MASTER IN COMPUTER SCIENCE

An exploratory analysis of an approach and prototype architecture for design transformations

Deliège, Lionel

Award date:
2004

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 2003 - 2004

An exploratory analysis of an approach and prototype architecture for design transformations

Lionel Deliége

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

Résumé

La modélisation d'une base de données complexe, efficace et sans erreur est une tâche complexe requièrant beaucoup de connaissances, tant d'un point de vue technique que d'un point de vue théorique. La méthodologie de modélisation est claire, simple, unique et efficace. Les données sont collectées et un schéma regroupant les concepts est crée. Il est ensuite transformé en un schéma logique puis implémenté dans le système final. La transformation du schéma conceptuel vers le schéma logique est une action bien définie dans la théorie et peut être faite automatiquement à l'aide d'outils. Ces outils, appelés CASE pour Computer Aided Software Engineering, sont sensés implémenter correctement la théorie des transformations. Mais cette implémentation est-elle correcte? Et que se passe-t-il quand on atteint les limites de la théorie?
mots-clés : Outil CASE, règles de transformation, OCL+.

Abstract

Modeling a complex, efficient and error free database is a complex task requiring a lot of technical and theoretical knowledges. The methodology is clear, simple, unique and efficient. Data are collected and, using these data, a conceptual schema is designed. It is transformed into a logical schema and the implementation on the final system is made.
The transformation from the conceptual schema to the logical schema is a theoretically well defined action and could be made automatically using tools. These tools, called CASE (Computer Aided Software Engineering), normally implement correctly the theory of transformations. But what about the correctness of this implementation? And what appends when we are beyond the limit of the theory?
keywords : CASE tool, transformation rules, OCL+.

Acknowledgement:

The author wishes to thank Pr. V. Englebert, Mrs. C. Moyen and Mrs. V. Detienne (University of Namur, Belgium), Pr. B. Lundell, Dr. H. Gustavsson, Mr. A. Rehbinder (University of Skövde, Sweden), Pr. B. Lings (University of Exeter, UK), for helpful discussions and supervision of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Methodology and computer science

1.1.1 An efficiency problem

At the end of the 60's, software became a important problem if nothing is done. The cost of hardware steadily declined even as hardware performance steadily increased but software seemed headed in the opposite direction. Large software projects were consistently late, over budget, and full of defects [Shapiro, 1997]. We are in the software crisis and software developers addressed the adequacy of an engineering approach in their profession [Jackson, 1990]. Since 1968 and the NATO (North Atlantic Treaty Organization) conference on Software Engineering, good development process have to follow a method. Methodologies define each step of the cycle of development of an application. Developers have to produce well defined documents and diagrams in order to formalize the development process. Classical methodologies are monolithic, every step has to be respected. The theoretical result is an error free application with a shortest time of development.

1.1.2 Following a method

But this approach is criticized. Monolithic methodologies are considered as a time consuming process with not enough result in term of software quality. According to Hughes [Hughes and Wood-Harper, 2000]:

For example, the faster 'metabolism' of today's business environment means that developers do not have the luxury of being able to follow all the detailed steps in a monolithic methodology.

Experts take their distances with methodologies, considering most of the produced documents are not mandatory.

Anyway, according again to Hugues [Hughes and Wood-Harper, 2000]:

The study indicated that less-experienced developers relied more on formalised methodologies than did their experienced colleagues. These less-experienced developers did feel that the formal methodologies provided a psychological security and the more experienced developers, whilst cynical about standards and quality, recognised the need to produce 'what the managers wanted'.

Methodologies should be viewed as a security to guarantee a good result in the applications development process. Nevertheless, the time consumed by the product of each document, the exchange of these documents between all developers of a project, the faster development required by the market make the usage of classical methodologies impossible.

