.id*,*class.id*)
mod.attr (*attr.id*,*class.id*,*attr.type*,*attr.init*)
add.isa(*class.id1*,*class.id2*)
rem.isa (*class.id1*,*class.id2*)
add.constr(*constraint*)
rem.constr (*constraint*)
add.op (*op.id*,*body*)
rem.op (*op.id*)

Table 1. STL language

r.add.class (*class.id*,*expr*)
r.rem.class (*class.id*,*expr*)
r.add.attr(*attr.id*,*class.id*,*expr*)
r.rem.attr (*attr.id*,*class.id*,*expr*)
r.add.isa(*class.id1*,*class.id2*)
r.rem.isa (*class.id1*,*class.id2*)
r.mod.op (*op.id*,*body*)

Table 2. SRL language

The expressions that appear as a parameter in the r.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 SRL language does not permit to add or remove constraints and operations. It only permits to change the way in which an operation is defined. This is done both explicitly, through the r.mod.op operator, and automatically, as a side effect of the transformations that add and remove schema components. These automatic modifications add appropriate updates for each of the new schema variables, cancel the occurrences of the removed variables, and apply the proper variable substitutions.

A transformation can be applied when its *applicability conditions* are verified. These conditions, that must be verified before the application of the transformation, prevent from the execution of both meaningless transformations and transformations that can break the schema consistency and the correctness of the design. The applicability conditions are given by the conjunction of simple conditions, called *applicability predicates*. In SRL two kinds of applicability predicates can be distinguished[1]:

[1] The specification of the STL applicability predicates is omitted since it is not useful for the rest of the paper.

1. Applicability predicates that prevent from the application of meaningless refinements. These are the predicates requiring that an element be/be not in the input schema (these will be indicated in the rest of the paper with, respectively, $\mathtt{x} \in \mathtt{S}$ and $\mathtt{x} \notin \mathtt{S}$). For example, an attribute to be removed has to be in the schema, a class to be added has not to be in the schema.
2. Applicability predicates that prevent from breaking the correctness of the design. These, can be partitioned into:
   - predicates requiring that the expression in a transformation call be given in terms of state variables ($\{\mathtt{x}_1, \cdots, \mathtt{x}_n\} \subseteq \mathtt{S}$), where $\mathtt{x}_1, \cdots, \mathtt{x}_n$ 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;
   - predicates requiring that only redundant elements be removed ($\mathtt{Constr} \Rightarrow \mathtt{x=E}$). For example, a class cannot be removed if the removed element cannot be proved to be equivalent to E;
   - predicates requiring that the DBS inherent constraints be preserved ($\mathtt{Constr} \Rightarrow \mathtt{Inh}$). 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;
   - predicates requiring that the body of an operation be substituted with an algorithmic refinement of the previous one ($\mathtt{Body}' \sqsubseteq \mathtt{Body}$) (see [5] for the definition of $\sqsubseteq$).

STL and SRL are not independent since each SRL primitive transformation can be expressed as a particular composition of STL transformations. For example, *r.add.class(c, E)*, which adds a redundant class *c*, is defined as the composition of the STL primitives: *add.class(c, type(E), E)*, *add.constraint(c=E)*, and a set of *mod.op(op, subs)*, one for each operation that must be changed to introduce the appropriate updates for the new class. Similarly, the *r.rem.class(c, E)* is the composition of *rem.class(c)*, as many *rem.constraint* and *add.constraint* as are the constraints in which *c* must be replaced by *E*, and as many *mod.op* as are the operations in which a similar change of variable is needed.

The set of transformations that can be specified using SRL is a subset of those that can be specified using STL. As a matter of fact, STL specifies all and only the schema transformations that, when applied to a consistent schema, produce a consistent schema. SRL specifies exactly the subset of these schema transformations that produce refinements of the input schema, i.e., which produce schemas that satisfy the specification given by the input one. Certainly, these characteristics of the two languages hold under the assumption that the applicability conditions of the transformations are verified.

3 The Propagation of Changes

Figure 2a) shows the schematisation of a database schema design process. In the figure S_0 indicates the conceptual schema. S_0 is transformed into the final

logical schema S_2 through two refinement transformations, t_{r1} and t_{r2}, each of which produces a new schema. Each schema in the sequence contains more implementative details than the previous one. Figure 2b) shows what happen when the conceptual schema evolves as an effect of new requirements. We can schematise this evolution, again, as the application of a schema transformation. Let call this transformation t_0. The change applied to the conceptual schema is then propagated to the whole design process, i.e. new refinement steps are applied to the schema S_0' to produce the new final schema and new intermediate schemas, S_1' and S_2', are produced.

