PHENIX Project - BIKIT / FUNDP
IRSIA/IWONL "Tronc Commun" - Contract 5220

PHENIX Project

DATABASE REVERSE ENGINEERING

FINAL REPORT

Volume III : Technical appendices

Second Version - April 1993

Volume I : General concepts and Introduction

Volume III : Technical appendices

This appendix is a list of additional documents dedicated to the technical descriptions of the PHENIX CARE tools. Due to their technical focus, the documents themselves have not been included in this report, but can be obtained from the FUNDP on request.

PHENIX SYSTEM : PHYSICAL DESCRIPTION

Describes the components of the PHENIX system, and how to run it.

PHENIX SYSTEM : SCREEN EXAMPLES

Screen dumps of the major screens of the PHENIX system for a typical example.

PHENIX SYSTEM : TRANSFORMATIONS SPECIFICATION (FUNCTIONAL)

Describes in details the schema transformations that are available in the PHENIX system from the user view point.

PHENIX SYSTEM : LIST OF THE TRANSFORMATIONS (TECHNICAL)

Describes the schema transformations as methods on the object base.

PHENIX SYSTEM : OBJECT-BASE CONCEPTUAL SPECIFICATION (Version 2)

Describes the conceptual schema of the object base (the PHENIX repository).

PHENIX SYSTEM : NAME PROCESSING (TECHNICAL)

Describes the internal functions for name processing.

PHENIX SYSTEM : SPECIFICATION OF REAL, THE IMPORT/EXPORT LANGUAGE

Describes REAL, an import/export specification language for the PHENIX system. Allows interfacing with other tools.

PHENIX SYSTEM : SPECIFICATION OF THE INTEGRATION PROCESS

Describes the internal functions for schema integration.

PHENIX SYSTEM : OBJECT-BASE Version 2.03, TECHNICAL SPECIFICATION

Describes the internal functions that allows accessing and managing the object-base (repository) contents.

PHENIX SYSTEM : OBJECT-BASE IMPLEMENTATION

Describes the physical implementation of the object-base.

Facultés Universitaires de Namur
Institut d'Informatique

DATABASE REVERSE ENGINEERING

Transformation techniques for Database Reverse engineering

Jean-Luc Hainaut

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1.1 OBJECTIVES

Database design can be modeled as a transformational process based on schema restructuration techniques called schema transformation [BATINI,92] [HAINAUT,93b]. According to an idealized view of database design, the operational description must have the same semantic contents as the conceptual schema. In other words, this process should be semantics-preserving, and therefore reversible. A process P1 is reversible if there exists an identifiable process P2 that, in every case, can reverse the effect of applying P1. In this context, there should exist a process that can recover the conceptual schema of any database from its operational description. This process is called Database reverse engineering.

Most processes are carried out by executing other, lower level, processes. If a process is reversible, then its sub-processes must be reversible as well. Schema transformations are basic processes, i.e. processes that do not call for the execution of other processes, but that are used in other design processes. In order to be reversible, database design must rely on semantics-preserving schema transformations. Consequently, database reverse engineering, in its own turn, must rely on transformations that are the inverse of these forward transformations.

This document is a sort of reference directory of reverse engineering transformations. It presents some major schema restructuration techniques that allow recovering a conceptual schema from the data structures of an operational database. They have been classified according to the constructs on which they apply.

The current material has been borrowed from [HAINAUT,93c], that describes schema transformations for both forward and reverse engineering.

1.2 SCHEMA REPRESENTATION

Any database structure, be it of a conceptual nature, or at the physical level, is expressed as an ER schema. According to the abstraction level and to the target DMS[1], the ER concepts will be interpreted differently. For instance, at the conceptual level, an entity type will be interpreted as a class of conceptual objects, while at the physical level, it will be interpreted as a relational table, an IMS segment type, a CODASYL or COBOL record type, or a TOTAL dataset.

As far as semantics specification is concerned, the supporting ER model includes the following concepts :

- *entity type*, comprising any number (including zero) of attributes;
- *IS-A hierarchy*;
- *relationship type* (called *rel-type* from now on), comprising two or more roles and any number of attributes; a role is taken by one or several entity types (multi-ET role), and is given a cardinality constraint `[min-max]` that states the minimum and maximum number of relationships in which any entity participates in this role;
- an *attribute* is either atomic or compound; an atomic attribute has a domain of values; each attribute is given a cardinality constraint[2] `[min-max]` stating how many values can be associated with its parent object (i.e. entity type, rel-type, compoud attribute); a multivalued attribute (cardinality `max` > 1) can be pure (set of values), bag (multiset of values) or list (indexed set or multiset);
- an entity type can have any number of identifiers, including zero; an identifier is made of attributes and/or roles (i.e. connected entity types)[3];
- a rel-type has at least one identifier made of roles and/or attributes; any role with cardinality `[min-1]` is an identifier; when no identifier is specified or can be deduced, then all the roles of the rel-type form its identifier;
- a multivalued attribute can be given an identifier which is either itself or a subset of its (grand-) children components;
- integrity constraints can be associated with these constructs; let us mention inclusion constraint,

[1] The term DMS, or standing Data Managemlent Systems, encompasses any computerized software that organizes data according to definite data structures, or data model. DBMS, File management systems, programming languages, spreadsheet processors and information retrieval systems are DMS.

[2] Note that this constraint allows for the specification of optional/mandatory attributes as well as single-valued/ multivalued attributes. In addition using this constraint for both roles and attributes simplifies greatly many transformations.

[3] In [BATINI,92], identifiers are called *internal* when they comprise attributes only, *external* when they include roles only, and *mixed* when they are made of both.

referential constraint[4], redundancy constraint, exclusion constraint, mutual coexistence and functional/multivalued dependency.

Specific graphical representations are used to express such constructs. They will be introduces in the following of this document.

1.3 TRANSFORMATION REPRESENTATION

A transformation is made up of two mappings `T` and `t`. Informally, `T` states how to replace the source construct by the target construct, while `t` states how to convert any source instance into a target instance [HAINAUT,91a]. Only mapping `T` will be discussed in this document. However, the concept of semantic preservation requires the specification of mapping `t` as well.

In most case, a transformation <`T1,t1`> will be expressed as in figure 1.1. It corresponds to a *symmetrically reversible* transformation. This is the highest degree in semantics preservation. Its reversibility property is defined as follows :

- there exist an inverse transformation <`T2,t2`>;
- applying T1 to SCHEMA 1 produces SCHEMA 2;
- applying T2 to SCHEMA 2 produces SCHEMA 1;
- any instance s1 of SCHEMA 1 can be converted (through mapping `t1`) into an instance s2 of SCHEMA 2, in such a way that s1 can be converted (through mapping `t2`) into s1 through the inverse transformation;
- any instance s2 of SCHEMA 2 can be converted (through mapping `t2`) into an instance s1 of SCHEMA 1, in such a way that s1 can be converted (through mapping `t1`) into s2 through the inverse transformation.

SCHEMA 1 ⟺ SCHEMA 2

Figure 1.1 - *Symmetrically reversible transformation*

A weaker degree in semantics preservation will be used in some cases. It will be specified as in figure 1.2. It is defined as follows :

- there exist an inverse transformation <`T2,t2`>;

4 This special case of inclusion constraint that is not based on the concept of primary key; it only requires the presence of an identifier in the target entity type.

- applying T1 to SCHEMA 1 produces SCHEMA 2;
- applying T2 to SCHEMA 2 produces SCHEMA 1;
- any instance s1 of SCHEMA 1 can be converted (through mapping `t1`) into an instance s2 of SCHEMA 2, in such a way that s1 can be converted (through mapping `t2`) into s1 through the inverse transformation.

However, any arbitrary instance of SCHEMA 2 is not garanteed to be recovered by applying `t2` then `t1`.

$$\textbf{SCHEMA 1} \quad \Longrightarrow \quad \textbf{SCHEMA 2}$$

Figure 1.2 *- Simply reversible transformation*

In this document, the expression *reversible transformation* should read *symmetrically reversible transformation*, except when specified otherwise.

Precise discussion and specification of the concept of transformation can be found in [HAINAUT,91a] and [HAINAUT,92a] for instance. [BATINI,92] proposes some major techniques in a more informal way.

The presentation that will be developed in this document is based on generic examples. Relying on examples allows for an intuitive approach, while the genericity of examples (using abstract names for instance) allows the reader to apply these examples on actual schemas. A more general presentation should have been proposed (see [HAINAUT,93c] for instance). However the redadability of the document, and therefore its usefulness, could have been questioned.

The document is organized as follows.

Chapter 2 : transformations that apply on attributes.

Chapter 3 : transformations that apply on relationship types

Chapter 4 : transformations that apply on entity types.

Chapter 5 : other transformations

Chapter 2

TRANSFORMATION OF ENTITY TYPE ATTRIBUTES

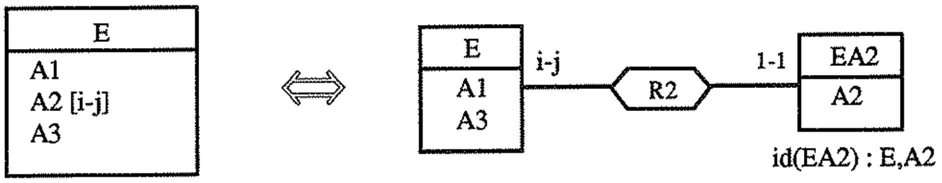
2.1 Introduction

This chapter discusses the techniques that alter the structure of an attribute, or or a subset of attributes, of an entity type.

Though some general techniques can be used to restructure several kinds of attributes, they will be repeated for each of these kinds. However, two techniques are of particular importance, and can be applied to most attributes. They are the *Instance representation* and the *Value representation.* Both can be described as *Replacing an attribute by an entity type.* They will be analysed in this section, but their specific versions will be discussed in each corresponding section.

2.1.1 General technique 1 (Instance representation)

Each instance of attribute A2 is represented by an EA2 entity that is associated via R2 with its E entity. Rel-type R2 is one-to-many. The cardinality `i-j` of R2.E (*right*) is that of E.A2 (*left*). The identifier of EA2 is (E,A2) due to the distinctness of the A values for each E entity.



Specific rules

1. If `i-j` ≡ `i-1`, then R2 is one-to-one, and the identifier of E (*right*) is trivial[1]. It is no longer

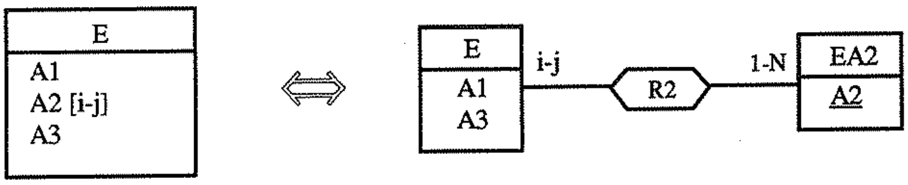
[1] This identifier is no longer minimal, since functional dependency `E` → `EA2` holds in R2. Therefore, component A2 can be trimmed out.

mentioned in the schema.

2. If A2 is an identifier of E (*left*), i.e. if no value of the domain of A2 is allowed to appear in the set of A2 values of more than one E entity, then A2 is an identifier of EA2 (*right*).

2.1.2 General technique 2 (Value representation)

Each value v of the domain of A2 that appears at least once in the A2 value set of some E entities is represented by an EA2 entity. This entity is associated with all the E entities whose A2 value set includes value v.



The cardinality `1-N` of R2.EA2 is worth some comments. The minimum cardinality is 1 since only values that appear at least once in some A2 value sets are represented. The maximum cardinality of R2.EA2 is undefined, and has been set to N (i.e. an arbitrarily large number). In some cases, this last value can be defined more precisely. For instance, if the size of the domain of A2 is known to contain exactly `M` values, then the cardinality of R2.EA2 can be set to 1-M.

Specific rules

1. If `i-j` ≡ `i-1`, i.e. if A2 is single-valued, then R2 is one-to-many.
2. If A2 is an identifier of E (*left*), then the cardinality of R2.EA2 is 1-1 (R2 is many-to-one).
3. If `i-j` ≡ `i-1` and A2 is an identifier of E, then R2 is one-to-one.

2.1.3 Organization of the chapter

The chapter will discuss attribute transformation techniques classified according to the various attribute classes.

- Multivalued attributes, subclassified into pure (i.e. *set*) multivalued, bag and list attributes.
- Single-valued attributes.
- Serial attributes.
- Groups of attributes.

- Compound attributes.
- Optional attributes.
- Reference attributes.

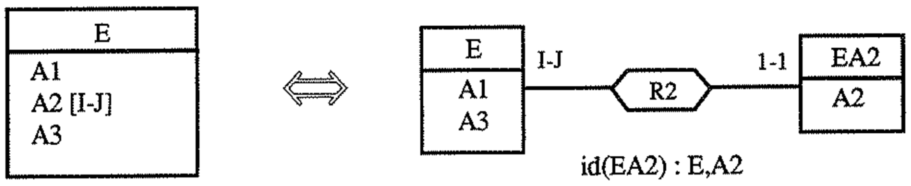
2.2 TRANSFORMATION OF (PURE) MULTIVALUED ATTRIBUTES

2.2.1 Principles

A2 is a multivalued attribute of entity type E if more than one A2 value can be associated with an E entity. If A2 is a *pure* multivalued attribute, the A2 values of any E entity are distinct. In this section, we shall consider that maximum cardinality J is greater than 1. Transforming a multivalued attribute can be suggested to clarify or to simplify a schema.

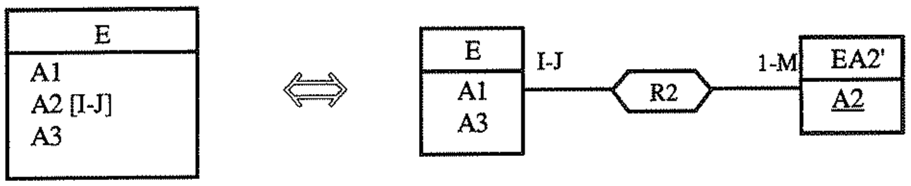
2.2.2 Technique 1 (instance representation)

Each A2 instance is represented by an EA2 entity that is associated via R2 with its E entity. Rel-type R2 is one-to-many.

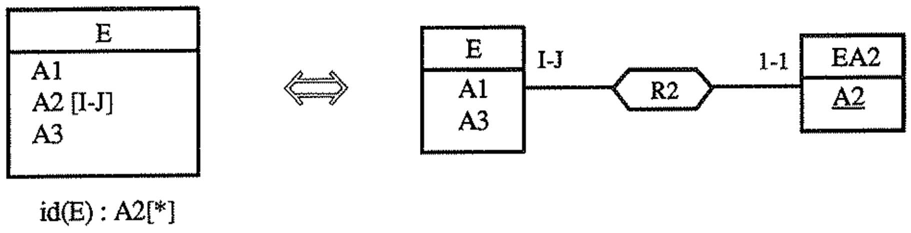


2.2.3 Technique 2 (value representation)

Each distinct A2 value is represented by an EA2' entity that is associated via R2 with all E entities in which it appears. Entity type EA2' appears as a *dictionary* of the A2 values (limited to those values that actually appear in E). Rel-type R2 is many-to-many.



If A2 is an identifier of E (i.e. if each A2 value is associated with only one E entity), R2 reduces to one-to-many.



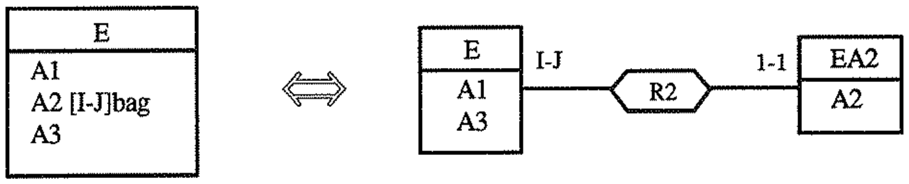
2.3 TRANSFORMATION OF BAG ATTRIBUTES

2.3.1 Principles

A2 is a *bag* (or *multiset*) attribute of entity type E if it is multivalued (more than one A2 values can be associated with an E entity) and if the same value can be associated **more than once** with an entity. In principle, the order of these values is irrelevant (in which case it would be a list attribute). Bag attributes are available in object-oriented DBMS for instance.Transforming a bag attribute can be suggested to clarify or to simplify a schema. It can be suggested when duplicate values are not desirable (bags are non-set-theoretic constructs).

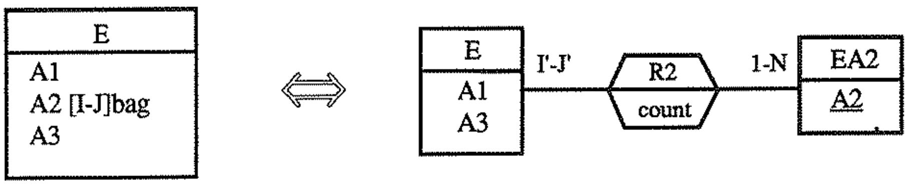
2.3.2 Technique 1 (instance representation)

Each A2 instance is represented by an EA2 entity that is associated via R2 with its E entity. R2 is one-to-many. EA2 has no identifier, except when A2 is declared with no duplicates.



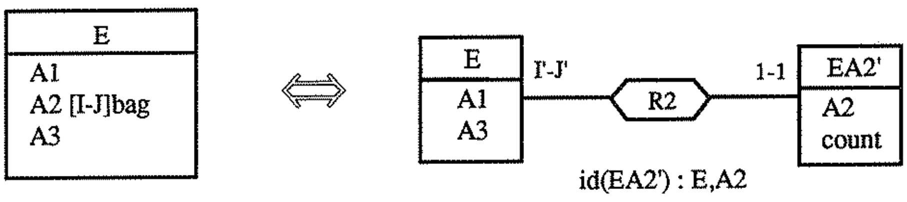
2.3.3 Technique 2 (value representation)

Each distinct A2 value is represented by an EA2 entity that is associated via R2 with its E entity. Rel-type R2 is many-to-many.



2.3.4 Technique 3 (condensation)

All identical A2 instances of an E entity are represented by an EA2 entity that is associated via R2 with the E entity. EA2 specifies the common value of these instances together with their number of instances. Rel-type R2 is one-to-many.



2.4 TRANSFORMATION OF LIST ATTRIBUTES

2.4.1 Principles

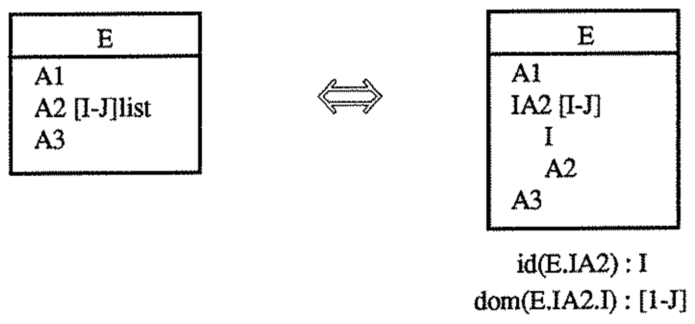
A2 is a list attribute of entity type E if it is multivalued, if the same value can be associated more than once with an entity (except when explicitly prohibited), if the order of these values is significant, and if an element of the list can be referred to by its position[2]. List attributes are the most popular implementation of multivalued or bag attributes. They are proposed in most programming languages : **occurs [indexed]** clause in COBOL, **strings** and **arrays** in PASCAL and BASIC, **arrays** in C,

2 To be more precise, this structure should be named *indexed list*.

lists in LISP (though generally unindexed) and PROLOG. Transforming a list attribute can be suggested to clarify or to simplify a schema, and particularly to express the semantics underlying a physical construct.

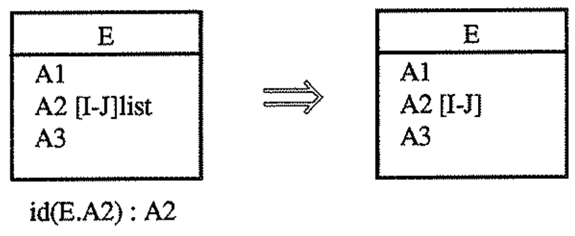
2.4.2 Technique 1 (explicit indexing)

Each A2 instance is represented by an IEA2 compound value that groups this instance with an I attribute value that acts as an index. The origin list can be recovered by sorting the IA2 values within each E entity by their ascending values of I.



2.4.3 Technique 2 (replace by a multivalued attribute)

When it can be proved that the A2 values are distinct for any E entity, and that the sequence is immaterial, this list attribute is merely the implementation of a multivalued attribute. This transformation is not strictly reversible since the ordering structure is lost.