1.2 CASE tool to implement methodologies

1.2.1 Computer should help to apply a method

In this context, CASE tool appears to support the development method. The potential of this software is terrible in terms of quality and productivity improvement. A citation made by Dixon [Dixon, 1992] as an example, the DuPost Corporation has '*..created over 400 programs, all failure-free, experiencing over 6:1 productivity gains.*'. But in practice, this expectation seems to be totally unrealistic. According to Butler [Butler, 2000]:

Recent research also lends support to the view that practitioners are disillusioned with CASE; Kemerer (1992) reports that 70% of CASE tools are not used 1 year after their intruduction and only 5% are widely used, but not to their full capacity. Recent *studies indicate that the situation may not be as bas as Kemerer suggests...*

Possible reasons for this are CASE tool is another technology which automates a series of design practices and tasks. CASE tool helps to use a development methodology. Logical modeling, generating documentation etc. are not made easier. Another reason is actual CASE tools implement methodology is a too strong way. The developer must follow each step defined by the tool without understanding exactly why the step must be done. Tools are not able to fit to a company and it is to the developers to fit their development processes to the tool. Tools help the developer to produce the right document to follow a defined methodology, but are not really able to fit to the development process of a company. Steps are hard-coded. If the steps of the used methodology of a company differ from the steps used in a tool, the company would have to change its methodology to adapt this one to the tool.

1.3 CASE tool assistance for database design

1.3.1 Are tools really in adequacy to the reality?

In this context, database modeling is separated from common CASE tool usage. Methodology to design a database is clear, simple, unique and easy to implement in a tool. All methods follow approximatively the same steps: firstly requirements are collected from the relevant parts of an organization, it will form a set of functional requirements affecting the database, secondly using these requirements a conceptual schema is designed, thirdly this schema is transformed into a logical schema closer of the implementation and finally the implementation of the database on the final system is made.

One of the main point in this process is the transformation applied on the conceptual schema to transform this one into a logical diagram. Number of authors considered the transformation based method is the best to get an error free and efficient database. To transform the conceptual schema, we use rules. These rules are well defined in the literature and CASE tools are able to implement these rules.

But even in database modeling, CASE tools have limits. Transformation are most of the time hard-coded, rules produce sometimes wrong results, dropping some constraints for example. No CASE tool can claim to cover any possible transformations and, even if tool provides a language to write our own transformations, this one is proprietary and rules can not be exported or used in another tool.

Transformation rules are well defined in the theory, but is this theory in adequacy with the practice based needs? Furthermore, are CASE tools that implement the theory in adequacy with the needs of its users? In this work, we will try to response to these two questions.

1.4 Paper's structure

The purpose of this document is presenting some real life database problems. Four examples present different usages of CASE tools in database design. We do a systematic exploration of tools support for forward engineering with specific schemes.

Chapter two introduces briefly the methodology used to conceive this tool evaluation. It presents how we selected our practice based schemes, the domain of each used schema and the assessed CASE tools.

Chapter three describes all tests used in the evaluation. The purpose of the evaluation is double. We want to assess the quality of the implementation of the transformation rules used by CASE tools. These rules are well defined in the theory and should be rightly implemented in each tool. We secondly want to assess the capacity of tools to apply uncommon transformation rules. An uncommon transformation is a rule not studied in the literature and required by the modeled domain. This chapter is the starting point of the work.

Chapter four presents the result of the evaluation of three CASE tools, Rational Rose, Computer and Associate ER-win and DB-main.

Chapter five introduces the work of Henrik Gustavson. It presents his transformation oriented language OCL+ and his system of active repository. Using this combination of repositories, rules and active databases, we are able to implement our own transformation rules.

Chapter six presents the result of the implementation of rules using OCL+ and the evaluation of the prototype build as proof of concept tools.

Chapter 2

Research methodology

2.1 Introduction

An important point in our research was working following a methodology in order to demonstrate we are systematic in what we claim. This project has two different objectives: firstly, using modeling of real life problems, we want to assess the capability of CASE tools to be able to model and transform schemes. If some problems are unsolvable by current tools, we present secondly another approach to model our schemes using a meta-CASE tool with a transformation oriented language.