In the literature, the above approach to the propagation of changes is considered the more appropriate since, by producing a complete documentation of the new situation, it renders the subsequent modifications easier [6,7]. However, since its cost can be high, it is often recommended to carry it out by employing techniques based on the reuse of the previous design documentation. Referring to our schematisation of the design process, we might think to reuse the refinement steps, the correctness proofs and the intermediate schemas that document the design process. The reuse of the refinement steps may be useful in those situations in which no rethinking on the optimisation decisions and on the DBMS choice is planned. The reuse of the proofs done may be helpful in assessing the correctness of the new design. Finally, the reuse of the schemas may be useful to avoid the regeneration of the schemas that document the design.

Reuse, however, is really effective in reducing the complexity of redesign only if it is clear how the reusable components can be identified and which modifications must be operated to satisfy the new needs. Most of the proposals for database (re)design that rely on reuse, miss this specification [1,3,11,2,15]. This lack renders the application of reuse complex. It also makes difficult to built really useful automatic or semi-automatic redesign supporting tools. The aim of this work is to overcome this drawback by characterising precisely each of the above identified kinds of reuse.

In order to illustrate these different form of reuse a simple case of schema refinement will be used along the whole paper. This consists of a single refinement

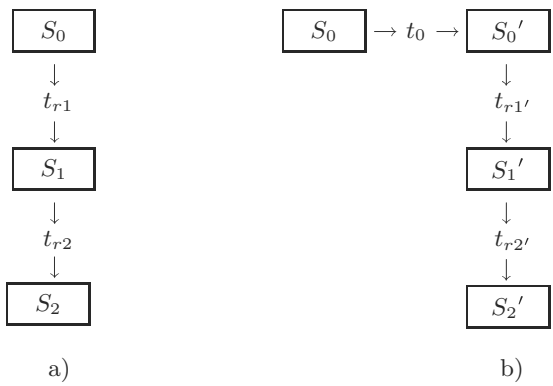

Fig. 2. The reuse of refinement steps

step (see Figure 3). It takes as input the schema given in Figure 1 and produces a schema in which the possibly overlapping *for_exterior_m* and *for_interior_m* classes are replaced with three mutually exclusive classes: *ei*, *e-i* and *i-e*. Moreover, the redundant attributes *i_colour* and *e_colour* are removed.

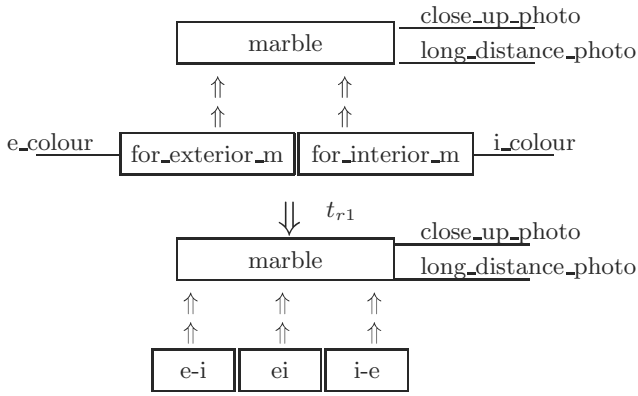

Fig. 3. An example of schema refinement

The transformation t_{r1} that specifies the refinement step[2] is defined as follows:

r.add.class(*ei*, *ei*=(*for_interior_m* ∩ *for_exterior_m*)) ∘
r.add.class(*i-e*, *i-e*=(*for_interior_m* - *for_exterior_m*)) ∘
r.add.class(*e-i*, *e-i*=(*for_exterior_m* - *for_interior_m*)) ∘
r.add.isa(*marble*, *ei*) ∘ r.add.isa(*marble*, *i-e*) ∘ r.add.isa(*marble*, *e-i*) ∘
r.rem.isa(*marble*, *for_interior_m*) ∘ r.rem.isa(*marble*, *for_exterior_m*) ∘
r.rem.class(*for_interior_m*, *for_interior_m*= (*ei*∪*i-e*)) ∘
r.rem.class(*for_exterior_m*, *for_exterior_m*= (*ei*∪*e-i*)) ∘
r.rem.attr(*i_colour*, *i_colour* = {(*x*,*y*)·*x*∈ *for_interior_m* ∧
∃ *k* (*k*∈*close_up_photo(k)* ∧ *y*=*get_colour*(*k*))}
r.rem.attr(*e_colour*, *e_colour* ={(*x*,*y*)·*x*∈*for_exterior_m* ∧
∃ *k*(*k*∈*long_distance_photo(k)* ∧*y*=*get_colour*(*k*))}