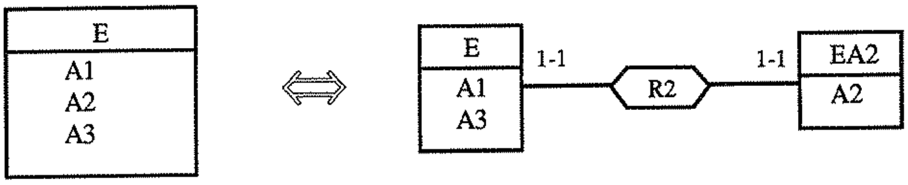
2.5 TRANSFORMATION SINGLE-VALUED ATTRIBUTES

2.5.1 Principles

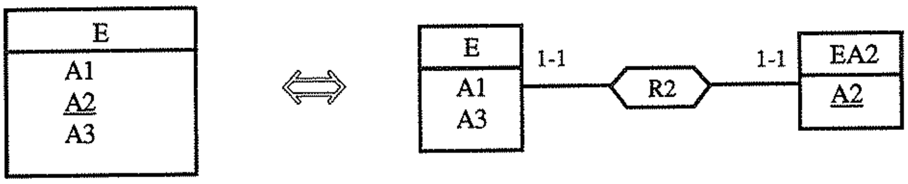
The transformations of an attribute A2 that will be presented are based on the representation by an EA2 entity, either of each distinct values of A2, or of each instance of A2. Since optional attributes are discussed in another section, we shall analyse mandatory attributes only. It is clear, however, that transforming optional attributes and transforming mandatory attributes are special cases of a more general transformation.

2.5.2 Technique 1 (instance representation)

Each A2 instance is represented by an EA2 entity that is associated via R2 with its E entity. Rel-type R2 is one-to-one. EA2 has no identifier. If A2 is an identifier of E (*left*), then A2 must be the identifier of EA2 as well (*right*).

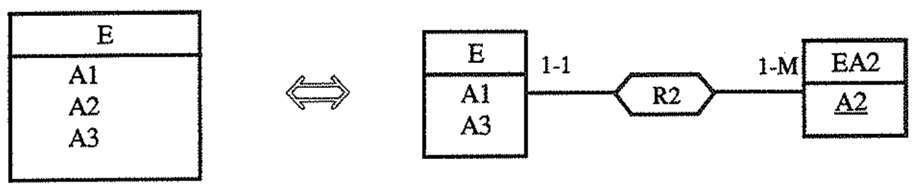


In addition, if A2 is an identifier of E (*left*), then A2 must be the identifier of EA2 as well (*right*).

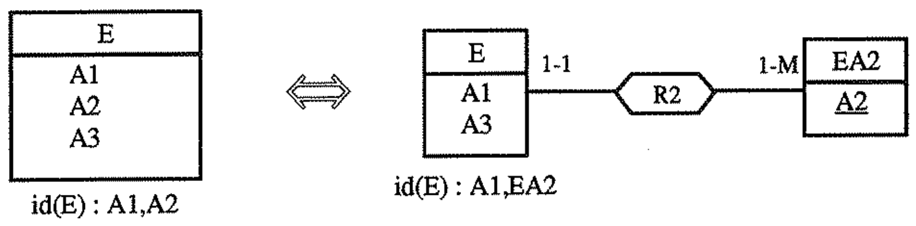


2.5.3 Technique 2 (value representation)

Each distinct A2 value is represented by an EA2 entity that is associated via R2 with all E entities in which it appears. Entity type EA2 appears as a dictionary of the A values (limited to those values that actually appear in E). Rel-type R2 is one-to-many.

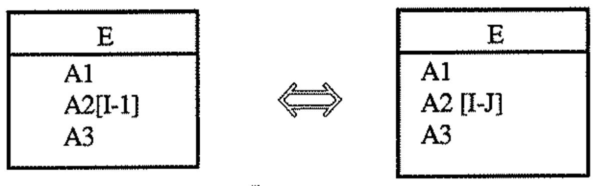


In addition, if we consider that A2 is a component of a multi-attribute identifier, this identifier must be converted in the transformed schema.



2.5.4 Technique 3 (convert string to multivalued attribute)

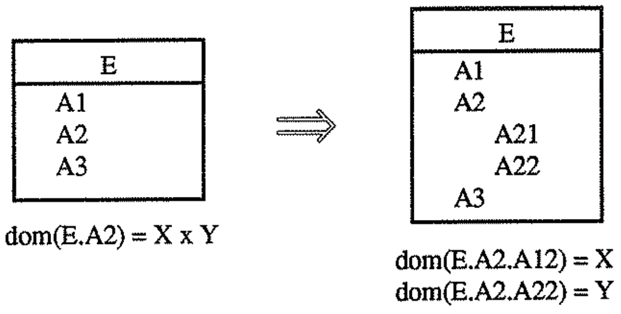
Through this transformation, a string attribute is interpreted as the concatenation of a list of similar attribute values. The anonymous attribute is replaced by a multivalued attribute.



dom(E.A2) = dom(*left(E.A2)*)^J

2.5.5 Technique 4 (convert string to compound attribute)

Through this transformation, a string attribute is interpreted as the concatenation of a list of heterogeneous attribute values. The anonymous attribute is replaced by a compound attribute.



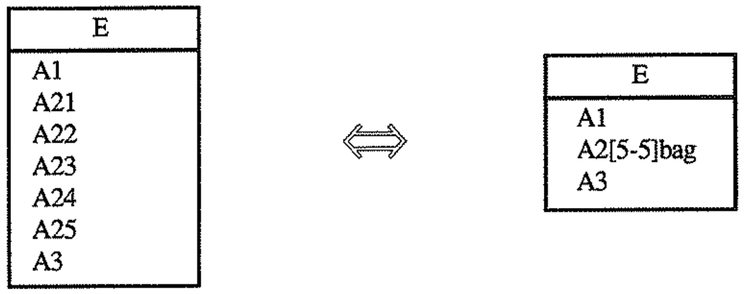
2.6 TRANSFORMATION OF SERIAL ATTRIBUTES

2.6.1 Principles

We call *serial attributes* a list of sibling[3] attributes that have the same domain, and whose semantics present similarities. Their names suggest an indexing structure or identical or similar meaning. These names may suggest the different states of the same phenomenon (e.g. the days in the week, names of the twelve months, department names, etc). The common principle is based on replacing these attributes by one multivalued attribute.

2.6.2 Technique 1 (replace by a bag attribute)

If the attribute values have not to be distinct, and if the names do not suggest indexing or semantic variants, the simplest translation is through a bag attribute.

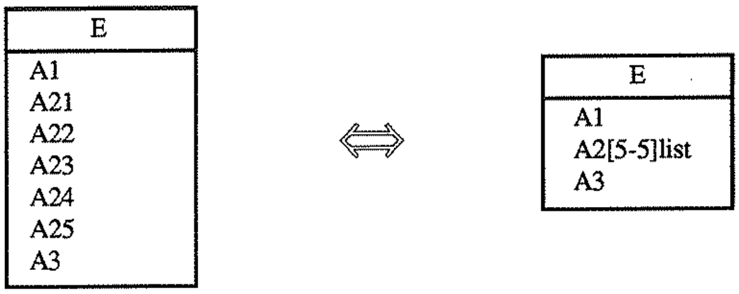


dom(A2i) = dom(A2j), 1 ≤ i,j ≤ 5
names <A21, .., A25> suggest no indexed structure nor distinct semantics

3 I.e. that share the same parent object.

2.6.3 Technique 2 (replace by a list attribute)

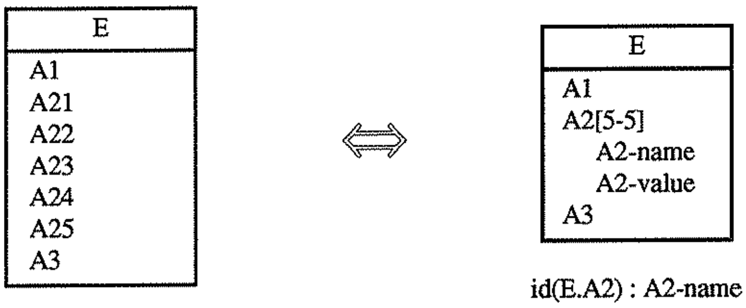
If the attribute values have not to be distinct, and if the names suggest indexing (such as SALES1, SALES, .., SALES12), but no semantic variants, the simplest translation is through a list attribute.



dom(A2i) = dom(A2j), 1 ≤ i,j ≤ 5
names <A21, .., A25> suggest
an indexed structure

2.6.4 Technique 3 (replace by a compound attribute)

If the attribute values have not to be distinct, and if the names suggest semantic variants (such as department names), the simplest translation is through a multivalued compound attribute grouping the origin value (A2-value) together with the former attribute name (A2-name).

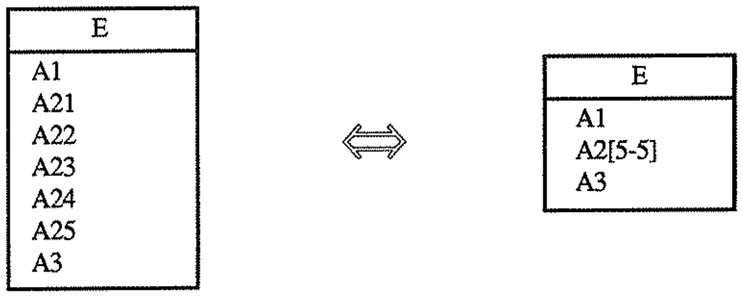


dom(A2i) = dom(A2j), 1 ≤ i,j ≤ 5
names <A21, .., A25> suggest
related but distinct semantics

id(E.A2) : A2-name
dom(E.A2) : {'A21','A22','A23','A24','A25'}

2.6.5 Technique 4 (replace by a multivalued attribute)

If the attribute values are distinct, the simplest translation is through a mere multivalued attribute.



$\mathrm{dom}(A2i) = \mathrm{dom}(A2j),\ 1 \le i,j \le 5$
$i \ne j \Rightarrow e.A2i \ne e.A2j,\ e \in E$

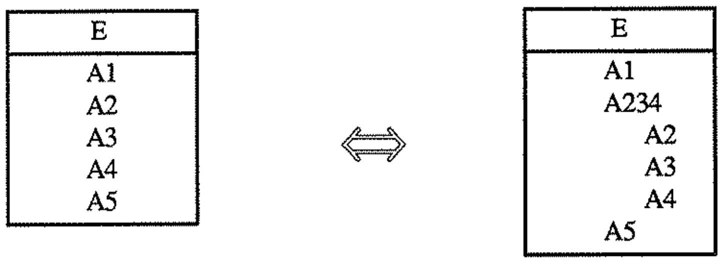
2.7 TRANSFORMATION OF A GROUP OF ATTRIBUTES

2.7.1 Principles

A group of sibling attributes can be grouped as a compound attribute. Their domains need not be the same. The objective is to clarify the attribute structures and to make its semantics more explicit.

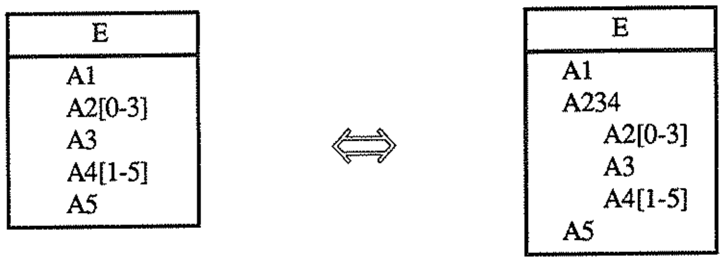
2.7.2 Technique 1 (replace contiguous attributes)

If the attributes are contiguous, the physical layout (offset within the physical record structure) of the new components is the same as that of the origin attributes.

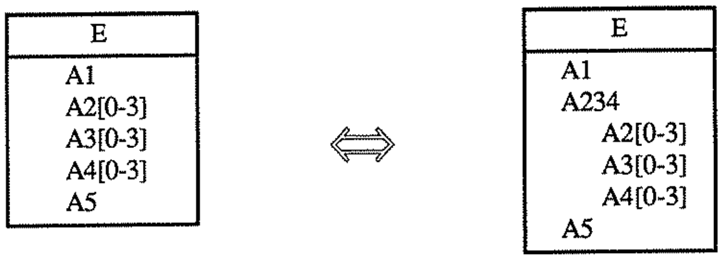


Special case - multivalued attributes

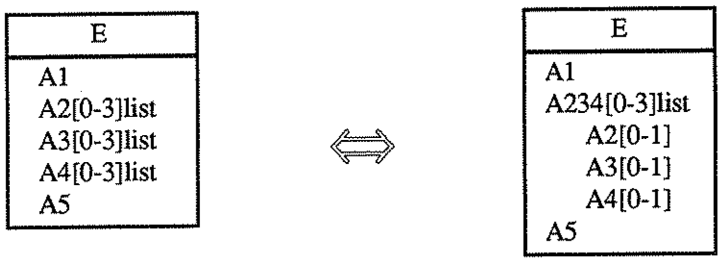
When some of these attributes are multivalued, they keep their cardinality in the compound attribute.



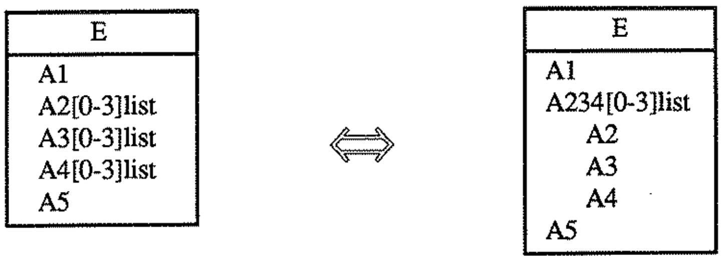
In general, this principle remains valid even when all the grouped attributes have the same cardinality.



Indeed, this common cardinality cannot by associated with the compound attribute in pure multivalued attributes. This would imply a correlation among the attributes that does not exist in the source schema. With list structured attributes, factorizing the common cardinality can be valid. However, since the actual number of values can be different for A2, A3 and A4, these attributes must be optional for compound attriibute A234.

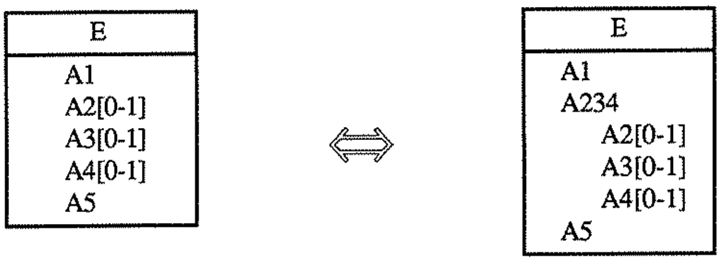


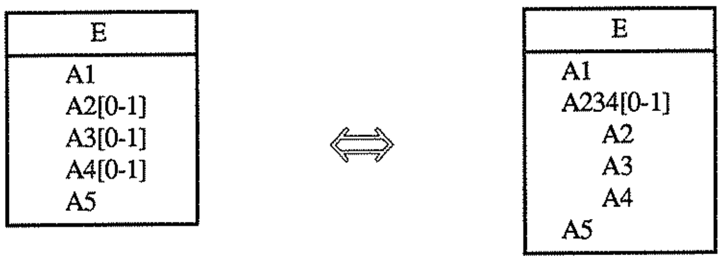
With this transformation, however, the physical layout is disturbed. If this number of values is known to be the same, then the target schema can be simplified as follows :



size(e.A2) = size(e.A3) = size(e.A4), ∀e ∈ E

This discussion is still valid for **optional attributes** :

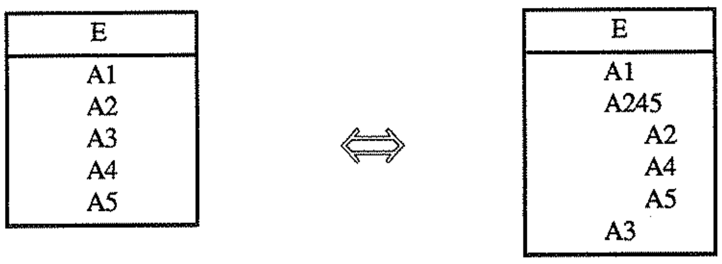




absent(e.A2) ⇔ absent(e.A3) ⇔ absent(e.A4), ∀e ∈ E

2.7.3 Technique 2 (replace non-contiguous attributes)

In that case, the physical layout of the components is disturbed. Otherwise, the discussion developed so far is still valid.



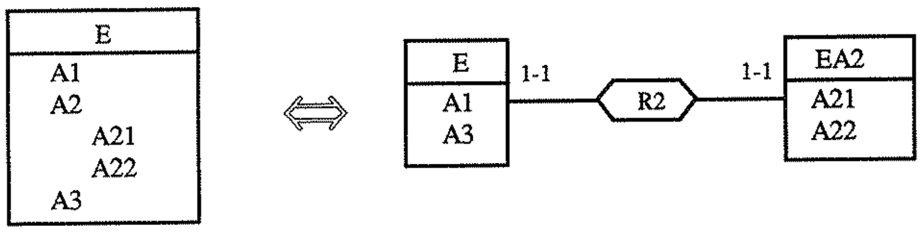
2.8 TRANSFORMATION OF COMPOUND ATTRIBUTES

2.8.1 Principles

A compound attribute value can be perceived as an enbedded entity type

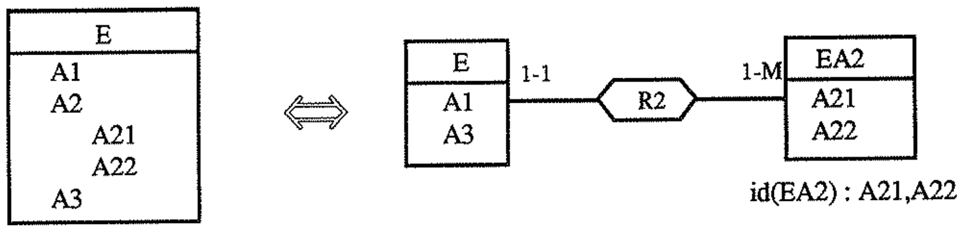
2.8.2 Technique 1 (instance representation)

Compound attribute A2 is transformed into an entity type whose attributes are the components of A2. EA2 has no identifier (except if A2 is an identifier for E).



2.8.3 Technique 2 (value representation)

Compound attribute A2 is transformed into an entity type whose attributes are the components of A2.



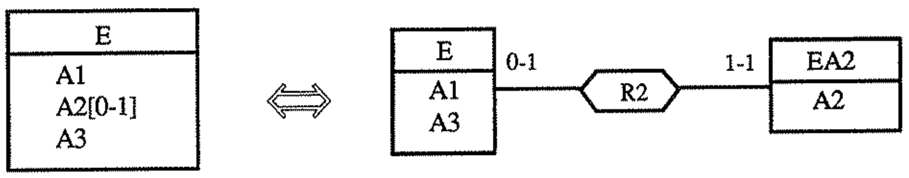
2.9 TRANSFORMATION OF OPTIONAL ATTRIBUTES

2.9.1 Principles

If A2 is an optional attribute of E, then some E entities may have no A2 attribute value. Optional attributes induce some logical and practical problems, known as the *null* value problems. Some authors propose to get rid of these attributes [DATE,86], [CODD,90], [DATE,92]. The transformations consist in replacing A2 either by a mandatory attribute or by an optional role.

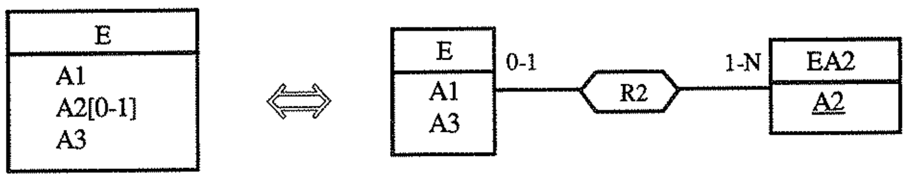
2.9.2 Technique 1 (instance representation)

Instances of A2 values are represented by EA2 entities.



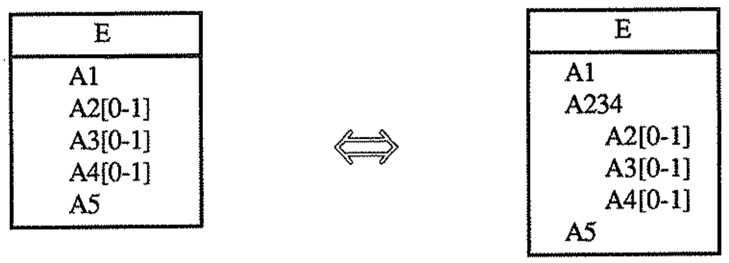
2.9.3 Technique 2 (value representation)

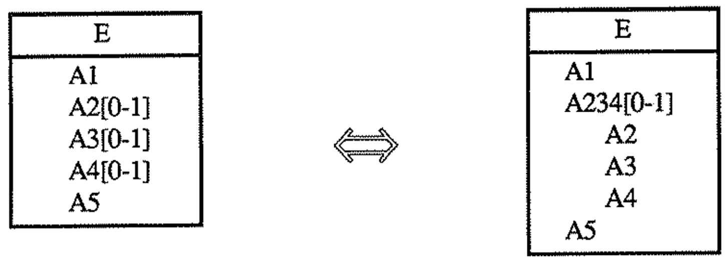
Distinct values of A2 are represented by EA2 entities. Note that if A2 is an identifier of E (*left*), then the cardinality of R2.EA2 is 1-1 instead of 1-N (*right*).