The first part of our research was a systematic examination of the literature in order to find practice based schemes. By literature analysis, we do not mean review of existing works about transformation rules in database engineering but an examination of real problems undertaken with a specific purpose in mind. These problems are already modeled using a specific modeling language and using notations to define specific domain's constraints.

The second part of our research consist in the presentation of OCL+, a transformation oriented language and in the implementation of rules to solve our practice based examples.

2.2 Identifying the schemes

The first objective was to identify real life schemes. We do not want only *school* examples. By school examples, we mean examples solving an imaginary problem using some good theoretical practices and producing a perfect database without redundancy. These examples are not based on real specifications. These specifications are precise and without any ambiguities. Furthermore, some of these examples are made to explain theoretical constructs.

It was important for us to assess the gap between theory and practice in database engineering. Indeed, it not always possible to adapt real problem to the theory. Some constraints are impossible to express using common modeling languages. Furthermore, even if respecting the rules is the best way to produce a strong and error free solution to a problem, we want to demonstrate that is not always possible.

We selected four different papers modeling four different real life problems. Each schema uses entity relation models. This selection was made on two criteria: the schema's complexity, that means the number of entities and the number of relationships linking theses entities, and the complexity of the rules used to transform the conceptual schema into its logical schema. For each schema, the result of the analysis is a list of transformation rules. Each transformation will be tested in CASE tools to assess the quality of each tool.

2.3 Practice based schemes

We selected finally four different papers and five different schemes. For each selected schema, we present firstly its background and its studied domain and secondly we summarize the important points we extracted from.

First article [Chen and Carlis, 2003] is about representation of DNA data. The goal of this paper is to represent biologists' current understanding of their biological knowledge and to support biologists' subsequent biological discovery activities. Number of researchers characterizes biological data as more complex than business data, a biologic data is often heterogeneous in data sources (Davidson 1995), uncertain, inconsistent, and complicated [Willson, 1998]. New discoveries are regularly made and structure of data could change due to this fact.

Building a schema able to capture data efficiently is complicated and furthermore, according to the authors, quality of schemes used in biology is poor mainly because they are made by biologist them self without any knowledges in database design. The result is most of time inefficient, with redundancy and not enough constraints. Requests to extract data are slow and difficult to write.

Two schemes, 2.2 and 2.3, were extracted from this article. They were created using ER-win, the semantic of the notation is illustrated by the figure 2.1. We are mainly interested in the subschema to manage *sequence similarity search* and more precisely to the relationships between Query Set A Member, Query Set B Member, Identification Set and Pairwise Similarity Hit. The authors use foreign keys and call attributes member of these foreign keys with special names. Furthermore, the schemes presented in this article uses different relationships between the entities. Schemes are huge and the transformation from entity relation to relational model is too complex to be achieved without a tool.

The second article [Penicka and Friedsam, 2002] is about APS (Advanced Photon Source) survey and alignment database. An important number of data is needed for precise positioning of beamline components for the APS accelerator systems. These data can not be stored in raw mode because users need to access to them quickly and easily. The tool used until now (Geonet) was developed under a DOS environment and became slowly obsolete in every used domain (measure, digital system, operating system, database). The subject is specific and no commercial tool exists to help them to achieve their need. This paper presents a new database schema. Its goal of this was to produce a 3NF schemes for efficiency and saving space goal.

The selected schema 2.4 uses entity relationship notation. An important point appears in the used Is-a relationship linking Survey Point to 1D Survey Point, 2D Survey Point, and 3D Survey Point.