As a side effect of the above refinement transformation the first two integrity constraints become logically equal to true. The third constraint becomes:

$$(e\text{-}i \cup ei \cup i\text{-}e) = marble$$

and, finally, the *return_colour(x)* operation becomes:

[2] Here, the refinement transformation is given in terms of elementary transformations. Usually, more complex transformations are used. These are defined as composition of elementary ones and permit a more concise expression of the refinement steps.

y ← *return_colour(x)* =
pre $x \in$ *marble*
then if $x \in$ *(i-e)* **then** *y*:= {*get_colour(close_up_photo(x))*}
else if $x \in$ *(e-i)* **then** *y*:= {*get_colour(long_distance_photo(x))*}
else *y*:= {*get_colour(close_up_photo(x))*,
get_colour(long_distance_photo(x))}
end

The applicability conditions for t_{r1} are generated according the rule given in [10] [3]. Stated informally, these conditions are:

- there are no classes in the previous schema with identifiers *e-i*, *ei* and *i-e*;
- the is-a links added are not in the previous schema and they do not generate any is-a loop;
- the removed elements belong to the schema;
- the removed classes have no attributes defined on them;
- the removed elements are redundant; i.e. the information they represent can be derived from the remaining component of the schema.

The above conditions can be easily proved to be verified on the initial schema S_0. This ensures that the output produced by t_{r1} is a refinement of S_0.

We can now begin our characterisation of the different levels of reuse by considering the most simple case: the reuse of the refinement steps.

4 The Reuse of the Refinement Steps

The reuse of refinement steps is the simpler form of reuse. It is useful when the implementation decisions that have been taken during the original design are valid also for the new design. In this case, the transformation may be either re-applied as it is or he can be extended to specify additional implementational details. The extension may be required, for example, for refining the components that have been added to the conceptual schema. When the reuse of the refinement step is chosen then $t_{ri'}$ is defined either as:

$t_{ri'} = t_{ri}$ or
$t_{ri'} = t_{ri} \circ t_{rext}$ where t_{rext} is an enrichment transformation.

Referring to our case design, suppose that the initial schema is changed so to include an additional attribute *tech_char* assigned to the class *marble*. The designer may decide that no refinement of the new attribute is needed. In this case, he can reuse the refinement steps t_{r1} as it. Alternatively, he may decide that he want to move the new attribute from the class *marble* to each of the three subclasses. In this case he can simply build the new refinement step as the composition of t_{r1} and the following transformation:

[3] Actually, the rule cited generates only the last condition since it is able to automatically discharge the others. Here we have chosen to report the complete set of applicability conditions for a better understandability.

t_{rext}= r.add.attr(*e-i_tech.char*, *e-i*) ∘ r.add.attr(*ei_tech.char*, *ei*) ∘
r.add.attr(*i-e_tech.char*, *i-e*) ∘ r.rem.attr(*tech_char*)

where the added attributes are those that replace *tech_char*.

The decision to reuse a refinement step has always to be ratified by proving that the chosen refinement step does not contradict the semantics of the schema it is applied to. In our framework this is expressed by requiring that the applicability conditions of the refinement transformation, evaluated on the new schema, be verified. The next Section discusses how this verification can be done by reusing the proofs done during the previous design.

5 The Reuse of the Correctness Proofs

The aim of this Section is to specify which are the applicability conditions that can be discharged by reusing the proofs done in the previous design.

In this discussion we will restrict ourself to consider only the relations among t_0, S_0 and t_{r1} as defined in Section 3. In the next Section we will see how, by adopting a different view, the whole design can be reduced to a sequence of cases that have the shape that has been just considered. This will permit us to generalise the reusing rules discussed here to the whole design process.

In order to simplify the characterization, it will be assumed that there is no overlapping between t_0 and t_{r1}. This not causes any loss of generality since, in case this overlapping exists, we can think to use, during the replaying of the design, refinement transformations in which the overlapping part has been removed. It will also be assumed that the consistency of $t_0(S_0)$ has been proved and that the previous design is correct.