2.9.4 Technique 3 (multiple transformation)

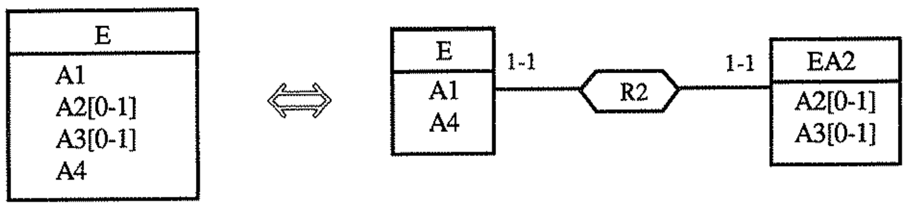
The *grouping transformation* can be applied to optional attributes as discussed earlier. It is recalled herebelow[4].





coexist(A2,A3,A4)

It is then possible to further transform A234 according to techniques related to single-valued attributes. An complete example is given herebelow.



[4] It could be argued that A234 can be optional, since the simultaneous absence of A2, A3 and A4 values is allowed. In this case, one could consider that the A234 value is absent. However, such a representation would be much more complex :

(1) to obtain the value of, say, A2, one have to control two levels of optionality : is there an A234 value ?, in this case, is there an A2 value ?; the complexity of controlling optional values is well known, as testified by the manipulation of *null* values in embedded SQL;

(2) one should specify another constraint, stating that in case of an existing A234 value, it should include a value for at least one of its component (i.e. an existing A234 value cannot be empty).

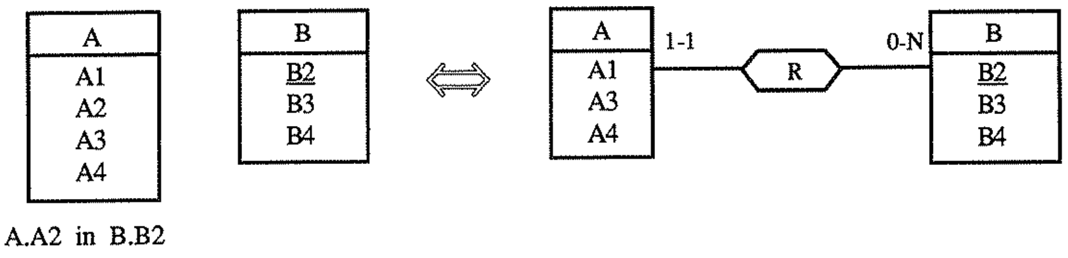
2.10 TRANSFORMATION OF REFERENCE ATTRIBUTES

2.10.1 Principles

Attribute A2 (or the group of reference attributes A2, .., A3) of entity type A is called *reference attribute*(s) if its values are used as references to entities of type B. To be more precise, let us consider that entity type B has an identifier[5] made of attribute B2 (or attributes B2, .., B3). Therefore, at any time, the value of A2, .., A3 of any A entity must be a value of B2, .., B3 of some B entity (except when this A entity has no A2, .., A3 value, in which case this property does not hold). The latter property is called *referential integrity (constraint).* In the model used in this document, a referential constraint is expressed as an *inclusion constraint* where the target attribute (B.B2) is an identifier. Reference attributes are common in relational databases, where they are called *foreign keys*. They are used in standard file structures (e.g. COBOL files) as well. However, in the latter case, the referential constraint is not explicitly declared, and must therefore be elicited through data usage analysis, name similarities or file contents analysis. Finally, network (CODASYL, IMAGE, TOTAL, etc) and hierarchical (IMS) database schemas may include reference attributes instead of relationships.

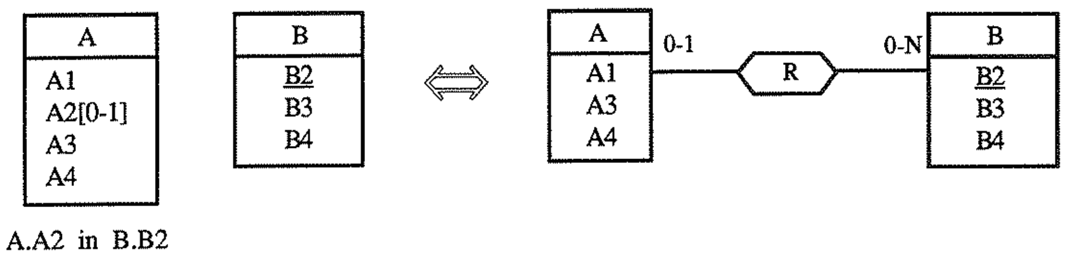
2.10.2 Technique 1 (one single-valued attribute)

Attribute A2 is single-valued. It is replaced by a one-to-many rel-type R between A and B.

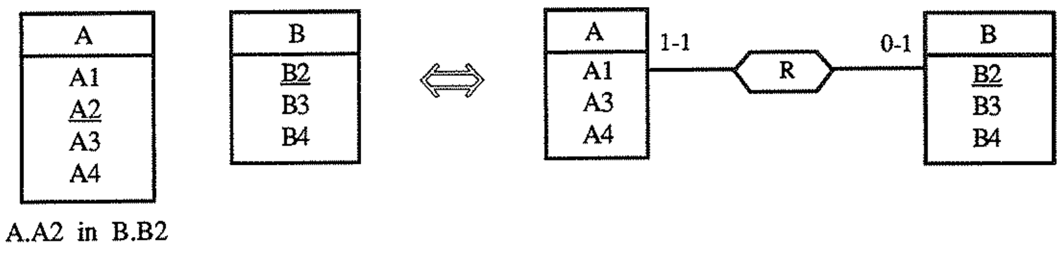


If A2 is optional (cardinality 0-1), the cardinality of R.A is 0-1 as well.

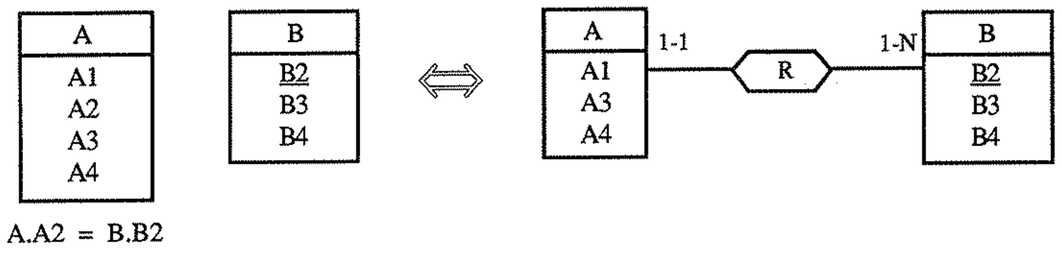
[5] This identifier must be made of single-valued attributes, but need not be a primary identifier.



Attribute A2 can be an identifier of A. In the resulting schema, rel-type R is one-to-one.

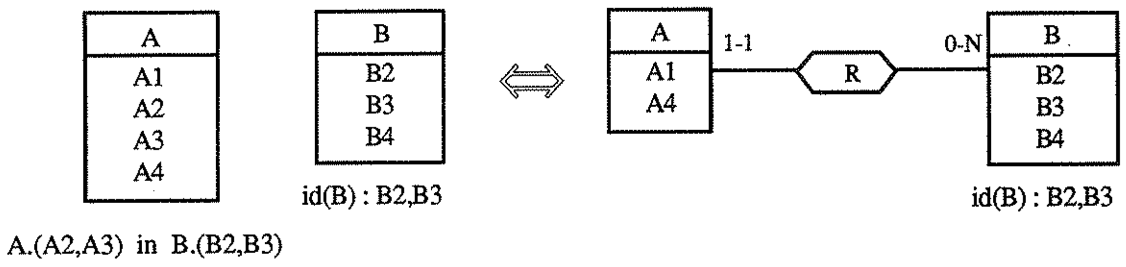


The inclusion constraint can be birectional, i.e. from A.A2 to B.B2 **and** from B.B2 to A.A2. It corresponds to an *equality constraint.* In that case, the cardinality of R.B is 1-N instead of 0-N.



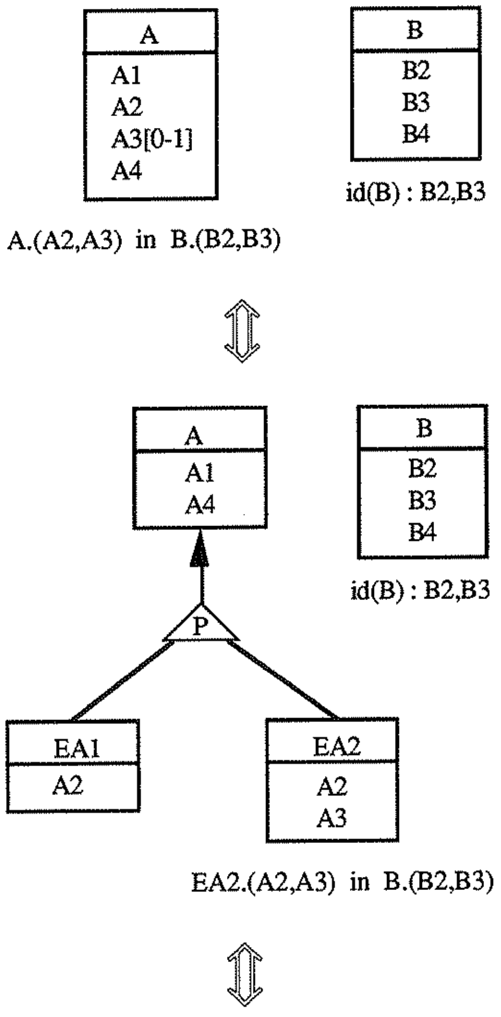
2.10.3 Technique 2 (group of single-valued attributes)

Each attribute A2, .., A3 is single-valued. The principles are the same as above and are illustrated for the most common case.

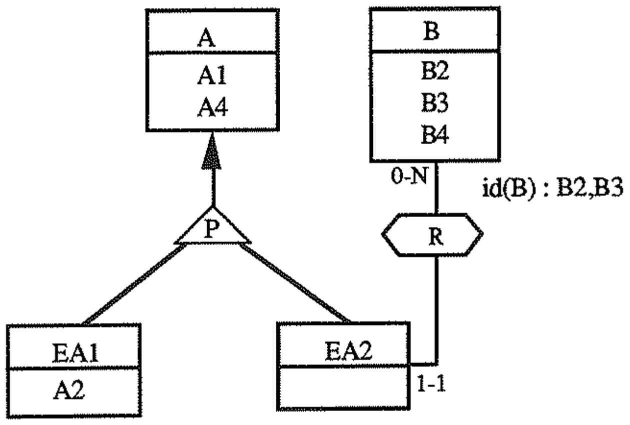


However, the situation can be more complex when attributes A2, .., A3, or some of them, are optional. A entities for which either all these attributes, or none of them, have a value, can be considered as being linked (or not linked) to a B entity. However, the case of an A entity for which some of these attributes have a value while others have no values is more difficult to interprete. Such an A entity seems to be *half-linked* to a B entity, or linked to several B entities.

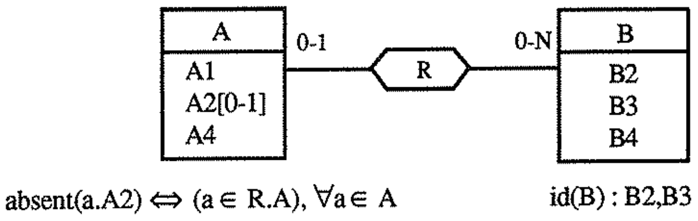
Such situations are not unfrequent since they are allowed in SQL-compliant relational schemas for instance.This problem can be solved by considering that there are two categories of A entities : those that have values either for all A2, .., A3 attributes, or for none of them, and those that have values for a strict subset of these attributes[6]. Only the first category can be transformed through the current technique. This two-step solution is sketched herebelow for a typical abstract example.



[6] In this case, the inclusion constraint *A.(A2,A3) in B.(B2,B3)* should be interpreted as follows :
- if both A2 and A3 values exist, then a corresponding B entity must exist with these values as identifier values;
- if one of them only exist (A2), then the constraint is ignored.

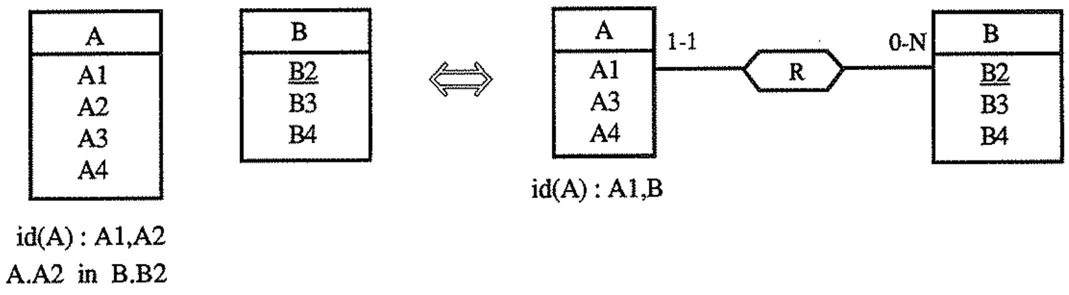


If necessary, the final schema can be further restructured by upward inheritance as follows. Attribute A2 and role A in R are exclusive for A.



The situations in which several attributes, or all attributes of the group are optional are a bit more complex, but can be solved by similar transformations.

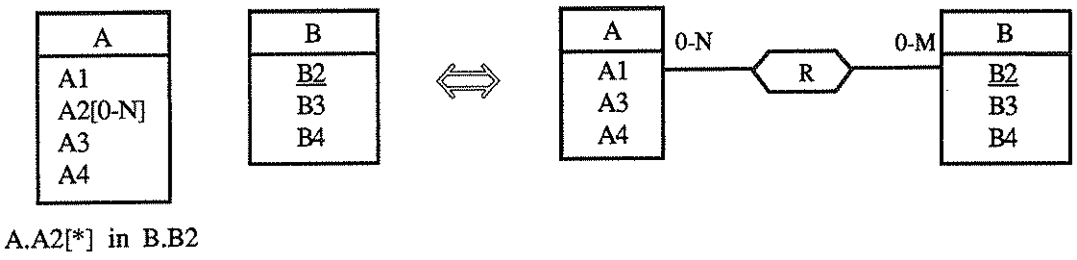
Reference attributes can be a part of an identifier. Their transformation must include the translation of the concerned identifier as well.



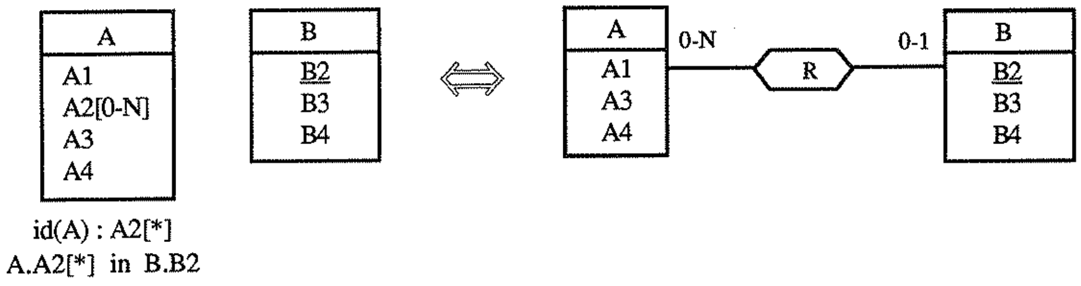
2.10.4 Technique 3 (multivalued attribute)

Attribute A2 is multivalued. Such a structure can be found in COBOL, C or PASCAL data structures, but also in object-oriented languages and DBMS.

Attribute A2 is replaced by a many-to-many rel-type R between A and B. The cardinality of A in R (*right*) is that of A2 in A (*left*).



If A2 is an identifier of A, the cardinality of B in R reduces to 0-1.



Chapter 3

TRANSFORMATION OF RELATIONSHIP TYPES

3.1 INTRODUCTION

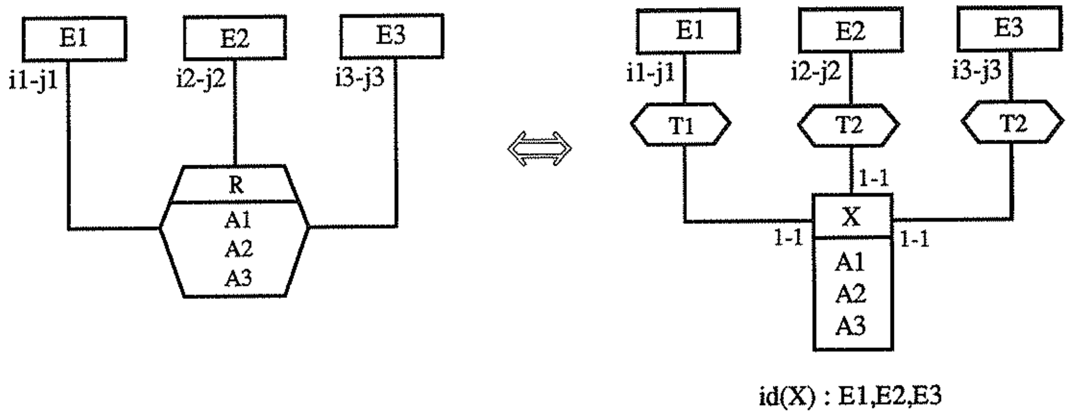
A relationship type can be transformed when it is considered as too complex (in which case it can be replaced by an entity type for instance), when it is unnormalized (it should be decomposed), or when it links two fragments of the same entity type. These techniques are mainly aimed at conceptual normalization.

3.2 TRANSFORMATION OF N-ARY REL-TYPES

A non-binary rel-type has at least 3 roles and may have attributes. Its transformation can be required for several reasons : normalisation, promoting the representation of the underlying concept, weakness of the supporting E-R model.

3.2.1 Technique 1 (Transformation into entity type)

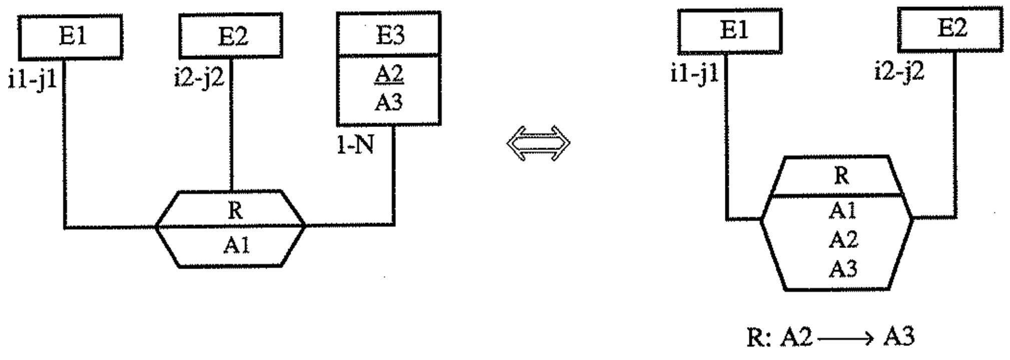
The rel-type is replaced by an entity type that inherits its attributes.



The target schema includes binary rel-types only. The constraints on R, e.g. identifiers, must be translated into constraints on X.

3.2.2 Technique 2 (Reduction of a role by integration)

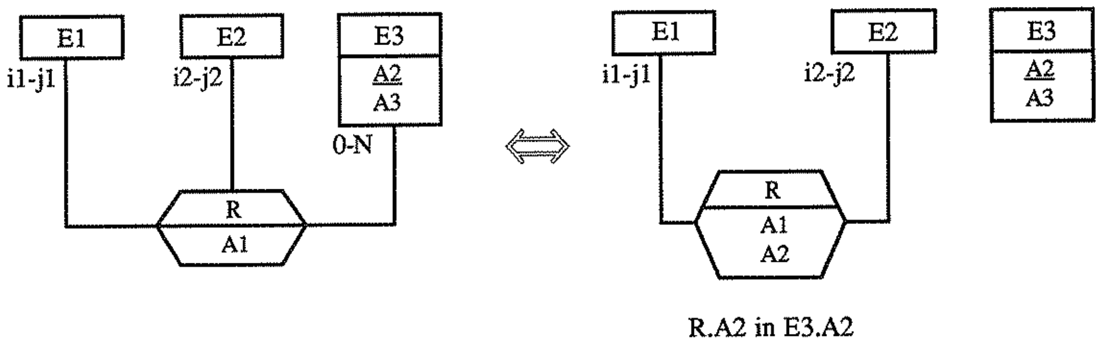
The attributes of a participating entity type can be integrated into the rel-type through a variant of the extension transformation. The corresponding role is replaced by these attributes. This transformation concerns entity types whose cardinality is 1-j, i.e. whose entities cannot exist without participating in R instances.