The Survey Point has three defined subtypes: 1D Survey Point, 2D Survey Point, and 3D Survey Point. The m indicates that Survey Point may belong to anywhere from one to many subtypes. The

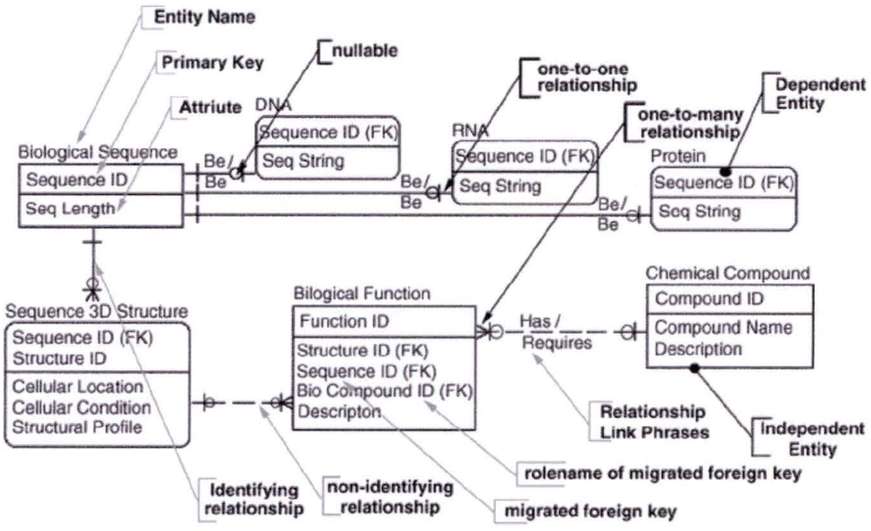


Figure 2.1: Description of the ER-win's [Chen and Carlis, 2003].

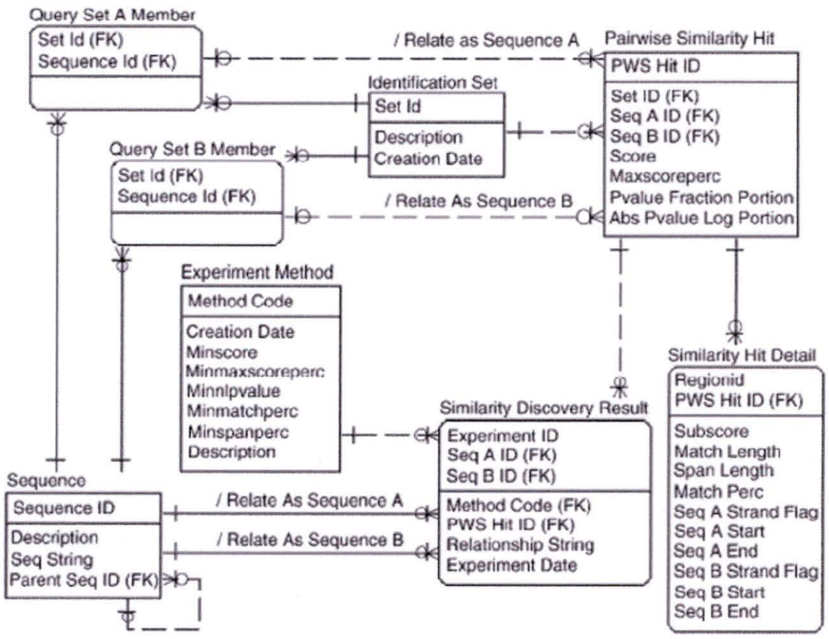


Figure 2.2: A genomic schema fragment to manage sequence similarity search [Chen and Carlis, 2003].