Let S_0 be defined as $<Cl,Attr,IsA,Constr,Op>$ 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. The set Op always contains an operation *Init* that specifies the schema initialisation. According to the semantics given in [9] the application of t_0 to S_0 has the following effect:

$$t_0(<Cl,Attr,IsA,Constr,Op>) = <Cl \cup AddedCl - RemovedCl,\\ Attr \cup AddedAttr - RemovedAttr,\\ IsA \cup AddedIsA - RemovedIsA,\\ Constr \cup AddedConstr - RemovedConstr,\\ Op \cup AddedOp - RemovedOp>$$

where: *AddedX* and *RemovedX* stands for the set of components of kind X that are, respectively, added and removed from the schema by t_0. In particular, *AddedConstr* comprises the inherent constraints that are associated to the new elements of the schema. In order to reuse a refinement transformation for carrying out a redesign, we must be ensured that the transformation can still be applied to the modified schema. This means that the applicability conditions of the reused transformation alone, or, in case the reused transformation is extended, those of the composed transformation, have to be proved. Let us consider, first, the

case in which there is no expansion, i.e. the case in which we have to prove the applicability conditions $appl_{t_{r1}}(S'_0)$ where $S'_0 = t_0(S_0)$. Let us reason on the type of predicates that compose these applicability conditions. Each type of predicate will be indicated with the same label used in Section 2. Remember that we have assumed that the refinement transformations never add and remove the same schema component.

- `x` $\in$ `Cl`
  The following subcases can be distinguished:
  - x $\in$ AddedCl $\wedge$ x $\notin$ RemovedCl ; the predicate is true.
  - x $\notin$ AddedCl $\wedge$ x $\in$ RemovedCl ; the predicate is false.
  - (x $\notin$ AddedCl $\wedge$ x $\notin$ RemovedCl) ; the predicate has to be evaluated on the schema S_0.
- `x` $\in$ `Attr`, `x` $\in$ `IsA` ; these cases are similar to the previous one.
- `x` $\notin$ `Cl` ; The following subcases can be distinguished:
  - x $\in$ AddedCl $\wedge$ x $\notin$ RemovedCl ; the predicate is false.
  - x $\notin$ AddedCl $\wedge$ x $\in$ RemovedCl ; the predicate is true.
  - (x $\notin$ AddedCl $\wedge$ x $\notin$ RemovedCl) ; the predicate has to be evaluated on the schema S_0.
- `x` $\notin$ `Attr`, `x` $\notin$ `IsA`
  ; these cases are similar to the previous one.
- `Free(E)` $\subseteq$ `SchemaVar`
  ; if t_0 does not remove/add elements then the predicate has to be evaluated on the schema S_0, otherwise it must be proved on the new schema.
- `Constr` $\Rightarrow$ `x=E, Constr` $\Rightarrow$ `Inh`
  ; if t_0 does not remove/add constraints then the predicate has to be evaluated on the schema S_0, otherwise it must be proved on the new schema.
- `Op ref Op`$'$
  ; if Op belongs to RemovedOp then this predicate has not to be proved. In all the other cases, it has to be evaluated on the schema S_0.

This analysis shows that there are conditions that can be evaluated directly through simple checks, conditions that has to be evaluated on the schema S_0 and conditions that have to be proved on the new schema. The second ones are those that have been proved during the previous design. We can thus reuse the proofs already done to discharge them. The actual "re-applicability conditions", i.e. those conditions which need to be proved, are only the third ones.

The following two examples show how the reuse of proofs can be fruitfully exploited. These assume as previous design that given at the end of Section 2.

Example1

The conceptual schema given in Figure 1 specifies that a marble can have associated zero, one or two photographs. Let us imagine that the requirements become stronger so that there has to be at least one close up photograph for each marble suitable for interior use and at least one long distance photograph for each marble suitable for exterior use. In order to be consistent with the new requirements, a new constraint is added to the conceptual schema. This is done through the following schema transformation:

t_0=add.constraint((∀*m*·*m*∈*for_exterior_m* → ∃*k long_distance_photo(m)*= *k)* ∧
(∀*m*· *m* ∈ *for_interior_m* → ∃*k close_up_photo(m)*= *k))*

Let us imagine that in this redesign we decide to reuse t_{r1}. In order to safely do that, we have to verify that its applicability conditions are verified on the new schema. The new applicability conditions differs from those already proved during the previous design only because of the last condition that has a richer antecedent. This difference does change the truth value of this condition. The new applicability conditions can thus be discharged completely by relying on the proofs done previously.