Note the presence of a derived functional dependency, due to the fact that E3 has two attributes, one of them being an identifier. This pattern is frequent for entity types that represent *dimensional concepts* in the application domain, such as TIME, DATE, COUNTRY, CITY, CATEGORY, TYPE, etc. In these cases, the representative entity type has most often only one attribute that identifies the instances of the concept. This entity type may participate in more than one rel-type. In this case, it is best to transform them simultaneously.

3.2.3 Technique 2 (Reduction of a role by reference)

This technique is similar to the former one, but the role is replaced by a reference attribute to the participating entity type.

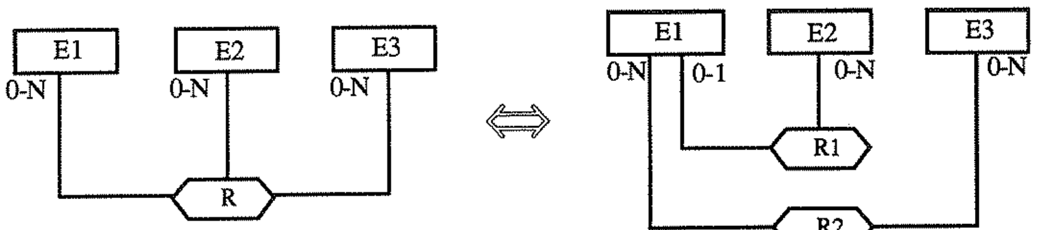


The most frequent application field is also that of dimensional concepts.

3.2.4 Technique 3 (Project/Join transformation)

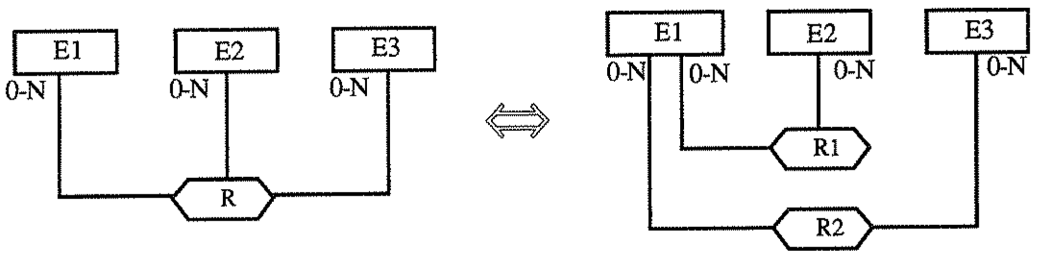
This technique is applicable whenever a non-trivial functional dependency holds in a strict subset of the components (roles + attributes) of R, or if a non-trivial multivalued dependency holds in R[1].

In the following example, the functional dependency implies normalization problems in R. Indeed, R is in 1NF but not in 2NF, due to the *partial dependency*.



R: E1 $\longrightarrow$ E2 $\qquad$ $(e \notin R1.E1) \Leftrightarrow (e \notin R2.E1), \forall e \in E1$

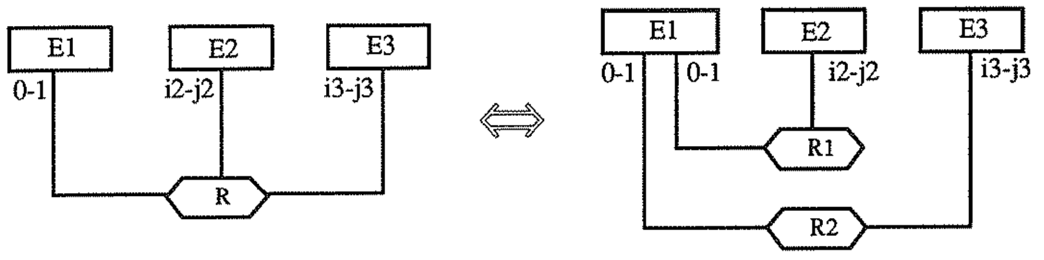
In the next example, R is in 3NF, but not in 4NF due to the multivalued dependency.



R: E1 $\rightarrow\!\!\rightarrow$ E2 $\qquad$ $(e \notin R1.E1) \Leftrightarrow (e \notin R2.E1), \forall e \in E1$

A special case of functional dependency appears when a role has cardinality `i-1`, as discussed in 3.1. In terms of the relational theory, such a schema is normalized. However, it can be decomposed without loss. Another example is depicted herebelow.

[1] This is a mere paraphrase of the preconditions of the P/J transformation.



(e ∉ R1.E1) ⇔ (e ∉ R2.E1), ∀e ∈ E1

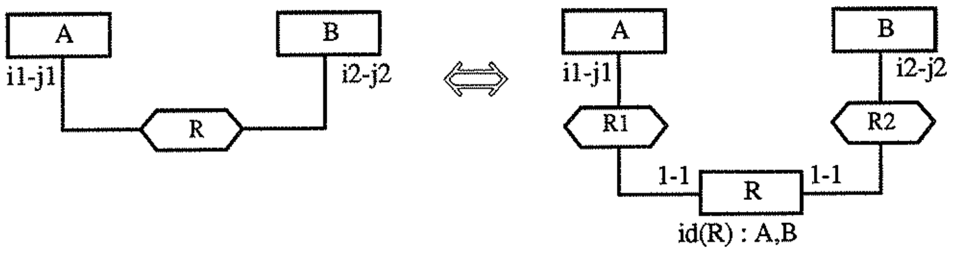
In all these examples, the constraint on the target schema can be dropped if the cardinality of R.E1 is `1-j` instead of `0-j`.

3.3 TRANSFORMATION OF MANY-TO-MANY REL-TYPES

Some E-R models do not allow many-to-many rel-types. They can be transformed into entity types.

3.3.1 Technique 1 (Entity type)

This technique is based similar to technique 3.2.1.

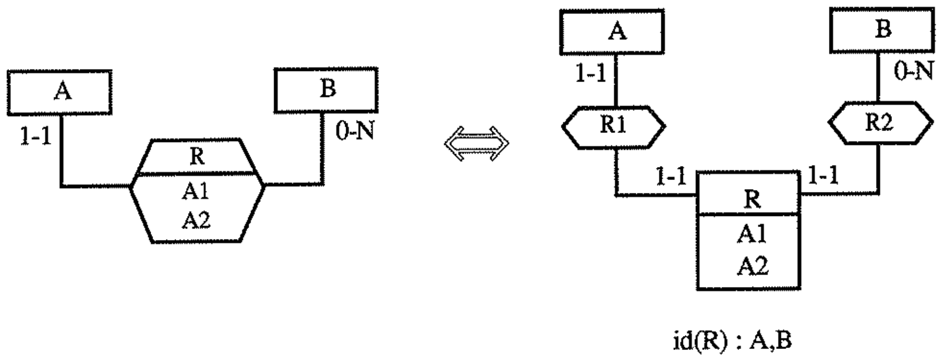


3.4 TRANSFORMATION OF FUNCTIONAL REL-TYPE WITH ATTRIBUTES

Some E-R models do not allow rel-types with attributes. Such constructs can be normalized as follows.

3.4.1 Technique 1 (Entity type)

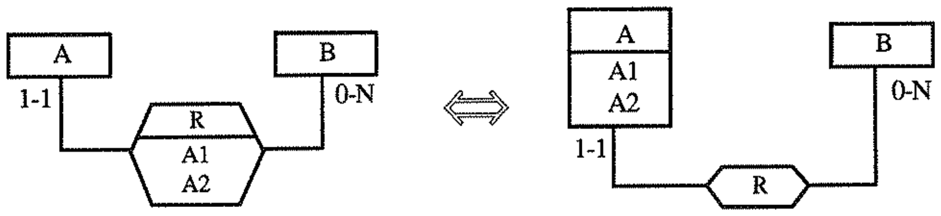
The rel-type is replaced by an entity type that inherits its attributes.



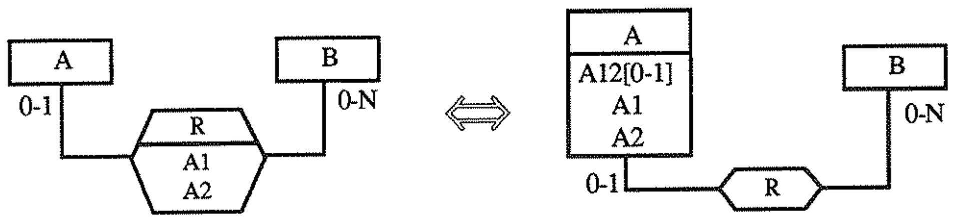
3.4.2 Technique 2 (Attribute migration)

This technique consists in moving the attributes to the participating entity type with cardinality 1-1 or 0-1. We shall distinguish these cases.

If the cardinality of R.A is **1-1**, the attributes keep their cardinalities :



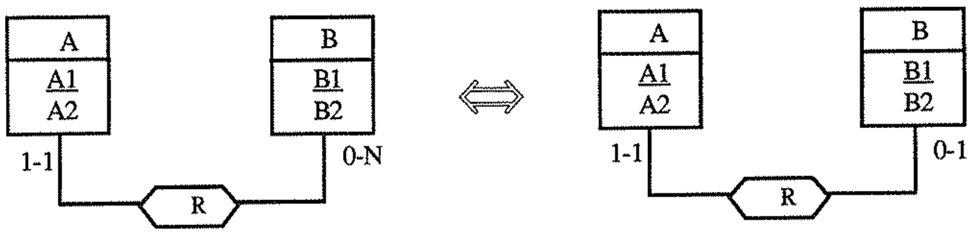
However, if the cardinality of R.A is **0-1**, the attributes become optional, and are submitted to a complex constraint. This practice should be avoided if possible.



absent(a.A12) $\Leftrightarrow$ (a $\notin$ R.A), $\forall$a $\in$ A

3.5 TRANSFORMATION OF FUNCTIONAL REL-TYPES

It may happen that a one-to-many rel-type is used to support a one-to-one rel-type. That can be the case in DBMS models that propose one-to-many rel-types only. CODASYL, IMS, IMAGE and TOTAL are some popular exemples. Relationship type R is therefore declared as many-to-one, but an additional constraint corrects the situation. This constraint will not be controlled by the DBMS, but rather by the application programs or by the dialog manager for instance.

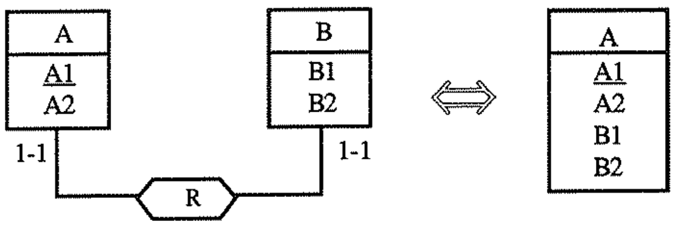


R is one-to-one

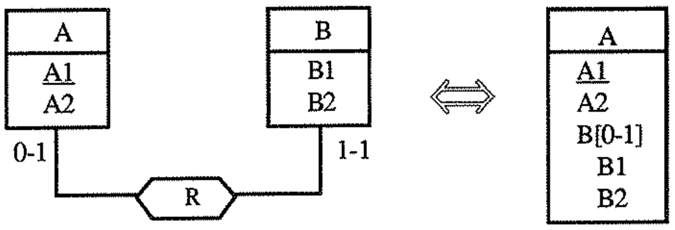
3.6 TRANSFORMATION OF ONE-TO-ONE REL-TYPES

3.6.1 Technique 1 (Entity type merging)

Some one-to-one rel-types are used to link two fragments of a former entity types. The latter has been split, for instance for technical reasons.

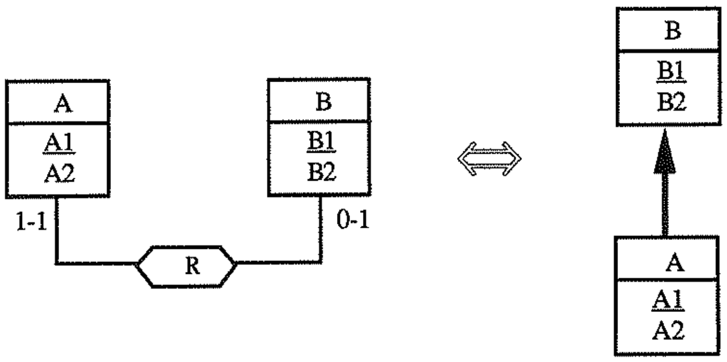


When R.A is an optional role, the inherited attributes can be grouped into an optional compound attribute.



3.6.2 Technique 2 (IS-A relation)

An IS-A relation can be represented through a one-to-one rel-type.

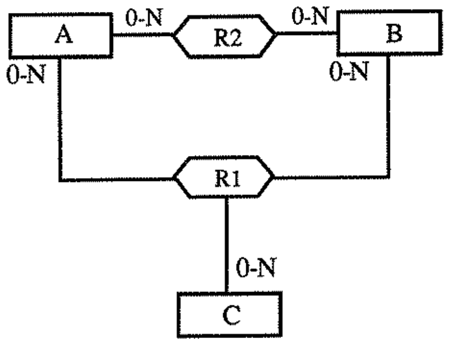


3.7 MULTIPLE TRANSFORMATION IN CASE OF INCLUSION CONSTRAINT

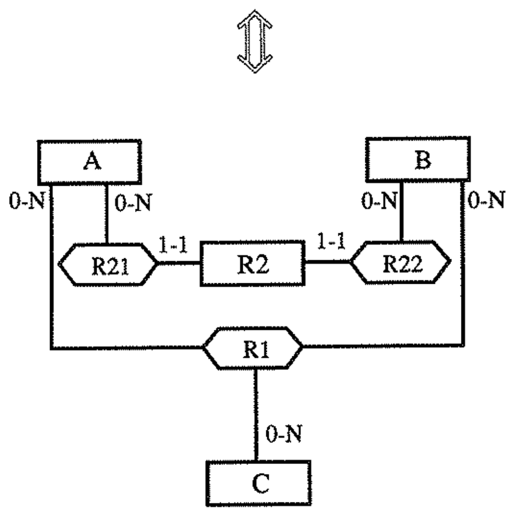
An inclusion constraint states that some part of each instance of rel-type R1 can be derived from some instance of rel-type R2. The transformation aims at factoring these parts by representing them only once.

3.7.1 Technique 1 (Non-binary rel-type simplification)

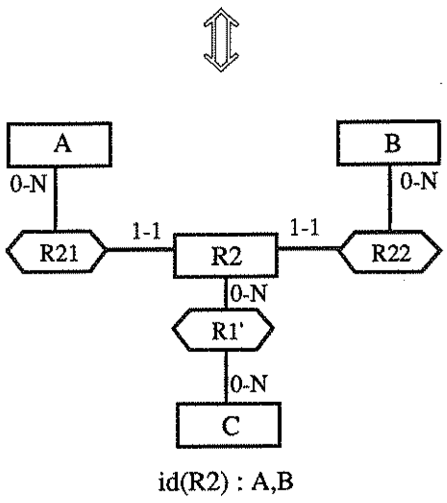
The couple (A,B) from each R1 instance is an instance of R2. By representing this couple by an explicit entity, of new entity type R2, it is then possible to replace the couple (A,B) in R1 by new entity type R2.



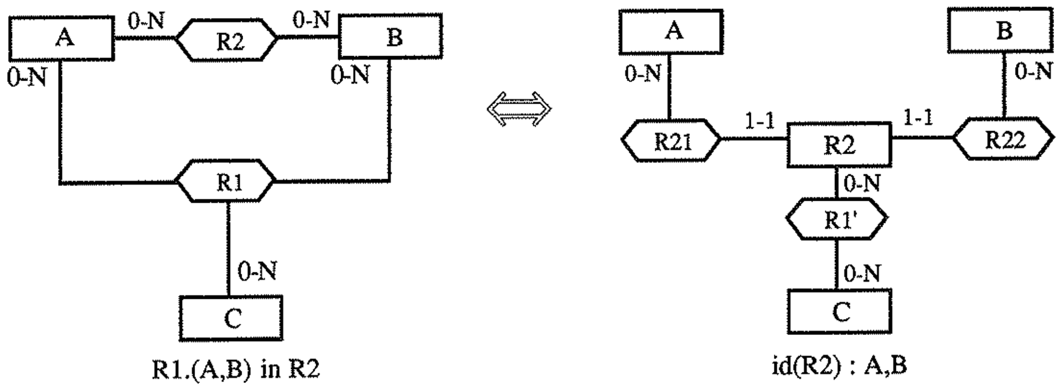
R1.(A,B) in R2



id(R2) : A,B
R1.(A,B) in R21.R22



Hence the global transformation :



In general, this transformation is valid whenever,

- the structures of R1 and R2 overlap,
- and a part of each R1 instance can be derived from an R2 instance.

For instance, it is still valid if R1 and R2 have attributes, or if R1 and R2 have other non-overlapping roles.

Chapter 4

TRANSFORMATION OF ENTITY TYPES

4.1 Introduction

The following techniques will be used mainly in conceptual normalization. They allow processing large and complex entity types, unnormalized entity types, split entity types, relationship and attribute entity type, as well as defining supertypes and subtypes.

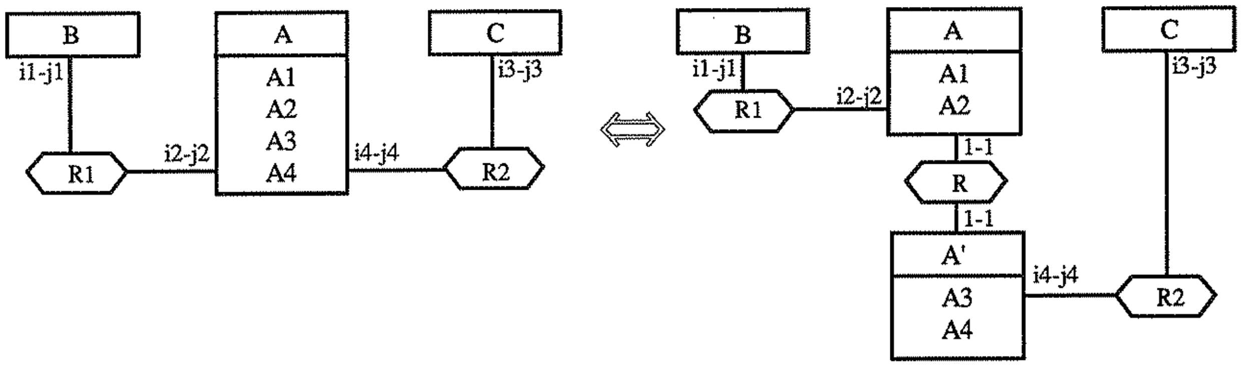
4.2 Complex and large entity type

These techniques aim at replacing an entity type by two entity types among which the attributes and roles of the former are distributed. The source entity type appears as being split into two fragments.

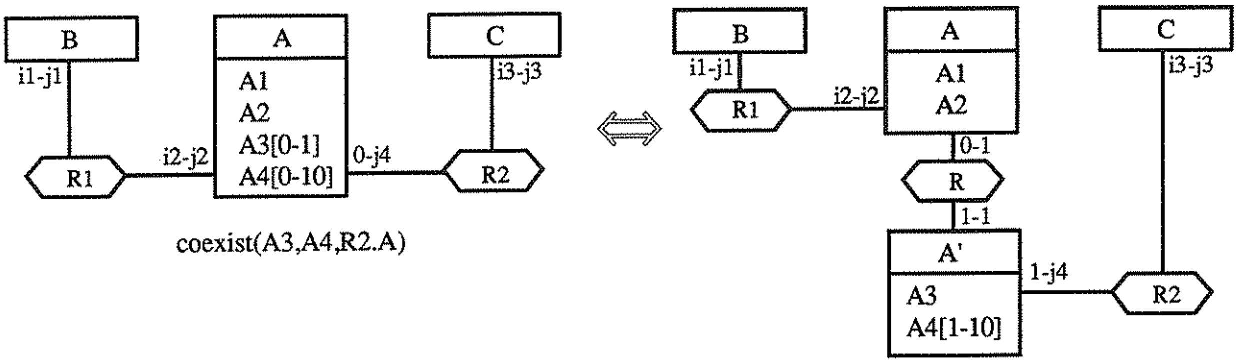
Such practices are frequent when an entity type seems too complex either because it describes too many aspects of an application domain concept, or because it describes several distinct, but related, application domain concepts.

4.2.1 Technique 1 (Splitting through instance representation)

A subset of attributes of entity type A (here A3,A4) and/or roles played by A (R2.A) are extracted and associated to new entity type A'. A and A' are connected through **one-to-one rel-type** R. Both roles of R are mandatory and the cardinality of each attribute/role extracted is left unchanged.



If the components extracted form a *coexistence group* (or a subset of such a group), then the cardinality of R2.A is 0-1 and all the components of A' are mandatory :



If the set E of components extracted from A include components of an identifier of A, this identifier must be modified as follows.