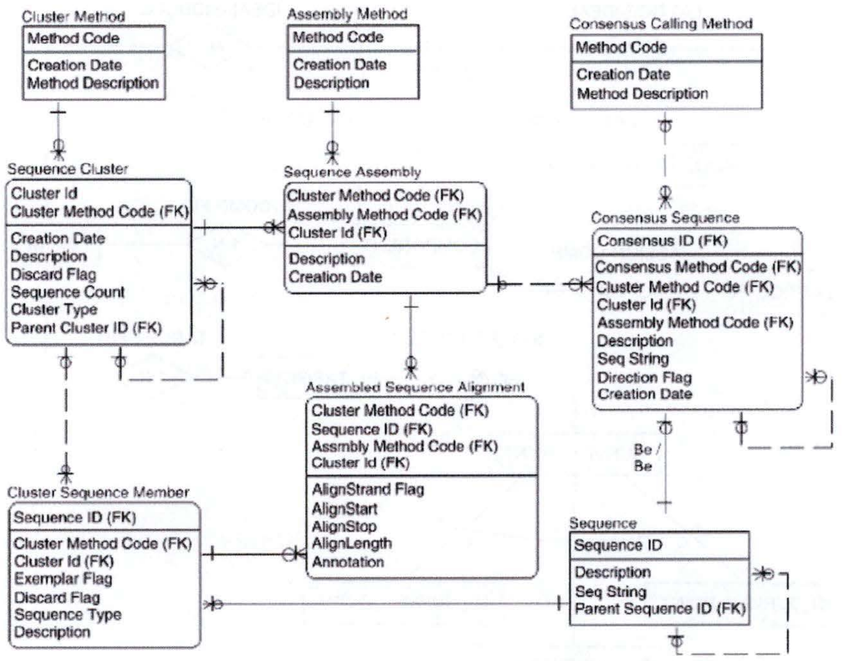


Figure 2.3: A genomic schema fragment to manage sequence clustering [Chen and Carlis, 2003].

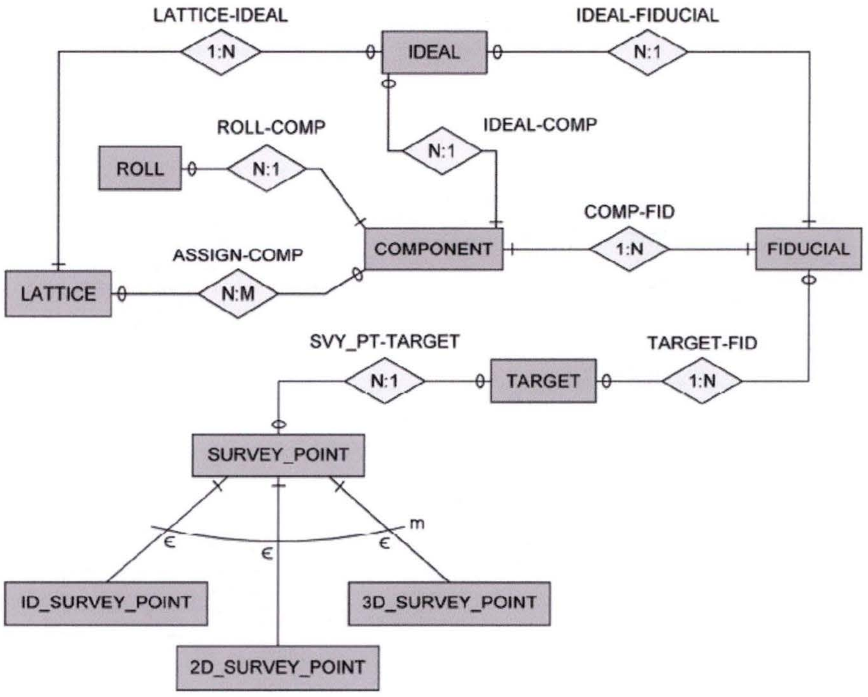


Figure 2.4: APS Entity Relationship diagram [Penicka and Friedsam, 2002].

subtypes are not mutually exclusive, because one specific instance of a Survey Point can be measured with a level instrument as a 1D Survey Point or with a laser tracker as a 3D Survey Point. This generalization hierarchy contains an IS-A relationship, which implies that the subtypes have the same identifier as the supertype Survey Point, and they can also inherit many other attributes of the Survey Point.

(..)