Example 2

As another example, let us suppose that the colour of marbles is not meaningful anymore. All the reference to this information has to be removed from the conceptual schema. The following is the definition of the transformation transformation t_0 that does the job. It cancels the attributes *i_colour* and *e_colour* and the schema components that refer them.

rem.attr(*i_colour, for_interior_m*) ∘ rem.attr(*e_colour, for_exterior_m*) ∘
rem.constr(*e_colour*={(*x,y*)·*x* ∈ *for_exterior_m* ∧
∃ *k* · *(k*∈ *long_distance_photo(k)* ∧ *y*= *get_colour*(*k*))} ∘
rem.constr(*i_colour*={(*x,y*)·*x* ∈ *for_interior_m* ∧
∃*k* · *(k*∈ *close_up_photo(k)* ∧ *y*= *get_colour*(*k*))}
rem.op(*y* ← *return_colour(x))*

Note that, the above schema transformation partially overlaps with the refinement transformation since both remove the *i_colour* and *e_colour* attributes. We can thus restrict ourself to consider a refinement transformation t_{r1} in which the overlapping changes are missing. The applicability conditions of this restricted t_{r1}, evaluated on the new schema, can be completely discharged by reusing those checked during the previous design. This can be easily understood by noticing that t_0 only removes elements and the elements that are added in the refinement transformations are independent from the removed ones.

Let us now consider the case in which the refinement step is expanded to include additional implementational details. A precise treatment of this case requires a reasoning on the semantics of the composition operator. Due to the limitation of space, this semantics has not been included here. We then just describe this case informally.

The refinement transformation which is applied to S_0' is the composition of t_{ri} and t_{rext}. It may happen that:

1. The two transformations contains the same schema modifications. For example, t_{ri} contains a *r.add.class(x,E)* and t_{rext} contains an *r.add.class(x,F)*. As the composition operator is defined only if E and F are equivalent, then it is as if the effect of the second modification be null.
2. The two transformations contains opposite schema modifications. For example, t_{ri} contains a *r.add.class(x,E)* and t_{rext} contains an *r.rem.class(x,F)*. As the composition operator is defined only if E and F are equivalent, then it is as if the effect of these two modifications is null.

3. The two transformations specify disjoint schema modifications.

In the first case, it is possible to replace t_{rext} with another transformation where the overlapping changes have been removed. We can thus restrict ourself to consider only the cases 2 and 3. The situation in these cases is a little tricky since it may happen that, as effect of the composition, the proof of certain applicability conditions of t_{ri} are no longer valid. Consider, for example the case in which t_{r1} contains a *r.rem.attr(a,x,E)* $\circ$ *r.rem.class(x,G)* and t_{rext} contains *r.add.attr(a,x,F)*. Imagine that a is the only attribute of the class x. If we consider t_{r1} only, x can be removed since it has no attribute defined on it. If we compose t_{r1} with t_{rext} this condition is no longer true. An exact analysis of the semantics of the transformations permits to identify the situations like the ones of this example [9]. This allows to characterises completely the extent to which reuse of proofs can be exploited.

6 Reuse of Schemas

The reuse discussed above relieves from the burden of selecting the transformation to be applied for the new design. The redesign, however, is still complex, since all the intermediate schemas has to be generated. In this Section we examine the reuse of schemas to explore when the complete regeneration of the schema can be avoided.

Figure 4 shows two alternative paths for producing an intermediate schema of the new design. The schema S_1' can be generated by following either path1 or path2. If we follow path2, we can take advantage of the already existing schema S_1. A single transformation t_1, applied to S_1, suffices to return the desired schema.

Viewing the redesign as a subsequent application of path2, the redesign process takes the shape depicted in Figure 5.

In order to exploit this form of reuse, the local transformations to be applied to each of the intermediate schemas of the previous design have to be known explicitly. The theorem given below, demonstrated rigorously in [9], specifies this information.