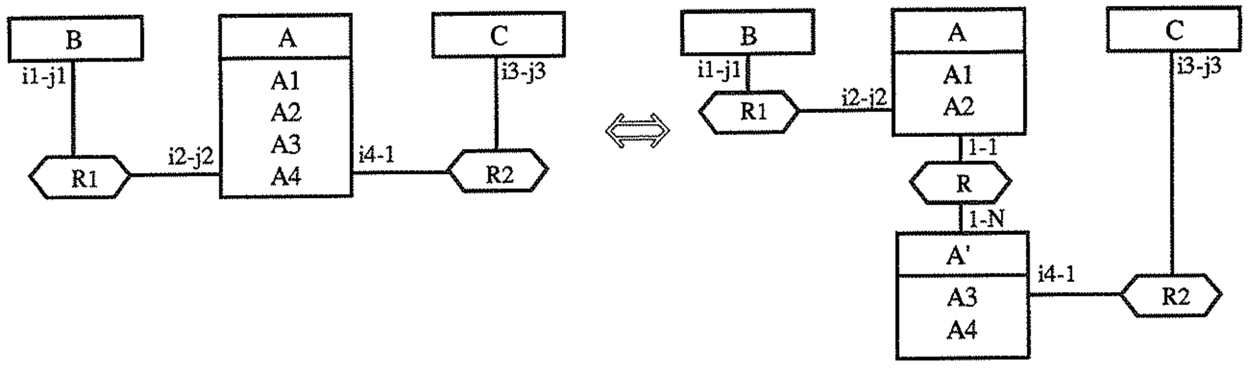
- the identifier is included in E : it is transfered to A', and disappears from A;
- some components of the identifier (but not all) are transfered to A' : each of them is replaced by its target version. For instance, the source identifier "`id(A):A2,A3,R2.C`" is modified into the target identifier "`id(A):A2,R.A'.A3,R.A'.R2.C`".

Similar translation will be carried out for other integrity constraints such as inclusion constraints.

4.2.2 Technique 2 (Splitting through value representation)

According to this technique, the extracted components are represented without duplicates : when their values appear in more than one entity, they will be represented as one A' instance only. The main differences with the first technique are that,

- the extracted components form the **identifier** of the new entity type,
- rel-type R is now a **one-to-many rel-type.**



id(A') : A3,A4,R2.C

The extracted components keep their cardinalities.

If the extracted components form a mutual coexistence group, the minimum cardinalities of R.A, A'.A3, A'.A4 and R2.A' must be revised following the same principles as in technique 5.2.1.

If a functional dependency (such as `A:A3,R2.C` $\longrightarrow$ `A4`) holds among the extracted components, then the source entity type is unnormalized, and should be processed as such (section 5.4).

4.3 Entity types linked by a one-to-one rel-type

These entity types can be merged through transformations that are the inverse of techniques 5.2.1. They consist in integrating the components (attributes and roles) of A' into A through rel-type R. This technique is sometimes called *entity type merging*.

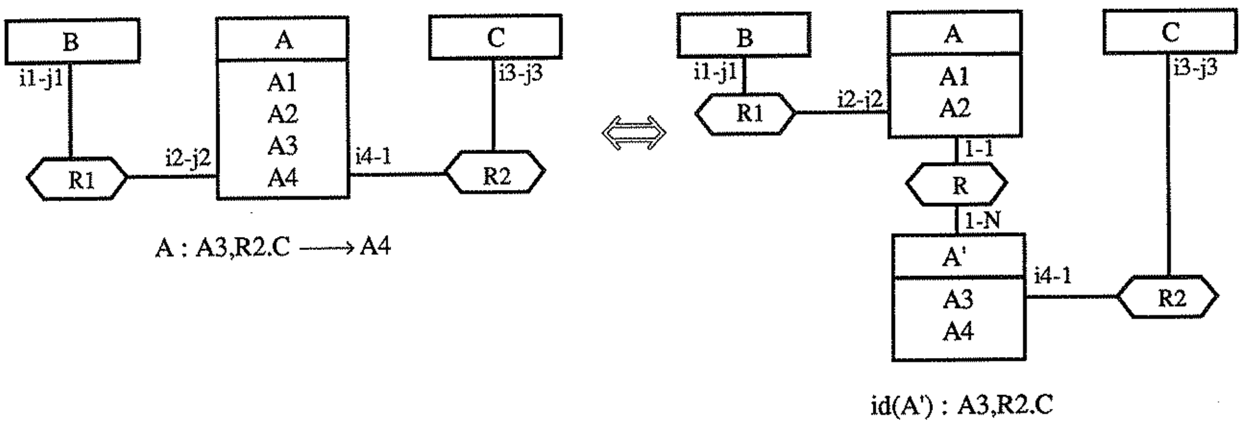
4.4 Unnormalized entity type

An *abnormal* dependency holds in the set of attributes and (i-1) roles of entity type A. A dependency is said to be abnormal whenever its left-hand side (its determinant) is not an identifier of A[1]. We shall distinguish the cases according to the type of dependency : functional or multivalued.

1 It is important to recall that this notion is not strictly equivalent with that of the standard relational theory. Indeed, the concept of normal form can be defined even when no identifier has been defined for the entity type. More on this topic can be found in [HAINAUT,89].

4.4.1 Technique 1 (Elimination of abnormal FD)

This situation can be solved through a technique that is somewhat similar to technique 5.2.2 of *Entity type splitting*. The extracted components are all the components of the left-hand and right-hand sides of the concerned dependency. In case of functional dependency, the new entity type is given an identifier made of the left-hand side components. The new rel-type is one-to-many. Note in particular the cardinalities of R2.A, R.A' and R2.A'.



4.4.2 Technique 2 (Elimination of MD)

The concerned MD are those which,

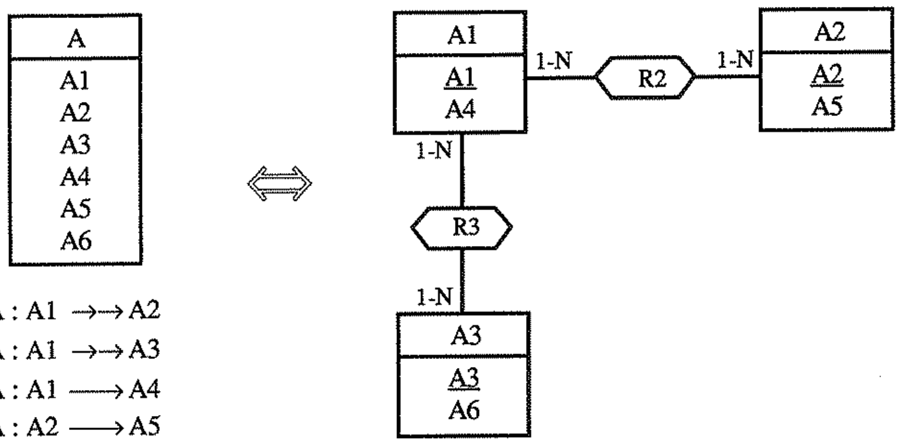
- are not functional (in which case technique 5.4.1 applies),
- are not trivial (in which case no action has to be performed),
- are not the complement of an abnormal functional dependency (in which case technique 5.4.1 applies on this FD).

Let us consider the general following situation, that is expressed as a set of dependencies among the components {C1,C2,..,C5,C6} of A. C1, C2, C3 are either single-valued attributes of A or roles played by a partner of A in a one-to-many rel-type.

The set of dependencies is as follows :

```
A: C1 →→ C2
A: C1 →→ C3
A: C1 ⟶ C4
A: C2 ⟶ C5
A: C3 ⟶ C6
```

Components C4, C5 or C6 may be missing, in which case the corresponding FD's are missing as well. The following schema illustrates the normalization transformation when all A components are attributes {A1,A2,..,A5,A6}.



$A : A1 \twoheadrightarrow A2$
$A : A1 \twoheadrightarrow A3$
$A : A1 \longrightarrow A4$
$A : A2 \longrightarrow A5$
$A : A3 \longrightarrow A6$

This net result can be obtained by,

1. repeatedly applying technique 5.2.2 to each functional dependency,
2. then transforming the resulting entity type A into a rel-type A through technique 5.6,
3. then normalizing rel-type A through technique 3.2.4.

The technique is easily generalized to situations in which C4, C5 or C6 are role components. When C1, C2 or C3 are role components, the roles can be transformed into attributes through techniques 3.2.2 and 3.2.3, processed as propose in this section; the source roles can then be recovered with techniques such as those of section 5.2 or chapter 4. The reasoning is still valid when each component C1 to C6 is made of several attributes and roles.

Of course, the initial dependency pattern on which we have based the reasoning does not cover all the situations that could occur. However, true multivalued dependencies are not that frequent in actual database schemas, so that the above discussion can be considered quite sufficient for real-life problems. However, we shall mention two additional refinements to this technique.

1. An additional FD holds in A in the source schema : `C1,C2` $\longrightarrow$ `C7`
   Action : add attribute C7 to rel-type R2 of the target schema.
2. An additional FD holds in A in the source schema : `C1,C2,C3` $\longrightarrow$ `C8`
   Action : keep the 3-ary rel-type A mentioned in step 2 of the development of the current technique; add attribute C8 to this rel-type; normalize rel-type A, which leads to rel-types R2 and R3, together with rel-type A with attribute C8. Add inclusion constraints.

4.5 Relationship entity type

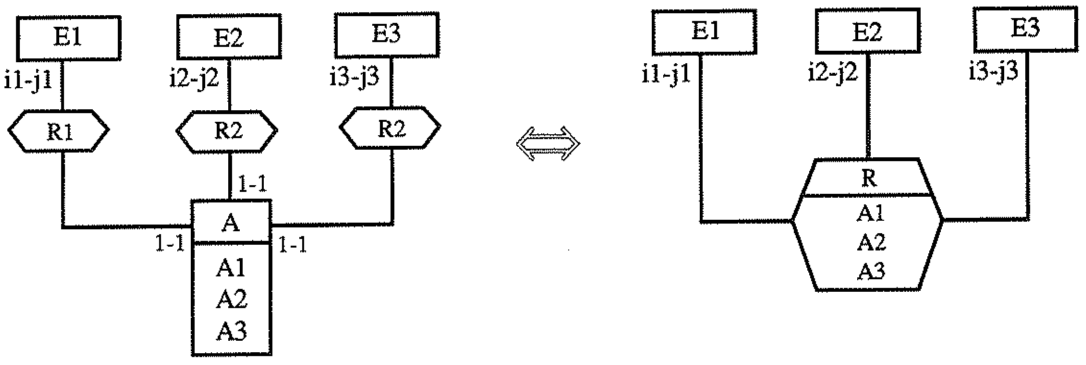
A relationship entity type is an entity type that is perceived as the *representation of a link between other entity types*. According to this view, it can be better to transform it into a pure relationship type. Whether an entity type is a relationship one or not depends on every one's perception, and can be considered as a mere matter of taste[2]. Nevertheless, this concept appears is some E-R models and methodologies. It is also relevant in reverse engineering activities, since few DBMS offers a specific representation of relationship types, but the simplest forms (if at all).All the techniques are the inverse of some variants of the extension transformation described in chapter 3, dedicated to relationship types. Indeed, the extension transformation is aimed at replacing a rel-type by an entity type.

4.5.1 Technique 1 (Transformation into entity type)

This technique is the simplest and most popular one. It consists in representing each entity by one relationship. The source entity type A satisfies the following conditions :

- A participates in N functional, non-recursive, rel-types that have no attributes, with $N \geq 2$; the other roles are played by entity types E1, .., EN;
- A participates in no other rel-types;
- all the roles of A have cardinality 1-1;
- A has at least one identifier, comprising some of these roles and/or some of its attributes.

The target rel-type R inherits these attributes, has N roles played by E1, .., EN which keep their source cardinalities.

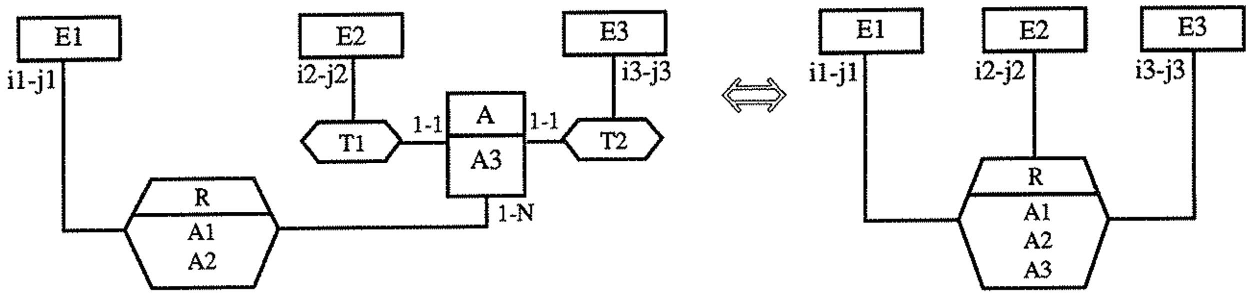


2 It is interesting to observe that both entity type and rel-type representations of relationships can generate the same relational structure. This has often been used to advocate against the undeterminism of the E-R model as compared with the relational model. The argument is valid. However, the greater multiplicity of representations of a single fact type can be considered as an advantage as well, provided a formal proof exists of the equivalence of these representations. This is precisely one of the objectives of this document.

The identifier of source entity type A is converted on target rel-type R. For instance, the source identifier "`id(A):R2.E2,R3.E3,A1`" is converted into target identifier "`id(R):E2,E3,A1`".

4.5.2 Technique 2 (Absorption into existing rel-type)

The source entity type appears as the development of a member of a source rel-type. It is possible to integrate it into this rel-type thanks to a variant of the extension transformation [HAINAUT,91a]. In the following pattern, A can be perceived as the development of role R.A of rel-type R. The transformation consists in replacing A by roles E2 and E3 of R.

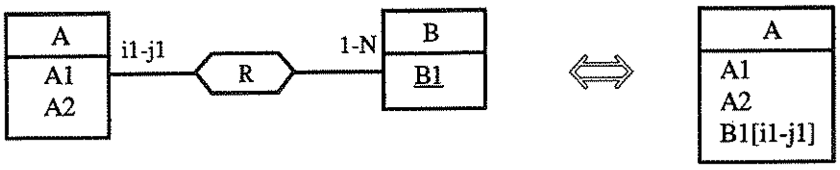


4.6 Attribute entity type

An attribute entity type is an entity type that is perceived as the representation of a simple property of one or several entity types. Typically, it has one attribute (or a small number of attributes) that is its identifier. This can suggest to represent this property by an attribute instead. The discussion about the user's perception is the same as for relationship entity types.

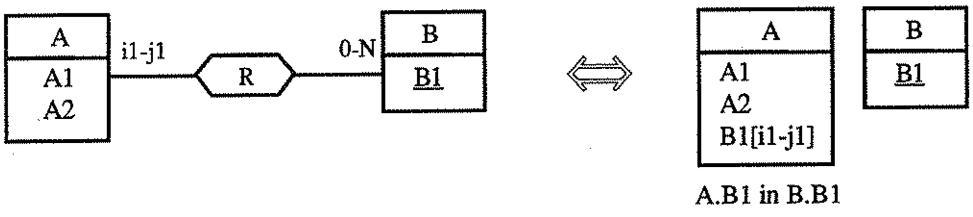
4.6.1 Technique 1 (Entity type merging)

Attribute B1 of entity type B is added to those of entity type A. Its cardinality is that of R.A. If the cardinality of R.B is 1-1, then B1 is an identifier of A.



4.6.2 Technique 2 (Reference attribute)

Though this technique is intended to transform rel-types, it is mentionned here because it appears as a variant of the previous one. It basically is the same as technique 1, but the source entity type B is not removed. It is recommended when role R.B is optional for B (cardinality 0-j2), and when B is shared by several entity types.

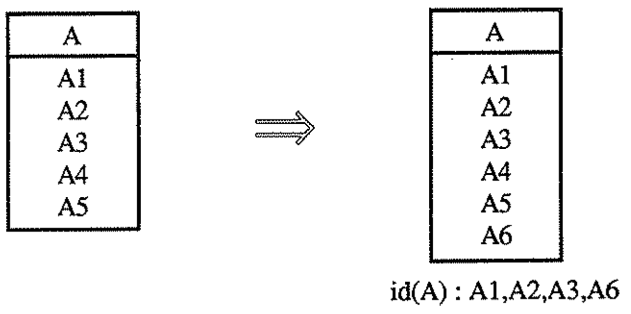


4.7 Entity type without identifier

In general, an entity type need not have an identifier, as testified by many E-R conceptual formalisms and by DBMS such as CODASYL and OO-DBMS. However, some ER-based methodologies may require the existence of an identifier for each entity type.

4.7.1 Technique 1 (Complementation)

This technique consists in augmenting the entity type with a new attribute that forms a natural identifier when appended to a list of existing attributes. This practice is frequent, and has lead to the ubiquitous *sequence number* that populates many schemas. In the following pattern, attribute A6 is added because it forms an identifier with A1, A2 and A3.

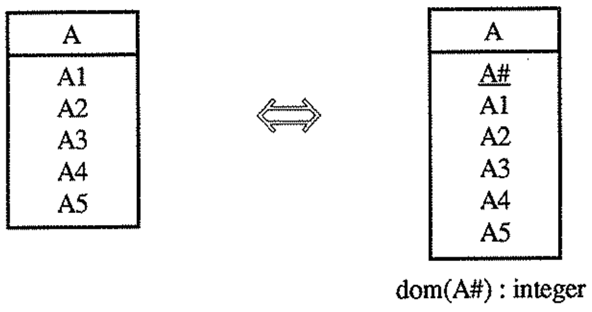


Due to the augmentation of the semantics in the source schema, this transformation **is not reversible**.

As an example, let us consider a source schema that describes *product transfers* from an *origin warehouse* to a *destination warehouse*. In addition, we are interested by the *date* and *volume* of the transfer. However, several transfer may occur on the same date. If we are not interested by transfer identification, but if an identifier is required for technical reasons, we can add attribute `Hour`, that indicates the hour at which a transfer occurs, or attribute `Seq-number`, that indicates the order of the transfer among all those that occurs the same day, between the same warehouses.

4.7.2 Technique 2 (Artificial identifier)

A new attribute is added, and is made the identifier of the entity type. This attribute is given no semantics, and is chosen in such a way that the generation and management of its values is easy.



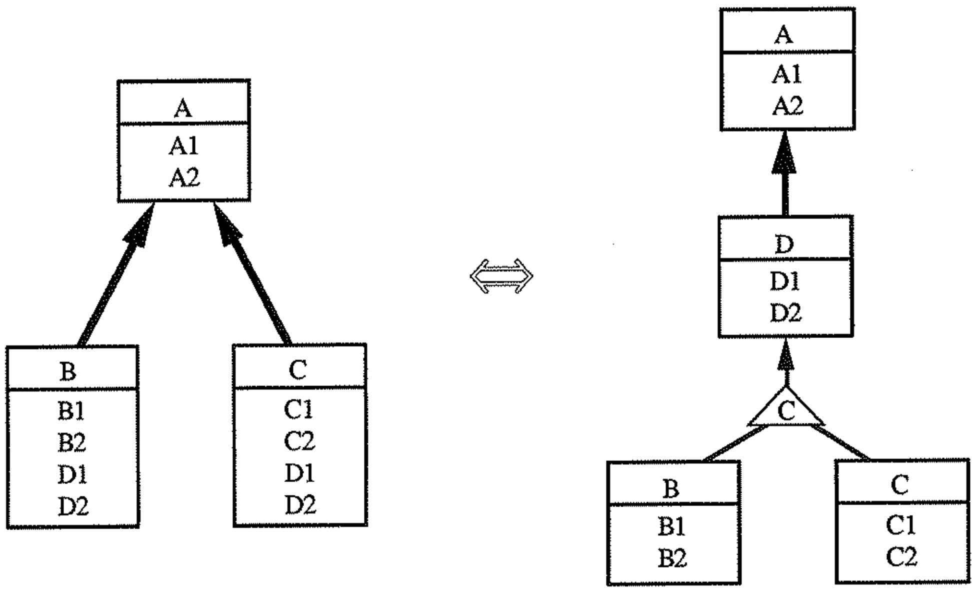
This technique is often preferred to the former one because, since the new attribute bears no semantics, this transformation is reversible.

4.8. Entity types with overlapping structures

The principle is to introduce a supertype of these entity types that collects the common structure. These techniques are useful when simplifying a conceptual schema or when several schemas have to be integrated. Their aim is to gather common attributes, roles and constraints of two or more entity types into a common supertype. The presentation will first tackle the processing of attribute structure of two subtypes of a common supertype.

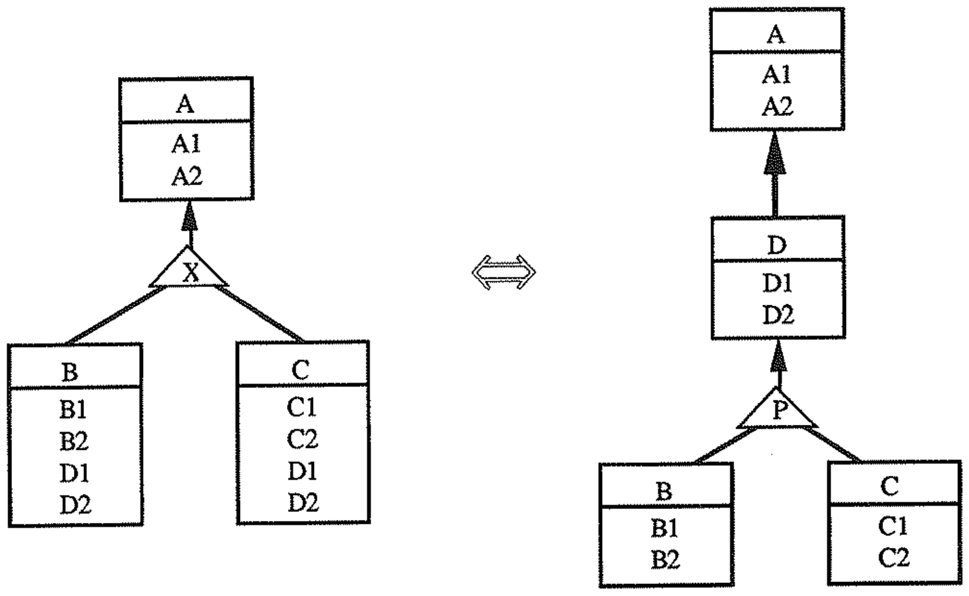
4.8.1 Technique 1 (Factoring two subtypes)

Subtypes B and C of entity type A have common attributes, i.e. attributes that have the same syntactic properties (domains, constraints) and that are asserted[3] to have the same semantics (they are synonyms, they represent the same properties for similar objects). In the following example, the common attributes have been given the same names for simplicity. These attributes are associated to a common supertype D, which in turn is made a subtype of A. If A has other subtypes, they are left unchanged.

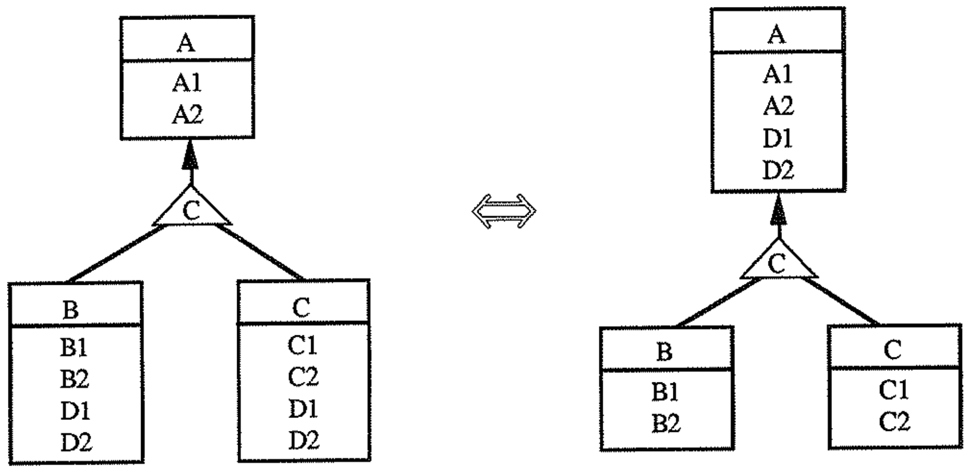


If B and C are disjoined within A, then they are disjoined within D as well, and therefore form a partition :

3 This means that the correspondence between pairs of attributes must be specified externally, e.g. by the designer, and is not an intrinsic property of the schema.



If B and C form a cover (or a partition) of A, then D and A have the same population, and are merged :



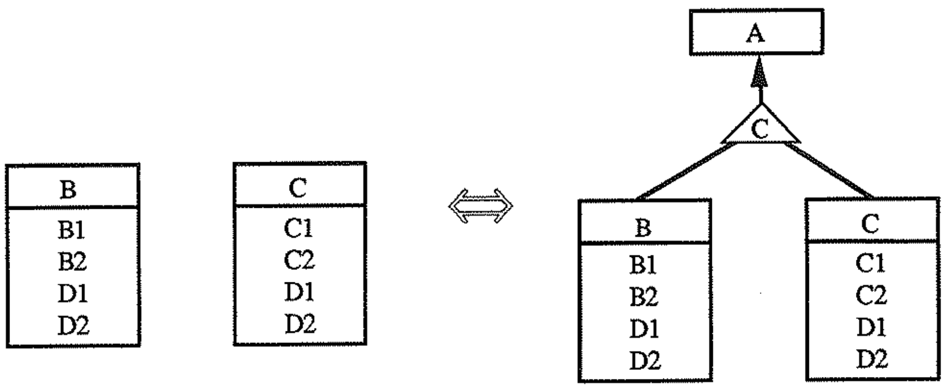
Extension to roles and constraints

Roles that are common to B and C can be associated to D instead. The concept of common role can be materialized by a multi-ET role, or by representation thereof, e.g. by exclusive rel-types.

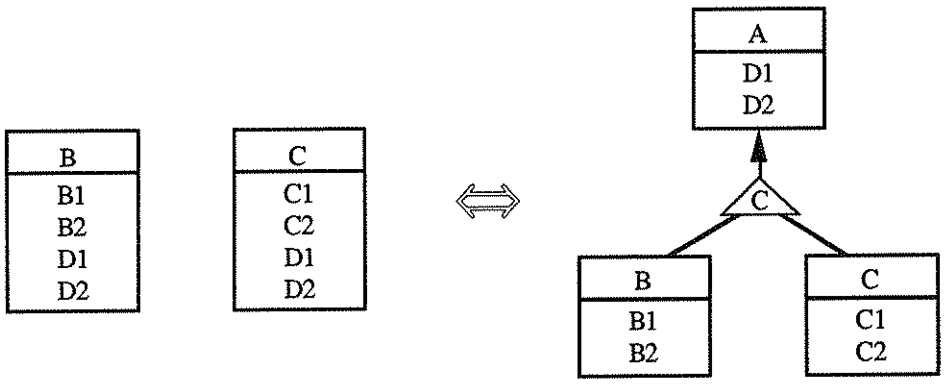
Common constraints can also by associated to A, provided their supporting components are common to B and C as well.

4.8.2 Technique 2 (Factoring two entity types)

This case can be dealt with by transforming it into the previous one. A common, empty supertype is first added in the schema. In an unstable schema, one can admit that the populations of A and B may overlap. If such is the case, B and C form a cover of new entity type A. Otherwise, they will be supposed to be disjoined, and that they form a partition of A.

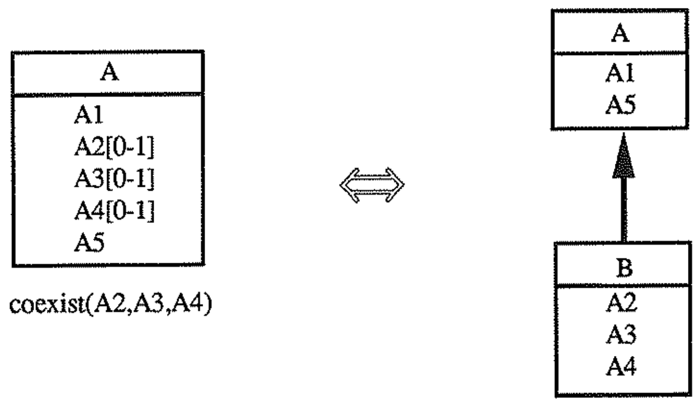


According to technique 5.8.1, the target schema is further processed, yielding the following pattern :



4.9 Defining a subtype for an entity type

We have seen that a coexistence group can be transformed into an optional compound attribute (technique 2.9.4), and that an optional attribute can be transformed into an explicit entity type , linked to the origin entity type through a one-to-one rel-type (technique 2.9.2). In addition, a one-to-one rel-type can be interpreted as the implementation of a generalization relation (technique 3.6.2). By composing these transformation, we obtain the transformation of a coexistence group into a subtype. This technique can be extended to roles as well.


A
A1
A2[0-1]
A3[0-1]
A4[0-1]
A5
coexist(A2,A3,A4)
A
A1
A5
B
A2
A3
A4

Chapter 5

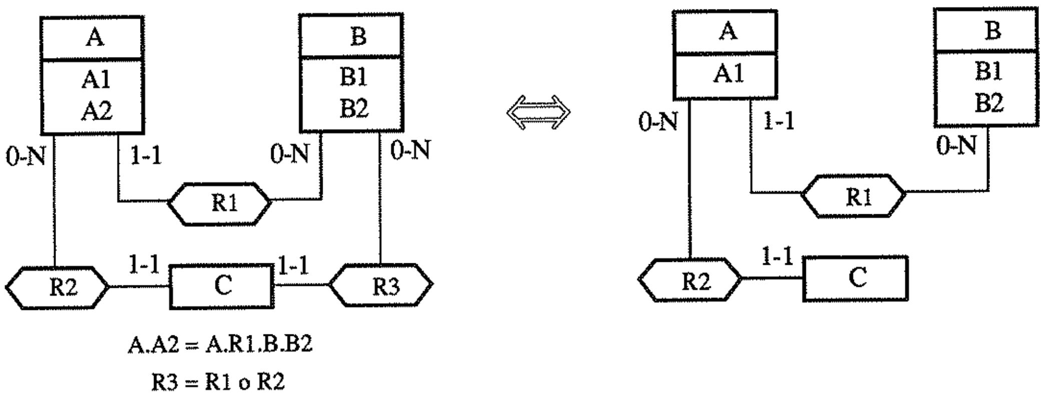
OTHER TRANSFORMATIONS

5.1 Introduction

The following techniques concern physical constructs that appear in DMS schemas. They are useful in *untranslation* and *de-optimization* processes.

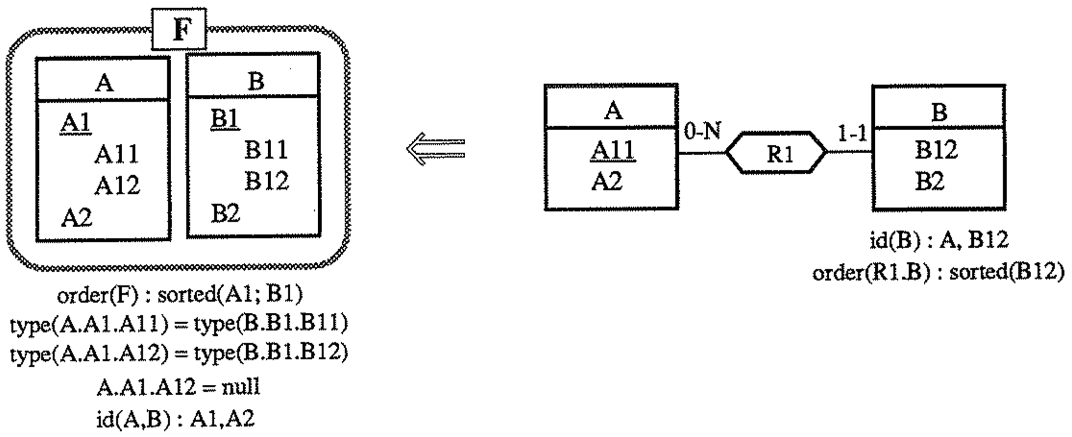
5.2 Structural redundancy reduction

Structural redundancy techniques consist in adding new constructs in a schema such that their instances can be computed from instances of other constructs. Attribute duplication, rel-type composition and agregate representation are some examples of common optimization structural redundancies. These transformations are reversible since they merely add derivable constructs without modifying the source constructs. In the following example, a duplicate attribute and a redundent rel-type are removed.



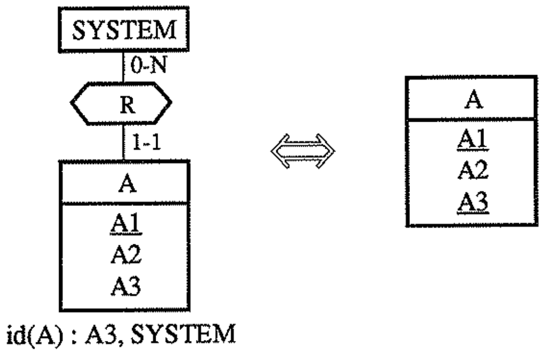
5.3 Multi-record-type files

A one-to-many rel-type can be implemented as a sorted multi-record-type sequential or indexed sequential file. The identifiers/keys are structured in such a way that an A instance is followed by its associated B instances in the file sequence. The transformation is not reversible unless a referential constraint from B.B1.B11 to A.A1.A11 can be proved, for instance by file contents examination. However, this physical pattern is sufficiently frequent to make its rel-type origin strongly probable.


F
A
A1
A11
A12
A2
B
B1
B11
B12
B2
order(F) : sorted(A1; B1)
type(A.A1.A11) = type(B.B1.B11)
type(A.A1.A12) = type(B.B1.B12)
A.A1.A12 = null
id(A,B) : A1,A2
A
A11
A2
0-N
R1
1-1
B
B12
B2
id(B) : A, B12
order(R1.B) : sorted(B12)

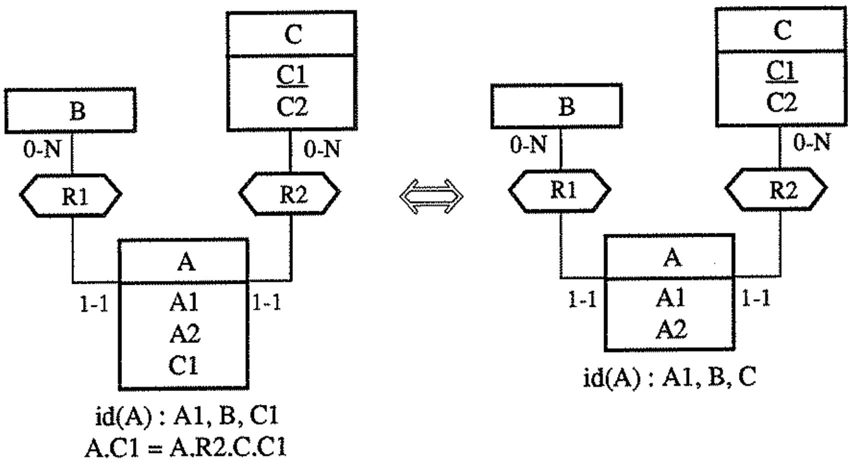
5.4 Identifier in singular rel-type

When an identifier is made of attributes and of one role, and when this role is taken by a single-instance entity type, then this role can be discarded. This practice is typical of CODASYL schemas, where an entity type can have one all-attribute identifier only. The trick is to insert the entity type (i.e. the record type) into a SYSTEM-own set type, and to declare the identifier as local in this set type.



5.5 Exchanging attribute and role components in an identifier

When a component of an identifier is an attribute that is a copy of the identifier of a linked entity type, this attribute can be replaced by this entity type. This pattern can be found in CODASYL schemas, where an identifier (or a access key) cannot include more than one role. The other roles can be replaced by a copy of the identifier of their entity type.



Phenix Project
BIKIT-FUNDP
IRSIA/IWONL Convention nb 5421

PHENIX CARE Tool Specification

The User's View

Version 2.0

Specification report PHENIX-SPEC-9303-01

March 1993

Abstract

This document specifies the user's view of the Phenix CARE tool (the Phenix end-user is the so-called reverse engineer).
Section 1 describes the user model of the CARE tool concepts. Section 2 details the functionalities available for the user. Section 3 illustrates the use of these functionalities in a scenario-oriented way.

About this version

This version presents concepts and functionalities of the Phenix CARE tool. Detailed concepts and functionalities are the ones provided in the system. Some concepts and functionalities are just mentioned: they are considered as interesting in a CARE tool, but could not be developped within Phenix.
Section 3, which illustrates the use of these functionalities in a scenario-oriented way, will be completed in the version of this document.

Table of Contents

Warning

This document is build up to introduce progressively the Phenix CARE tool. Concepts and functionalities are first defined in their basic meaning, and further detailed as long as you go on reading for advanced use of the tool.

I. User model of the CARE tool concepts

1. Project and application

a. Project

A **project** is the basic work unit of the CARE tool. It collects the general information about one application to reverse engineer up to its final result.

A project has:

a name[1],

a creation date, (automatically set by the CARE tool),

an associated directory where a copy of the source text files of the application is available,

associated textual comments (optional),

a responsible reverse engineer (optional).

b. Application

While using the CARE tool, the user wants to recover the (conceptual) description of a (sub)system, called an **application**.

When starting this reverse engineering process, an application has a set of its source text files. Further on, it will also be viewed as a (set a of) data schema.

An application:

is a set of source text files,

is a set of schema,

is described by one project.

2. Source Text Files, Module and Data File

a. Source Text Files

The **source text files** of an application contain its (whole) source code[2]. Most of these source text files contain the description of one or several programming module(s) with, possibly, one or several data file description(s)[3].

A source text file is written in a specific programming language. According to source text inclusion facilities existing in many languages, the text of a source text file is 'pre-processed' to solve precompilation techniques such as include, copy with/without replacement, instanciations, etc. Two texts are thus associated to any source text file of an application: the original source code text (with precompilation statement(s)) and a

[1] The name of a project is preferably the name of the corresponding application to reverse engineer.

[2] An application is basically described by the collection of its source text files. Source text files only are modelized in the CARE tool. Other knowledge sources about the application, such as external documentation reports, are not directly handled by the CARE tool. Nevertheless, the user always keeps the opportunity to introduce schema description by himself.

[3] Any source file text must be in a syntax recognized as valid by the extraction process

text where all precompilation statements are expanded.

Source text files can be discriminated on the following criteria. While parsing the expanded text of a source file, the syntactical analyser can detect a (set of) autonomous programming declaration unit(s) or not (according to the programming language; for instance 'program-id' in Cobol). Each time such a programming unit is found out, a (programming) module is associated to the corresponding portion of text of the source file.

At last, source text files have a specific physical location according to the (running) application[4]. This information is considered as optional, because it is not very easy to introduce. In fact, it could be helpful when sources files must be grouped to launch a common extraction but it has not many other uses.

A source text file has:

a name,
a physical location (optional),
an original text,
an expanded text,
an associated language,
includes and/or is included into 0 to n other source text file(s),
has 0 to n associated modules.

b. Module

A **module** is a (textual) declaration unit. Two kinds of **modules** are managed by the CARE tool: programming modules and declaration modules.

Programming modules are the most usual modules. They correspond to processing units of the application (mainly such as programs). They are automatically created by the parser of the source text files when it syntactically detects such a processing unit.

Declaration modules correspond to a declaration unit defined by the reverse engineer in the expanded text of a source file that does not contain any programming module declaration.

A module:

has a name,
originates from one expanded source text file at a definite position (start and end position),
has 0 to n submodule(s) and/or is the submodule of 0 or 1 module,
calls 0 to n module(s) and/or is called by 0 to n module(s).

c. Data File

Data files declarations are the basic start elements of the CARE tool when reverse engineering an application[5]. They are found in the (expanded) source text file of modules (the highest granularity level programming modules or a declaration module).

[4] Nevertheless, as mentionned in the Project definition, a copy of all the files that the user wants to get analyzed during the application reverse engineering are gathered into one unique directory.

[5] Other start elements such as data object (schema) introduced by the user are considered as a secondary knowledge source by regards to the application to reverse engineer.

In the CARE tool, the actual distinction between a data file as declared in a programming module, called a **logical file**, and its assigned data storage medium, called the **physical file**, is maintained. As the same physical file may be declared (and used) in different programming modules, a physical file can be associated with several (and possibly dissimilar) logical files.

A physical file has:

a name,

a physical location.

A logical file:

has a name,

describes a physical file,

is declared in a module,

is accessed 0 to n times in its declaration module for I, O or I/O (access mode),

its description contains 0 to n main object declarations.

3. Data Object and Schema, Status (abstraction level)

a. Data Object

The concept of **data object** is used to register any kind of data or information structure.

Data object covers four specialized concepts:

- '**main object**': data object registering a 'main' information concept (self-defined, rather independent concept); physical or logical data structures such as a Cobol record type, a Codasyl record type, or a relational table, as well as conceptual entities such as an entity type in a given ERA model or an 'entite' in the Merise model, are instances of a main object;
- '**property**': data object registering properties or characteristics of other data objects (called their 'father' objects), a property takes one or several (groups of) values; a Cobol record field, a Codasyl data-item, a column of a relational table, an attribute in an ERA model or an 'attribut' Merise model are instances of a property;
- '**link**': data object registering links between at least two main objects, each one playing a specific 'part' in the link; Codasyl sets, relationship types in an ERA model or 'relation' in the Merise model are instances of a link;
- '**programming variable**': any kind of data object used as programming variable in the source text of a module.

A data object is said **elementary** or **compound** whether it comprises no or at least one property.