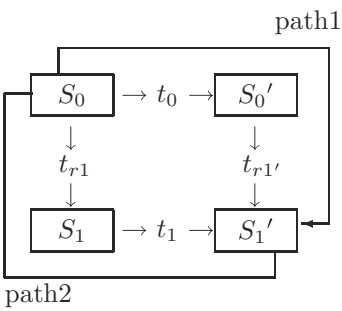

Fig. 4. Alternative paths

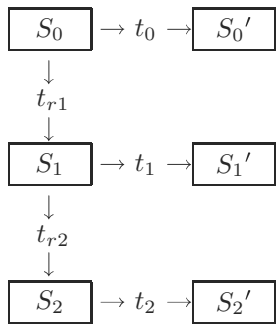

Fig. 5. The reuse of schemas

Before presenting this theorem few preliminaries are needed. First, we will assume that each transformation never add and remove the same schema component. This is not a limitation since, as we have seen before, when this occurs, the whole effect is null. Moreover, we will use the notation $[Subst]Y$ to indicate the expression that is obtained by applying the substitution *Subst* to the expression *Y*. For example, $[x{=}E]R(x)$ is $R(E)$. Finally, we will make use of the following operator between transformations:

Definition 61 (Removal by overlapping) *Let t_i and t_j be either schema or refinement transformations. The removal from t_i of the schema transformations that overlap with those in t_j, (t_i-t_j), returns the following transformation t_k:*

$$
\begin{aligned}
&t_k(<Cl,\ Attr,\ IsA,\ Constr,\ Op>)=\\
&<Cl \cup (AddedCl_{t_i} - AddedCl_{t_j}) - (RemovedCl_{t_i} - RemovedCl_{t_j}),\\
&\quad Attr \cup (AddedAttr_{t_i} - AddedAttr_{t_j}) - (RemovedAttr_{t_i} - RemovedAttr_{t_j}),\\
&\quad IsA \cup (AddedIsA_{t_i} - AddedIsA_{t_j}) - (RemovedIsA_{t_i} - RemovedIsA_{t_j}),\\
&\quad [RemSubst_{(t_i-t_j)}]\ (Constr \cup (AddedConstr_{t_i} - AddedConstr_{t_j}) -\\
&\qquad\qquad (RemovedConstr_{t_i} - RemovedConstr_{t_j})),\\
&\quad [RemSubst_{(t_i-t_j)}]\ Mod_{(t_i-t_j)}\ (Op \cup AddedOp_{t_i} -\\
&\qquad\qquad (RemovedOp_{t_i} \cup RemovedOp_{t_j}))
\end{aligned}
$$

where: if t_i is a schema transformation then $RemSubst_{(t_i-t_j)}$ is the empty substitution and $Mod_{(t_i-t_j)}$ is a null function. Otherwise, if t_i is a refinement transformation, then $RemSubst_{(t_i-t_j)}$ is the set of substitutions dictated by the removal of the classes and attributes effectively carried out by $(t_i - t_j)$, and $Mod_{(t_i-t_j)}$ is a function that modifies each of the schema operation specifications by adding/removing the appropriate updates for any new class or attribute added/removed by $(t_i - t_j)$.

The theorem that specifies how a correct reuse of schema can be achieved is the following.

Theorem 1 (Equivalent paths) *Let $t_{r1}; t_{r2}; \ldots; t_{ri-1}$ be a sequence of refinement transformations that implement a correct refinement from the conceptual schema S_0 to the schema S_{i-1}. Let t_0 be a schema transformation that can be safely applied to S_0 and t_{ri} a refinement transformation that generates a correct refinement of the schema S_{i-1}. Let t_{i-1}/t_i be the transformation that is obtained by taking t_0, by changing its constraint and operation parameters according to the substitutions dictated by t_{r1}, t_{r2}, $\ldots$ t_{ri-1}/t_{r1}, t_{r2}, $\ldots$ t_i, and by removing from the resulting transformation all the transformations that overlap with those in t_{r1}, t_{r2}, $\ldots$ t_{ri-1}/t_{r1}, t_{r2}, $\ldots$ t_i. Let $t_{ri'}$ the transformation that results from the removal by overlapping (t_{ri}-t_{i-1}) and $appl_{t_{ri'}}$ its applicability conditions. Finally, let "$\equiv$" be a relation between schemas. This relation associate two schemas that differ, at most, for the set of explicit constraints. The two set of explicit constraints may have a different shape but they has to be logically equivalent. Under this conditions, the following property holds:*

$$appl_{t_{ri'}}(t_{ri}(S_{i-1})) \Rightarrow t_{ri'}(t_{i-1}(S_{i-1})) \equiv t_i(t_{ri}(S_{i-1}))$$

This theorem specifies under which conditions it is possible to replay the new design following either path1 or path2. Moreover, it describes how the transformation t_i, which specifies how the schemas the document the design process has to evolve, is derived.