A **property** has a **minimal** and **maximal cardinality** indicating the minimal and maximal number of values that its father object must or may have for this property. A property with a maximal cardinality equal to 1 or greater than 1 is respectively called a **monovalued** or **multivalued** property.

The concept of **part** is introduced as an intermediary concept between a link and each 'linked' main object[6]. For a given link relating main objects, a part is defined for each main object to express the role played by this main object in the link. To add semantics about these roles and to avoid designation ambiguities in the case of recursive links, a part has a name. A part has a **minimal** and **maximal cardinality** indicating the minimal and maximal number of occurrences of its related link an occurrence of its related main object must or may participate in. A link has a definite number of parts, which is its **degree**.

A data object:
- has one name,
- has type and length information,
- is compound of 0 to n property(-ies),
- has associated textual comments (optional).

A main object:
- inherits data object features,
- most often originates from a logical file (see section I.2.c),
- plays 0 to n part(s).

A property:
- inherits data object features,
- has a minimal and maximal cardinality,
- is the component of one data object.

A link:
- inherits data object features,
- has 2 to n related parts and a degree.

A programming variable:
- inherits data object features,
- originates from one module (see section I.6).

A part has:
- a name,
- a minimal and maximal cardinality,
- a multiplicity.

b. Schema

A **schema** is a set of data objects. It either originates from extraction process (automatic or user-directed) launched on the (expanded) text associated to one or several modules -this is the most usual case-, or it is created by the user through an enrichment process.

More than one modules may be grouped to activate the automatic extraction process at once on a group of 'related' modules (and therefore source files). The result of this grouped extraction is a schema that contains the data objects descriptions issued from all

6 not necessarily distinct: recursive links are allowed

the modules gathered in the group.[7]

A schema:

has a name,
contains 0 to n data object(s),
groups 0 to n module(s),
associated textual comments (optional).

c. Status (abstraction level)

A **status** characterizes data object and schema. It registers their abstraction level as usually defined in database forward and/or reverse engineering.

Possible values of status are 'physical', 'logical', 'conceptual' and 'undefined'. These values are:

- automatically set by the CARE tool according to specific functionalities activated on a data object or a schema (for instance, objects created by an extraction process have a 'physical' status),
- simply maintained by most of basic functionalities,
- interactively set by the reverse engineer when (s)he estimates that a data object or a schema has reached a specific abstraction level (see section II).

This concept is defined in order to introduce methodological assistance in the CARE tool. For instance, it can be used to force some reverse engineering methodology based on the sequential reconstruction of the physical, the logical and finally the conceptual schema of the application. Moreover, the physical schema is considered as a specification criterion for the concept of origin.

A status:

equals 'physical', 'logical', 'conceptual' or 'undefined',
qualifies a data object or a schema.

N.B.: The status of a programming variable is always 'physical'.

4. Data Object Subordinate Concepts

4.1 Data Object Type

Data objects with a 'physical' status are always typed (programming languages force explicit typing or infers implicit typing for all data structure definitions). Type information includes:

a. a **domain type** such as numeric, alphabetic, alphanumeric, text, date, pointer, boolean, complex, 'user-defined';

[7] A module may belong to at most one group of modules so that its data objects descriptions are present in only one schema. The only case where data objects descriptions issued from the same source file can be found more than once in the schema(s) of an application comes from source text inclusion facilities provided in the programming language. Nevertheless, these multiple descriptions will immediately be suggested for integration because of their common and unique textual origin.

b. a logical and/or physical length (optional);

c. an editing format[8] (optional).

Moreover, the possible value domain of data objects may be more accurately constrained. For a given data object, type information can be completed by the definition of significant 'values'. Using this secondary concept, the value domain of a data object can be restricted to an interval or enumerated domain. An interval domain is defined via the definition of its lower and upper limit value. An enumerated domain is defined via the definition of all its possible values. Optionally, a significant name may be associated to any specific value or interval linked to a data object.

When and while appropriate, this kind of information is maintained for data objects with another status than 'physical'.

An elementary data object has:
 a domain type,
 a logical and physical length (optional),
 a format (optional).
A compound data object:
 has inferred type information from its component property(-ies) type information.
A domain type value is 'numeric', 'alphabetic', 'alphanumeric', 'text', 'date', 'boolean', 'pointer', 'complex', or any other user-defined value.
A significant value:
 equals a specific value,
 is linked to 1 to n elementary data object(s),
 is used as lower limit of 0 to n interval definition(s),
 is used as upper limit of 0 to n interval definition(s),
 has a name (optional).
An interval:
 is linked to an elementary data object,
 refers 2 values: its lower and upper limit,
 has a name (optional).

4.2 Data Object Key

In many programming languages, various access facilities are provided between data structures. As for data object type, this kind of information is useful to achieve some reverse engineering tasks or reasoning processes. Most of these access facilities correspond to key techniques[9]. On this basis, four kinds of key between data objects are conceptualized in the CARE tool.

[8] Format is specified using the Cobol-like format conventions such as XXBXX, A(4), S99V99,...

[9] Most of these key concepts basically relate to data objects with a 'physical' status. Nevertheless, as long as it migth be needed, key information will be accessible for data objects with a status value other than 'physical'.

Note: Link data object implies both side access 'facilities' between linked main objects. This is an implicit property of link data object. It is not explicitly registered in the CARE tool.

a. Access Key

An access key is a mechanism providing fast access to a data object given a set of values for one or several of its component property. Therefore, an access key is a set of one or more property(-ies), and is defined as primary or secondary (only one primary key is allowed per data object).

An access key can be an identifier, if all its components also compose an identifier.

An access key:

relates to one data object[10],

is constituted by 1 to n property data objects, components of this data object,

is primary or secondary.

b. Sort key

A **sort key** is a mechanism allowing (fast) sort facilities of data object occurences according to a set of values for one or several of its component property, each one considered with a specific sorting mode ('ascending' or 'descending'). Therefore, a sort key is a set of one or several oredered property(-ies). A sort key does not always provide a one-to-one access link.

A sort key:

relates to one data object,

is made up of 1 to n ordered property data objects, components of this data object.

c. Relative key

A programming variable may be used as a **relative** key for a data object[10].

A relative key:

relates to one data object,

is made up of one programming variable.

d. Pointing relationship

A data object[11] may **point** to a main object.

A pointing relationship relates:

a 'pointing' data object (0-n),

a 'pointed' main object (0-n).

4.3 Identifier

The **identifier** of a data object[12] (the identified one) is a set of one or several data objects (the identifier components) such that there may not exist more than one identified data object occurence with the same set of values for the data objects components of itsidentifier. Let's define temporarily a component of a data object

[10] This data object must be a main object if its status equals 'physical' but might be another specialization of data object otherwise.

[11] This data object may not be a link object if its status equals 'physical'

[12] except programming variable

identifier as either a property of this data object or a (property of a) main object 'linked' to this data object[13]. A more accurate definition using the concept of path is given in section I.8.4.

Note: An identifier must not be defined if the corresponding identifying property can be inferred from another concept such as the 1-1 cardinality of a part implied in the identifier.

Note: An identifier can be an access key, if all its components also compose an access key.

4.4 Constraint[14]

Constraints are additional features that must be satisfied by the data objects, parts, or values they concern. A major distinction sets apart strong-typed constraints and weak-typed constraints. **Weak-typed** constraints have no bound semantics (free text), whereas **strong-typed** constraints have a predefined semantics (so are referential constraint, functional dependency, existence constraint and inclusion, equality or exclusion of value, part or link). Specific inference and management processes are provided for strong-typed constraints as far as their semantics is 'understood' by the CARE tool.
Strong-typed constraints are briefly recalled here below[15].

4.4.1 Inclusion and Referential Constraint

An inclusion constraint states that the values of given attributes must be a subset of the values taken by other attributes.

A referential constraint defines an inclusion constraint between the value(s) taken by a (set of) referencing property object(s) and value(s) taken by a (set of) referenced property object(s) moreover declared as an identifier for its (their) father object.

4.4.2 Functional Dependency

a. Functional Dependency between Properties

Given a main object or a link, a set of one or several property objects (the determined one) is functionally dependent on another set of one or several property objects (the determining one) if, at any time, each value of the determining set of property(-ies) is associated with only one value of the determined set of property(-ies).

b. Functional Dependency between Parts

Given a link, a set of one or several of its parts (the determined ones) is functionally dependent on another set of one or several parts (the determining ones) if, at any time, each value of the determining set of part(s) is associated with only one value of the determined set of part(s).

[13] See Methodological guide, Vol I, chapter 2, section 2.3, for a detailed presentation of the concept of identifier

[14] Identifier, part and property cardinalities are constraints that have received an independent modelization

[15] See Methodological guide, Vol I, chapter 2, section 2.3.11, for a detailed presentation of strong-typed constraints

c. Multivalued Functional Dependency between Properties

A multivalued functional dependency corresponds to a functional dependency between a determining (set of) property(-ies) and a multivalued property.

4.4.3 Existence Constraint

An existence constraint makes the existence of a data object dependent to the existence or value constrainment of other data object(s)

4.4.4 Inclusion, Equality or Exclusion of Parts

These constraints refer to parts played by the same main object.
A parts inclusion constraint implies for a main object that plays a part (the including one) in a link, to play another part (the included one) in another link.
A parts equality constraint implies for a main object that plays a part in a link, to play another part in another link, and conversely.
A parts exclusion constraint expresses that several parts played by a main object are mutually exclusive.

4.4.5 Inclusion, Equality or Exclusion of Links

These constraints are similar to the inclusion, equality or exclusion constraints of parts but at the level of link: they concern all the parts of the link.

4.4.6 Inclusion, Equality or Exclusion of Values

These constraints are aimed to reduce the set of values that one (or several) property object(s) may take. Such restrictions make reference to constraining elements such as:
- the domain type itself,
- one or several property(ies) of the same data object,
- one or several property(ies) of another data object.

4.4.7 Weak-typed Constraints

A weak-typed constraint is a constraint with no predefined semantics: it is just free text. However, this text may explicitely refers two kinds of 'significant' elements: variable and path. In the text of a weak-typed constraint, a variable can be used to designate one given instance of a specific data object (main object, property, link or programming variable) while a path details one given connection between two data objects (see section I.8.4). Variables and paths are significant elements in the sense that the text in which they appear is automatically updated in accordance to any data objects structural modification.

5. Ancillary concepts

5.1 Name and Thesaurus

Names are used in various reverse engineering reasoning processes to infer semantic information. In fact, names appear as a concept to be handled in an autonomous way:

they can be made up of several parts, each one being significant by its own, the use of prefix, suffix and abbreviation is a common practice, synonyms and homonyms occur, etc. Moreover, names belong to a specific language (English, Dutch, French, etc), but have traduction(s) in some other languages.

Name conceptualization and management is synthesized in the Phenix CARE tool in the following way:

- names are considered as strings of character(s), made of substrings,
- substrings are detected upon the use of delimitors (such as '-' or '_'), changing between upper and lower cases or changing between numeric and alphabetic characters,
- standard substrings are predefined to indicate identifying, referencing or pointing features,
- serial names are detected upon name correspondence except a numeric substring,
- synonysms are recorded as pairs of two synonysm names,
- an abbreviation is viewed as a synonysm,
- prefix/suffix handling is provided only via string find (and replace) functionalities,
- names are handled independently of their so-called object(s) type,
- no language qualification is provided.

A **thesaurus** registers a set of synonysms (pairs of synonysm names) significant for the application to reverse engineer.

A name:

has a string value,

is a synonym of and/or has as a synonym 0 to n other names.

A thesaurus:

has a name (optional).

5.2 Connected Data Objects

Two **data objects** are **connected** if there exists a succession of other data objects composing a 'connection', between the two data objects said to be connected, via part(s) and/or property/'father' data object composition relationship(s). If a connection is entirely defined via only one part or one property/'father' data object composition relationship, the two connected data objects are said directly connected, otherwise they are said indirectly connected.

A connection will further be detailed through the definition of the concept of 'path' in section I.8.3.

5.3 Transfer Instruction

A **transfer instruction** between data objects is a concept pointed out in several extraction reasoning processes. As so, transfer instructions between data objects with a 'physical' status are recorded (and accessed while appropriate). Various programming

statements may be registered as transfer instructions: for instance in Cobol, so are move (corresponding), read/write, compute, etc.
Inference rules combine transfer instructions to find out more complex transfer links between data objects.
A transfer instruction relates:
   a 'transfered into' data object (0-n),
   a 'transfered from' data object (0-n),
   originates from one module.

6. Origin

6.1 Needs for the concept of 'origin'

Let's consider two main profiles of reverse engineer: a database administrator and a programming manager.

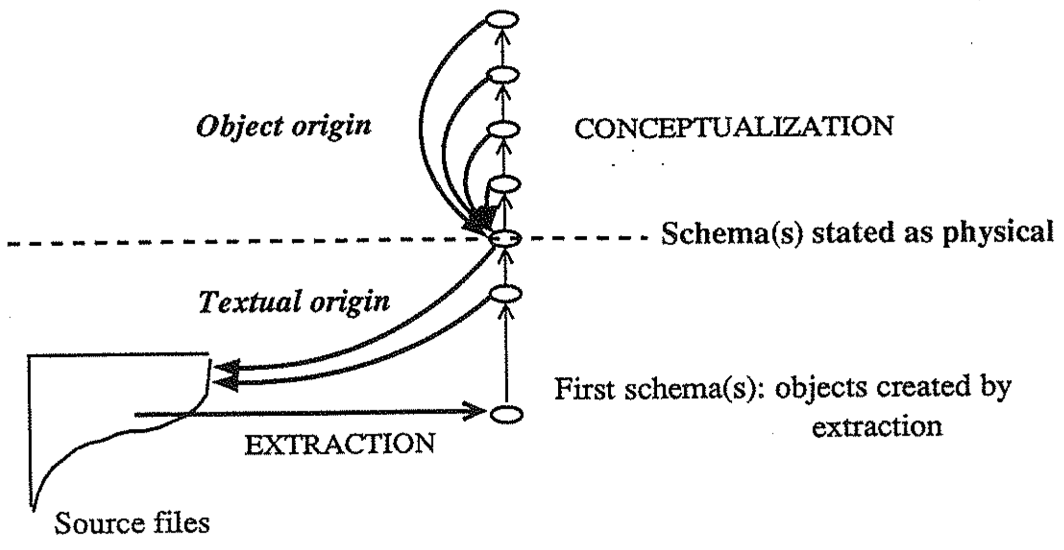
a. The end-user as a database administrator
   A database administrator manages the information system(s) of a company: he controls its conceptual model(s) and the corresponding translated physical model(s). He must be able to indicate whether any given information is present in the information system, and if so to locate this information in the implemented system in order to define an access request to it. To do so, starting with any element of the information system (from the conceptual model), he needs to know how it is present in the physical model. This link between a conceptual information and its translation in a physical schema corresponds to an origin in terms of objects.

b. The end-user as a programming manager
   A programming manager is responsible for the maintenance of programs developped in a company. Starting with any element of a physical schema, he needs to know where this element is referenced in programs to meet any software evolution requirements. The link between an object in a physical schema and the, possibly many, declaration(s) and use(s) in source files corresponds to an origin in terms of textual references.

6.2 Dual Definition: Textual Origin vs Object Origin

Because of these various needs, a dual definition of the concept of origin is proposed, based on the explicitation of the physical schema[16] of the application.

[16]as perceived by the user: the physical schema is a close image of the text of the source files of the application - it contains all the information declared in the source files - but it may have already been restructured (mostly via transformation and integration processes, including gen/spec structures, etc)


Object origin
CONCEPTUALIZATION
Schema(s) stated as physical
Textual origin
First schema(s): objects created by extraction
EXTRACTION
Source files

- **'Textual origin'**: position (start and end line number - possibly the same) in a expanded source text file where an extracted object is referenced (declaration statement).
- **'Object origin'**: data object from the physical schema from which a data object from a further schema derives[17].

Note: While the user does not define a schema as a physical one, the CARE tool considers that the physical schema is the first extracted schema.

A textual origin relates:

a data object,

to one module at a given position (start and end line number).

A data object has 0 to n textual origin(s).

An object origin relates:

a data object,

to a (set of) data object(s) from the 'physical' schema.

Note: The composition of textual and object origins provides the textual origin of a data object from a schema at an upper level than the physical one

7. Data Object Correspondence

Data object correspondences make explicit correspondences existing between two (sets of) data objects. They are characterized by features such as:

- semantically, correspondences are registered as 'equality' corrspondences[18];

[17]See Methodological Guide, Chapter 2, p 2-29. An object can have more than one origin. Any reverse engineered object has at least one origin.

[18]Two other types of correspondences should be interesting to register: they are 'inclusion' and 'subtype' correspondences. They are not provided as such in the Phenix CARE tool but data objects transformations, integration and creation processes handling generalization/specialization structures can be considered as assimilating them.

- a correspondence is based on various elements of 'comparison': name, structure, physical type information (domain, length, format, position), transfer instruction, textual origin, physical and/or logical file belonging and context (relationships with other data objects);
- a correspondence is defined either intra- or inter- schema.

Correspondences are intensely used throughout any integration process; they are also used to suggest some specific process execution.

How are they found out? First eventuality: the user spontaneously declares them. If not, the detection of data objects correspondences is a functionality, provided by the CARE tool, that is based on data objects analysis on the various comparison criteria described hereabove. Following this analysis, correspondences are either asserted as certain, or the user is asked to confirm one (or several) of the correspondence proposals.

A correspondence:
relates a set of 1 to n data objects,
to a set of 1 to n data objects,
is 'scored',
is based on specific comparison elements.

8. Advanced Definitions

8.1 Generalized and Specialized Main Objects

Generalization/specialization is allowed in the CARE tool. A generic main object has a set of at least two specific main objects, a specific main object is specific to at the most one generic main object. As far as the population of gen/spec main objects is concerned, a partition constraint is required upon the set of all the specific main objects of a generic one. Downward inheritance mechanism is automatically defined for the property and link objects of a generic main object.

A main object:
is generic of 0 or 2 to n main objects (called 'subtypes'),
is specific of 0 or 1 main object (called 'supertype').

8.2 Multiple Part

A part relates a link to the main object that plays a role in this link. If the same part may be played by several main objects, the part is called **multiple part**. The **multiplicity** of a part indicates how many different main object(s) may play this part.

8.3 Path and Arc

A **path**[19] is a way to designate a 'connection' between two data objects. The intermediary concept of arc is used to lead to the definition of a path. An arc is either a

[19]See Methodological guide, Vol I, chapter 10, for a detailed presentation of this approach

part or the component relationship between a property and its father data object. Therefore, a path is a list of arcs and intermediary data objects composing a connection between two data objects.
Arcs and paths have cardinalities which express, in both direction, the minimum and maximum number of data objects that must and may be linked by the arc or path.
A path is designated by a name automatically built up by the CARE tool on the basis of the path components.

Paths are essential for the definition of identifiers; they can also be used in the expression of weak-typed constraints. Their most important feature is that they are updated in conformity with any data objects transformation.
A path:
  is a list of arcs and data objects,
  has a minimal and maximal cardinality,
  has a system-defined name.

8.4 Complex Identifier

A **complex identifier** is an identifier made up of more than one identifying component that must not be directly connected to the identified object. To allow the definition of such identifiers, **identifier components** are considered as pairs defined as follows:
  (identifying data object,
  path relating this identifying data object to the identified one).
An identifier:
  has 1 to n components,
  identifies one data object (called the 'identified' object).
An identifier component:
  participates to 1 to n identifiers,
  is a pair of:
    - one identifying data object (main object or property),
    - a path relating this identifying data object to the data object identified by its identifier.

8.5 Position of a Property in its father Data Object, Concept of 'Zone'

Elementary data objects have type information including domain and length declaration.

8.6 Multiple Description

A data object is compound of 0 to n property(-ies) with overlapping possibilities. Indeed, more than one composition relationships may exist for the same data object when several descriptions are gathered for the same data object[20] (extracted from the code or introduced by the user). This feature is materialized in the CARE tool by the

[20]See Methodological guide, Vol I, 'The Representation Problems in Databases', chapter 8

addition of a start and end position of a property in its direct 'father' data object (property or main), according to its own type information and the one of any 'brother' property(ies).

A property data object has:
a start and end position.

8.7 Name Unicity Constraints and Name Enforcing

Name unicity constraints are imposed upon the instances of the concepts handled by the CARE tool. They are presented herebelow. May not have the same name the following objects:

- all main objects of a given schema;
- all programming variables of the given schema
- all the properties (possibly inherited) of a given main object;
- all the links of a given main object and of its subtype main objects;
- all the 'named' parts of a given link (the name of a part is optional)[21];
- all schemas of an application.

Name enforcing is automatically made by the CARE tool when it must create objects that viloate name unicity constraints. Such violating names are suffixed by '*' followed by a number, the whole suffix turning the name into a name meeting the name unicity constraints.