As a consequence of the above theorem, the design process can be seen a sequence of steps which has the shape discussed in Section 3. The role of the transformation t_0 is played in this case by the local transformations t_i that are applied to each of the intermediate steps. This permits us to generalize the results on the reuse of the refinement steps that are given in Section 3.1.

Let us now see how the theorem above can be generalised to cover also the case in which the new refinement transformation is built as the extension of a reused one. As before, due to the limited space, we only discuss this reuse case informally. Figure 6a shows a redesign process in which a new refinement step $t_{ri'}$, which is applied to the schema S'_0, is obtained by extending the initial refinement transformation t_{r1} with t_{rext}. Figure 6b shows a different interpretation of the same redesign process. We have said that the refinement transformations are particular schema transformations. Then t_{rext} can be interpreted as a schema transformation that is applied to S'_1 and the reused refinement transformation can be seen as applied to the schema $t_{rext}(S'_0)$. By taking this view, we reduce ourself to the situation that has been considered by Theorem 1. In this case the schema transformation is given by the sequential composition of t_1 and t_{rext}[4]. This new transformation, as soon as the re-design proceeds, has to be submitted to the changes established by the theorem. If a new extension is met, then the schema transformation has to extended again. This behavior continues until the end of the re-design is reached.

The theorem given above has important practical consequences. In particular, it makes possible to evaluate if a change on the conceptual schema requires a change in the sequence of design steps without generating the new sequence. It also permits to generate directly the logical schema without generating the intermediate ones. This can be useful, for example, when there are strict time constraints that impose to make the system to function again as soon as possible. The sequence of schemas can be generated later, as a distinct activity useful for the successive maintenance. As far as this point, notice that, for maintenance purpose, if the design steps have not been changed, it would also be sufficient to maintain only the original design and the sequence of schema transformations that have been effectively applied to the conceptual schema. By exploiting the rule given above, the schemas of the current design can be derived.

Let us now re-examine the examples of redesign presented in Section 4 to see how, according Theorem 1, the intermediate schemas have to evolve.

Example1

This case is trivial. The transformation t_1 is obtained simply by applying to t_0 the change of variables induced by the refinement step t_{r1}:

[4] Actually, the schema transformation is given by the sequential composition of modified versions of t_2 and t_{rext}. The modification is needed because of the difference between the composition operator "$\circ$" and the sequential operator ";".

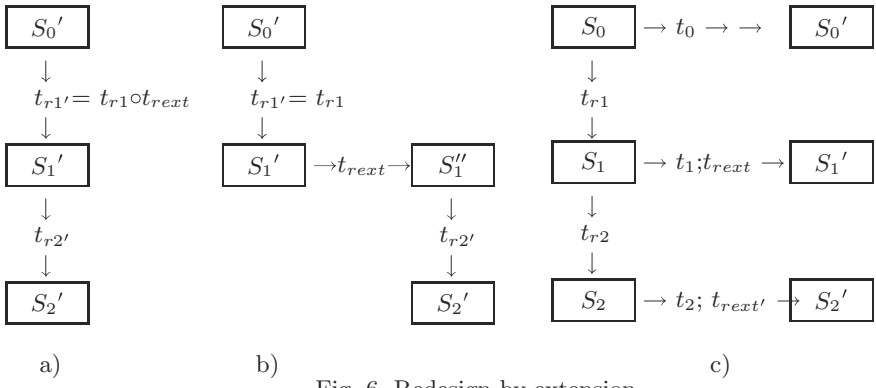

Fig. 6. Redesign by extension

t_1= add.constraint($\forall\ m \cdot m \in$ *(e-i* $\cup$ *ei)* $\rightarrow \exists\ k$ *long_distance_photo(m)= k)* $\wedge$
($\forall\ m \cdot m \in$ *(i-e* $\cup$ *ei)* $\rightarrow \exists k$ *close_up_photo(m)= k))*

Example 2

The new schema transformation is obtained in this case by removing the overlapping transformations and by making the appropriate change of variables:

t_1= rem.constr(*e_colour*=$\{(x,y) \cdot x \in$ *(ei* $\cup$ *e-i)* $\wedge$
$\exists k\ k \in$ *long_distance_photo(k)* $\wedge$ *y=get_colour(k))*$\}$ $\circ$
rem.constr(*e_colour*=$\{(x,y) \cdot x \in$ *(ei* $\cup$ *i-e)* $\wedge$
$\exists k\ k \in$ *close_up_photo(k)* $\wedge$ *y=get_colour(k))*$\}$ $\circ$
rem.op(*y* $\leftarrow$ *return_colour(x))*

7 Concluding Remarks

This paper has discussed the exploitation of reuse during the replaying of schema design. In particular, it has specified precisely when the refinement steps, the proofs and the schemas of the previous design can be reused. It also has made clear which modifications are needed to generate the new design.