9. State, state tree and version

9.1 State and state tree

All the states management facilities are based on the following two basic concepts: states and state tree. A **state tree** registering remarkable states is maintained during any work session started by the user (see section II.1). A **remarkable state** is created each time the user launches a reverse engineering process. Basically, they are registered sequentially in the state tree. But the creation of a remarkable state in parallel with another is afforded to allow the evaluation of several reverse engineering hypotheses.

A **multihypothesis** corresponds to the opportunity for the user to evaluate several reverse engineering sequence of processes. The multihypothesis state is a remarkable considered as the entering point of a hypothesis.

In the CARE tool, some implementation considerations implies the following features[22]:

[21] A part may have no name if no one of its main objects (and any of its generic or specific main objects) participates to the same link.

[22] therefore they are not justified by actual conceptual requirements

- the state tree is developed globally to an application: it is not local to a schema;
- the state tree can not be maintained further any schema (un)loading: it is reinitialized after any of these actions (states navigation can not bypass the execution of such actions);
- the state tree is maintained during a work session but is not registered outside the session (versioning facilities are provided to keep a permanent trace of remarkable schema states, including hypothesis evaluation).

9.2 Version

A schema can be saved under several versions (a schema has a version number). A version tree is maintained for each schema (independently of a work session). Thus, access to any version is allowed (though only one version is loaded at a time); modification of any of these version can be done in combination with the various savings facilities (save as the same version, save as a new version, save as another schema).

Among all versions of a schema, there can be one with a specific status - the image of the 'physical' level of abstraction of a schema; it determines access to object origin(s). This version of a schema is automatically saved by the system when the user declares it as corresponding to the physical schema; it will be maintained as a read only version (no 'save as the same version' allowed).

10. Method

A **method** refers to a reverse engineering methodology (to be) followed by the reverse engineer. As so, it must be regarded as a global process indicating what can be done and when. Considering the whole reverse engineering process, specific activities have been identified and ordered, so that the various problems encountered during the overall reverse engineering process are solved[23].

[23]See Methodological guide, Part IV, 'Reverse Engineering Methodology'

II. Available functionalities in the CARE tool

Preliminary Note.

Most functionalities are 'context sensitive': some of their application parameters are objects currently selected in the user interface. In the following sections, when the word 'current' will be used to designate an object, it is to be understood as the object 'currently selected in the user interface'.

1. Starting and Managing a Work Session

A work session is started each time the user activates the CARE tool. Within a session, the CARE tool runs by default as a toolbox: the reverse engineer is free to launch any functionality of the system at any moment. This is considered as a methodology-free running mode (nevertheless, the absence of methodology is itself a method).
On the contrary, the reverse engineer migth be constrained within a specific methodology upon selection of a specific reverse engineering methodology[24], but methods are not externalized in the Phenix CARE tool.

Some reverse engineering functionalities can be customized by the user. Customizable elements are: comparison criteria (with weight valuation) for correspondence detection and integration, naming conventions (specific names or delimiters), application description and schema graph contents.
The extraction process of a schema is not customizable[25] from the global session customization.
Standard and default customization values are defined in appendix; note that they are totally application independent within the tool[26].

a. Session customization

- **Customize...**
  Displays the several customization elements of a session with their current values and allows any update
- **Load Options**
  Restores the last saved values of the customization elements
- **Save Options**
  Saves the current values of the customization elements

[24] See Methodological Guide, volume I, Part II

[25] The initial extraction of a schema could be customized in terms of the concepts to be extracted. Nevertheless, such a customization would imply that the object base was completed by the registration of the 'not extracted yet' information. In fact, if instances of a concept are not present in the OB, it no more automatically means that this concept is not instanciated in the source files, for it can simply mean that it has not been extracted yet.

[26] There is no explicit registration of customization values according to a specific application.

b. Method definition

- **Define Method...**

c. Session Quitting

- **Quit**
  Closes the current project and exits the current work session
  If necessary, asks for project save confirmation before closing it

2. Project Handling

a. New pro-stack ject declaration

- New...
  Asks the user for:
  the new project (application) name
  the directory where (a copy of) the source text files composing the application to solve in this project are gathered
  Optional information is prompted too:
  the responsible reverse engineer
  textual comments
  Initializes the work session for this project

The work session initialization of a new project implies the following actions:
- declaration of all the source text files contained in the directory associated to the application,
- initial extraction of these source files to get all the information needed to show the application description and to organize it into schema

b. Existing project opening

- **Open...**
  If a project is currently opened, asks the user to confirm to close it
  Displays the list of all the projects known by the Phenix CARE tool to allow one project selection
  Initializes the work session for the project selected by the user

The work session initialization of an existing project implies to load all the initial information associated to a project: mainly application description and source organization

c. Import/Export

- **Import...**
  If a project is currently opened, asks the user to confirm to close it
  Asks the user for the name of a file where the contents of a project is stored (including the whole application description and contents)
  Initializes the work session for this project

The work session initialization of an imported project implies to load the project and its application description and contents as stored in the corresponding file

d. Project information

- **Info...**
  Displays the list all the projects known by the Phenix CARE tool
  For any selected project, displays the overall associated information (creation date, source text files directory, responsible reverse engineer and textual comments)
  For displayed project information, allows update of the responsible reverse engineer and textual comments

e. Project and its application description

- **Graph**
  Displays a graphical description of the application associated to the current project
  According to the current customization of the system, this description contains:
  source files,
  the modules (programming ones and user-defined declaration ones if any),
  the physical data files,
  the logical data files and their main (data) objects,
  call relationships between modules,
  include relationships between source files

f. Project current state saving

- **Save**
  Saves the current state of the current project, as it is at this moment of the work session
- **Save as...**
  Saves the current state of the current project under a new project name prompted to the user
  The new project creation date is set to the current date
- **Export...**
  Asks the user for the name of a file where the contents of the current project must be stored
  Generates this file according to the current state of the project

g. Project deletion

- **Delete**
  Asks for project deletion confirmation before deleting it

h. Project quitting

- **Close**
  Closes the current project in the current work session
  If necessary, asks for project save confirmation before closing it

3. Data Object Extraction Phase

3.1 Source Organization

a. Source organization and schema declaration

- **Organize...**
  Displays the list of all the files existing in the source text files directory associated to the current project (as already grouped if so) with the modules associated to them
  Allows a multiple selection of modules to define additional groups of files for which the user wants a common extraction process
  Asks for a name for any set up group and creates an empty schema, called with this name, in which all the concepts to be created by an extraction process launched on this group of files[27] will be located
  Optionally, textual comments are associated to any set up group or, in fact, to the schema corresponding to this group of modules
  Note: Already extracted modules of a schema are asterisked; such modules can not be removed from their associated schema

b. Module declaration

- **Associate Module To Source File...**
  Allows selection of lines in the current edited source file
  Creates a module that is associated to this source file with this portion of text
  Launches the extraction of concepts presented in the application description

3.2 Source consultation

a. Text browsing
The text browser provides usual text file browsing facilities: navigation and elementary find facilities

- **Show File...**
  Displays the list of all the source files of the current project and allows a single selection
  Allows the user to launch the browser on the expanded text of this source file
- **Show Module...**
  Displays the list of all the modules defined for the current project and allows a single selection
  Allows the user to launch the browser on the expanded text of this module

b. Pattern searching

- **Search**
  Displays the list of all the patterns that the system can search for and allows a multiple pattern selection

a. Pattern definition

- **Define...**

[27]By default, this name is the name of the first source text file selected to start a group definition, but it may be modified by the user

Allows the definition of a new search pattern

3.3 Extraction

a. Automatic concepts extraction[28]

- **Extract...** ('Schema' window)
  Displays the list of all the modules associated to the current schema (already extracted modules are asterisked)
  Launches the extraction process for all the not yet extracted modules and traces the extraction process in a standard file

b. Assisted concepts extraction

- **Search and Extract...** ('Schema' window)
  Displays the list of all the modules associated to the current schema (already extracted modules are asterisked) and allows a single selection
  Displays the expanded text corresponding to this module
  Allows concept(s) creation on the basis of pattern searching
  (Concepts are created in the schema of the module; this schema is possibly still empty)

4. Data Object Conceptualization Phase

4.1 Schema management

a. Schema display

- **Open Schema...** (menu 'Project')
  Displays the list of all the schema known for the current project and allows one schema selection
  Loads the schema and opens a window to display the contents of this schema[29]
  Displays a warning message if there is (are) module(s) associated to the schema that has (have) not been extracted yet

b. Schema reporting

- **Print...** (menu 'Schema')
  Prints the contents of the current schema (see section II.9 'Reporting')

c. Schema comments

- **Info...** (menu 'Schema')
  Displays the list all the schemas of the current project and allows a single schema selection
  Displays the textual comments associated to this schema and allows any update of it

[28] If this extraction was customized, the user would select the concepts to be extracted
[29] Main and property objects, gen/spec structures and links, links and parts are presented at once. Other concepts such as identifiers and constraints are presented at explicit user's request

- **Group**
  Displays all the property(-ies) of the current main data object (direct or indirect) and allows a multiple
  Groups the selected properties (they should have the same 'father' property) into a property the name of which is asked for (optional)

- **Ungroup**
  Displays all the property(-ies) of the current main data object (direct or indirect) and allows a single selection
  Ungroups the selected property

e. several property's ↔ one compound property + several component property's (aggregation / desaggregation of property(s))

- **Aggregate**
  Displays all the property(-ies) of the current main data object (direct or indirect) and allows a multiple selection
  Aggregates the selected properties into a property the name of which is asked for (optional)

- **Desaggregate**
  Displays all the property(-ies) of the current main data object (direct or indirect) and allows a single selection
  Desaggregates the selected (compound) property

f. one main object ↔ two main objects (merging / splitting of main object(s))

- **Merge Mains**
  Upon selection of two (nor specific, nor generic) main data objects, merges them

- **Split Main**
  Displays all the properties of the current main data object (direct or indirect) and allows a multiple selection
  Displays all the links of the current main data object and allows a multiple selection
  Splits the main according to the selected property(-ies) and/or link(s) by the creation of a new main data object the name of which is asked for (optional), related to the splitted main data object by a new link the name of which is also asked for (optional)

g. multiple role ↔ several links

- **Split Part**
  Displays all the parts of the current link object and allows a single selection
  Splits the selected multi-domain part

- **Merge Links**
  Upon selection of two links, displays the two corresponding sets of parts and allows a single selection in each set
  Merges the selected parts

h. gen/spec of main object(s) ↔ the supertype main object only

- **Generic only**
- **Specialize**
  Displays all the properties and all the parts of the current (not already generic) main data object
  Allows the designation of a 'type' property among these properties
  At least two specific main data objects must be described by coresponding multiple selections of properties and parts to be specialized
  Specializes according to these selections by the creation of new specific main data objects the name of which are asked for (optional)

i. gen/spec of main object(s) ↔ the subtype main objects only

- **Specifics only**
- **Generalize**
  Upon selection of two or more (not already specific) main data objects, displays the corresponding sets of all their direct properties
  For each main object, allows a multiple selection of its property(-ies) to be generalized
  Generalizes according to the selected property(-ies) by the creation of a new generic main data object the name of which is asked for (optional)

4.3 Semantic enrichment and refinement

a. Creation of a new schema

- New...   (menu 'Schema')
  Asks the user for the name of the new schema
  Prompts for optional textual comments to associate to it
  Updates the list of the schema known for the current project

b. Addition of new concepts in the current schema

- New Main...
  Asks the user for the name of the new main object
  Prompts for optional textual comments to associate to it
- **Add Property...**
  Asks the user for the name of the new property to add as the last property of the current data object
  Displays default cardinalities and allows update of it
- **Insert Property...**
  Asks the user for the name of the new property to insert behind the current property object
  Displays default cardinalities and allows update of it
- New Link...
  Asks the user for the name of the new link
  Asks for the definition of each part of this link:

allows (multiple) selection of the main object(s) playing this part,
prompts for a name (optional)
displays default cardinalities and allows update of them

- **New Gen/Spec...**
  Asks the user for the name of the new generic main object
  Asks the user for the name of the new specific main objects
  Prompts for optional textual comments to associate to them
- **New Correspondence...**
  Asks the user to select (a set of) data object(s)
  Asks the user to select the (set of) corresponding data object(s)
  Prompts for the correspondence type
- **New Value/Interval...**
  Allows the creation of a significant value or interval of values
- **New Identifier...**
  Asks the user for the definition of the identifier:
  allows (multiple) selection of main or property object(s), component of the identifier.
  For each component leading to a problem with its path to the identified object:
  asks for path definition or component redefinition.
- **New Constraint...**
  Displays a panel

c. Deletion of existing concepts

- **Delete**
  Asks the user for a deletion confirmation of the current schema, data object, part, data objects correspondence, value, interval or domain, constraint or identifier

d. Update of concepts

- **Rename...**
  Asks for a new name for the current object (schema, main object, property, link or part)
- **Add Part...**
  Allows the creation of a new part for the current link
  Allows multiple selection of the main object(s) playing this part
  Displays default cardinalities and allows update of them
- **Modify Part Main(s)...**
  Displays the list of the main object(s) playing the current part
  Allows update of it, that means addition of main object(s) from the list of all the main object(s) of the schema or removal of main object(s) already playing the part
- **Add Gen/Spec Link...**
  Asks the user to select the generic main object (single selection)
  Asks the user to select the specific main objects (multiple selection)

Asks the user to designate generic/specific properties

- **Modify Card...**
  Displays the minimal and maximal cardinalities of the current part or property object
  Allows update of them
- **Refine Property...**
  Allows the user to define component(s) for the current (elementary) data object.
  Asks for each component to create:
    its name,
    a physical length and/or a start position.
- **Unrefine Property**
  Eliminates all the elementary properties of the current compound property object (that is turned into a elementary one)
- **Change Domain...**
  Allows changing of domain type for the current elementary property
  Possible changes are:
    alphanumerical domain → boolean domain,
    alphanumerical domain → date domain
- **Modify Value/Interval...**
  Displays the list of all significant values and intervals already registered
- **Modify Comments...**
  Displays the comments (if any) registered for the current data object and allows update of them
- **Change Status...**
  Asks for a new status value for the current data object or schema
  Displays the list of data object(s) for which this change can be a problem

4.4 Integration

The integration process is considered as a sequence of the two processes: first correspondence detection and then data object fusion.

a. Correspondence detection
   See functionality "Find Correspondence..." (section II.5.2)

b. Merge of two data objects

   - **Merge...**
     On the basis of the currently registered correspondence between two data objects, achieves integration of the two data objects

c. Integration of two data objects

   - **Integrate...**

d. Integration of two schemata

- **Integrate...**

4.5 Conformity management < will not be implemented in the tool >

a. conformity diagnosis

b. conformity enforcing

5. Miscellaneous Functions

5.1 Name management[31]

a. Thesaurus

A thesaurus of synonysms (pairs of synonysm names) is maintained in the tool-defined text file 'thesaurus.txt'. This file can be updated externally with any textual editor.

- **Show Thesaurus**
  Displays all the synonysm pairs stored in the text file thesaurus 'thesaurus.txt'
- **Load Thesaurus**
  Loads the synonysm definitions stored in the text file thesaurus 'thesaurus.txt'
- **Save Thesaurus**
  Saves the current synonysm definitions in the text file thesaurus 'thesaurus.txt'

b. Search facilities

- **Find...**
  Displays predefined search scopes[32] (the current schema is the default selection) and allows a single selection
  Within the selected scope, finds a name or part of a name (the name of the current object is the default 'find suggestion') and displays the result of the search
- **Find and replace...**
  Displays predefined search scopes[33] (the current schema is the default selection) and allows a single selection
  Within the selected scope, finds a name or part of a name (the name of the current object is the default 'find suggestion')
  Prompts for the replacing string[34]

These search functionalities can be used to find (and replace) a prefix or suffix within name(s).

[31] See Methodological Guide, volume I, chapter 6, for a detailed presentation of name management

[32] Predefined search scopes: the current main object, the current schema, a set of schema of the current project, a set of main objects of the current schema,...

[33] Predefined search scopes: the current main object, the current schema, a set of schema of the current project, a set of main objects of the current schema,...

[34] Deletion of a part of name is achieved by using a replacing string that equals an empty string. Addition of a part of name can also be achieved through this 'find and replace' functionality.

c. Similitude management

- **Compare**
  Compares two (sets of) names according to the tool comparison techniques
  Displays the comparison result in a descriptive way and/or with a percentage value

5.2 Data object correspondence management

- **Find Correspondence...**
  Asks the user to select a (set of) data object(s)
  Asks the user to select the schema to be the search space:
    displays the list of schemas of the current project
    allows a multiple selection (the current schema is selected by default)
  Asks the user to confirm customized 'comparison' criteria to use (incl. weight valuation) or to select new ones:
    displays the possible criteria and heuristics
    allows a single or multiple selection (optional selection)
  Displays the results:
    any definite correspondence and/or any correspondence proposal to be confirmed by the user

5.3 Origin management

a. Schema evaluation

- **Set Physical Status**
  Displays the list of schemas of the current project and allows a multiple selection
  Sets the status of the selected schema(s) to 'physical'

b. Access to object origin(s)

- **Object Origin(s)**
  Displays the object origin(s) of the current object
  Note: These object origin(s) belongs to the physical schema version

c. Access to textual origin(s)

- **Textual Origin(s)**
  Displays the textual origin(s) of the current object: source file and position (start and end line number)
  Allows access to the corresponding text

5.4 State Management

Navigation in the state tree is feasible only between remarkable states. Except for 'undo', state management facilities are accessed after display of the state tree.

a. 'Undo' facilities

- **Undo**

Undo the last reverse engineering process launched by the user[35]

Note: An 'undo' may only be made in a 'leaf' state of the state tree

b. go to state x / go back (x times)

- **State Move...**

  Asks the user to specify the state move by choosing between;

  go back (n) times,

  go to (state number)

  Achieves the state move

c. start multihypothesis

To start a multihypothesis, just go in the state considered as the entering point of a hypothesis. As soon as you launch any reverse engineering process, a new hypothesis is created. In a new hypothesis, any action is allowed (except schema (un)loading that should reinitialize the state treeand therefore implied the loss of any hypothesis but the current one).

d. end multihypothesis

- **Keep hypothesis...**

  Prompts for a state number indicating one hypothesis to retain in a multihypothesis

e. reinitialize state tree

- **Init**

  Asks for user confirmation and if so reinitializes the state tree

6. Suggestion

Even in a methodology-free running mode, the user can launch specific functionalities providing different kinds of suggestion. The system makes such suggestion(s) according to the current work context and possibly to the current reverse engineering state of the project.

6.1 Source Organization

Suggest:

- to group modules

6.2 Extraction

Suggest:

- to get better description
- to get referential constraint through key and transfer intruction analysis

6.3 Name management

[35] An 'undo' corresponds to a 'go back' to the last state where the user launched a process and to the deletion of the subtree created because of this launch.

Suggest:
- to delete prefix/suffix
- names interrelationship

6.4 Data Objects Correspondence

Suggestion of data objects correspondence is provided vhen correspondence proposals are made by the 'Find Correspondence...' functionality (see section II.5.2).

6.5 Enrichment and Transformation

Detection of specific structural patterns is helped by the CARE tool. They are for instance:
- find key
- find identifier
- find referential constraint
- find pointing relationship
- find link object
- find main object
- find multivalued property
- find compound property
- find refinement
- find gen/spec

7. Method Definition

8. Man Machine Interface

8.1 Basic Presentation: menus, selectors, etc.

8.2 Specialized Editors

The user sees information through specialized editors provided in the MMI.

- Source Browser
  A source browser is used to display source texts.
- Application Description
  A graph editor is used to display the application description graph.
- Object Editor
  Specialized graphical and/or textual editors are used to display objects of the object base such as data objects, constraints etc.
- State Tree Editor

8.3 Explanation Facility

8.4 Help Facility

a. contextual

b. passive: glossary
   active: help text updated according to current variables

8.5 Error and Warning Messages

9. Reporting

a. Schema reporting

- **Print...** (menu 'Schema')
  Prints the contents of the current schema: main and property objects are presented via indented lists, each main object list is followed by a paragraph presenting identifiers and constraints defined on this main object or any of its property object; links are referred in paragraphs following the description of any main object they relate and detailed as a whole after all main object description

b. Object reporting

- **Print Origin** (menu 'Schema')
  Prints the cross-referenced origin relationships between all the data objects of the current schema and the data objects of the corresponding 'physical' schema version

Appendix A : Customization features

1. Customize Find Correspondence

Comparison criteria	Weights	Level
Name	2	.6
Sub-properties	4	.8
Transfer instruction search	4	3
File belonging	1	
Textual origin	4	
Zone	4	
Path	4	

Global Weight:	4
Max of correspondences:	3

2. Name Analysis

2.1 Identifier subnames

Standard subnames: "ID", "NUM", "CODE"

2.2 Gen/Spec subnames

Standard subnames: "TYPE", "CAT", "CLASS"

2.3 Referencing subnames

Standard subnames: "REF"

2.4 Pointing subnames

Standard subnames: "POINT"

2.5 Delimiters

Standard delimiters: "-", "_"