This precise definition is a necessary step towards the construction of maintenance environments that support an automatic, or at least assisted, replay of the schema design. As the framework that has been illustrated is formalised in terms of B-Method concepts, we are exploring the possibility of building an environment on top of the tools available to discharge proofs in B.

Another on going activity concerns the exploitation of the results illustrated in other database design frameworks. We have experimented the use of the technique illustrated in carrying out the maintenance of the MIAOW system multimedia database [12]. 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 design of this database consists of a sequence

of OMT-like schemas [16]. Each schema in the sequence is generated by ad-hoc transformations. The final schema has been directly mapped into a Informix Dynamic Server schema [13]. This experience suggested improvements to the reusing technique illustrated and other interesting uses of it. In particular, we found it very useful for avoiding useless re-design. Dynamic Server permits only certain kind of modifications on the schema of an already existing database. This means that not all the modifications required at high level can be satisfied. Each time a modification to the conceptual schema was required, we first generated the last schema of the design, i.e. the schema to be mapped directly into the Dynamic Server one. This permitted us to understand immediately, without re-playing the whole design, if the change at the higher level could be supported by preserving the same design steps. This experience suggested us to explore more in deep the problem and, in particular, to extend the analysis of reuse also the the relations between the schema modifications and the actual database contents.

References

1. Unified Modeling Language, *http://www.rational.com/uml/documentation.html* 1, 1, 5
2. Batini C., Ceri S. and Navathe S.B., *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin Cummings,1992. 1, 5
3. Hainaut J.L., Specification Preservation in Schema Transfromation - Application to semantics and statistics, *Data&Knowledge Engineering*, 16(1), Elsevier Science Publishing, 1996. 1, 1, 5
4. Ramage M. and Bennet K., Maintaining Maintainability, in *Proc. of The International Conference On Software Maintenance*, Bethesda, Maryland, November, 1998. 1
5. Abrial J.-R. *The B-Book*, Cambridge University Press, 1996. 2, 4
6. Victor R. Basili., Viewing Maintenance as Reuse-Oriented Software Development, *IEEE Software*, Vol. 7, No 1, (19-25), January 1990. 5
7. Mikael Lindvall and Kristian Sandahl., Practical Implication of Traceability, *Software Practice and Experience*, Vol.26, No.10, (1161-1180), October 1996. 5
8. Castelli D. and Locuratolo E., A Formal Notation for Conceptual Schema Specification, *Information Modelling and Knowledge Bases V*, IOS Press, (257, 274), 1994. 2
9. Castelli D. A transformational approach to database design and maintenance, *IEI-CNR Tech. Report*, 1999. 2, 8, 11, 11
10. Castelli D. and Pisani S., A Transformational Approach to Correct Schema Refinement, in *Proc. 17th International Conference on Conceptual Modeling*, LNCS, n. 1507, Springer-Verlarg, Singapore, 1998. 1, 3, 7
11. Castano S. and De Antonellis V., Reuse in Object-Oriented Information Systems Development, in *Lecture Notes in Computer Science*, n. 858, (346-358), Springer-Verlarg, 1995. 1, 5
12. B. Biagi and D.Castelli and F. Niccolini and S. Pisani, MIAOW: An Object Oriented Multimedia DB Application on the WWW for the Stone Market, *IEI-CNR Techical Report B4-37*, Pisa, 1996. 14

13. Informix Dynamic Server, http://www.informix.com/informix/products/ids. 15
14. Open Consortium, Proposing an Open standard, *Object Expert* Vol 2. No.1. 1996. 1
15. Ira D. Baxter. Design Maintenance Systems. *Communication of the ACM* Vol 35, No.4, April 1992. 5
16. Rumbaugh J., Blaha M., Premerlani W., E. Frederick and Lorensen W., *Object-Oriented Modelling and Design*, Prentice-Hall, 1991. 15