The 1D Survey Point, 2D Survey Point, and 3D Survey Point differ only in the number of coordinates and respective standard deviations they contain. The primary key is point ID in combination with measured date. These relations cover measured point coordinates, measuring methods, and order of survey networks. In addition, they hold hyperlinks to measurement data files, which will be stored in a hierarchical directory on a server. [Penicka and Friedsam, 2002]

The third article [Lundell and Lings, 1999] was written to expose a legacy problem. The company Skövde Systemutveckling AB (SSAB) has developed for an international corporation which is a supplier to the car-industry a system (CLARA) to support the management of non-conforming product in manufacturing. The first version of the system has been in use at the company since May 1995 and evolved as new requirements have been identified.

Schema 2.5 is an ER diagram using the Information Engineering notation. We are particularly interested in the relation between Comment, IndividualComment, MiscComment and MainActivity. Indeed, these four entities need to be merged into an unique entity for technical and legacy reasons. Existing Tools using the database are made to use this unique entity and to guarantee that every constraint is respected.

Last article [Kolp and Zimanyi, 2000] uses a school example. Schema 2.6 is used in the paper to test a system of schema optimization using prolog. This schema is interesting for two reasons: it introduces recursive relation-

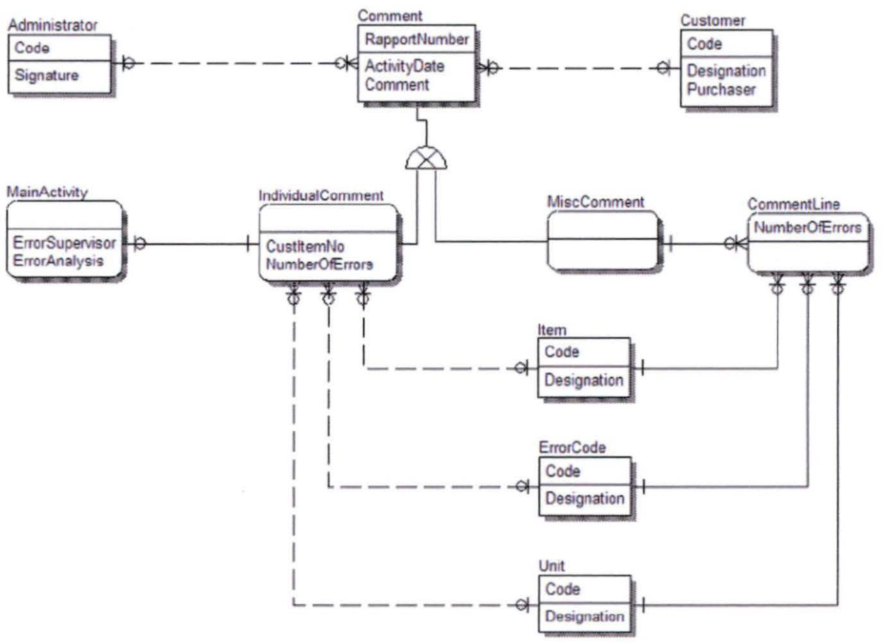


Figure 2.5: CLARA entity relationship diagram (redrawn in English)

ships between the entities and uses different kind of basic relationships that we did not found in the other papers. CASE tool is an implementation of database design theory and we want to assess if this implementation is correct or not. We extract from this example all kinds of construct we did not find in the other selected papers.

2.4 Methodology used for tool's assessment

2.4.1 Designing a database

Designing a database always follows the same methodology. First, the developer draws the concepts, called the conceptual schema. It is an abstract view of the problem, a high level design. The conceptual schema does not keep out of any implementation tricks and of the platform's destination. Different models exist to represent a conceptual schema, for the moment, ER (Entity Relationship) and UML (Unified Modeling Language) are the most used.

In this schema, the developer draws the entities, the attributes, chooses which attributes identify entities (identifiers are not mandatory but highly advised). The developer specifies too the relations between the entities and the cardinalities of these relationships.

After that, the conceptual schema is transformed it into a logical schema. Logical schema keeps out of the implementation and platform's destination. Actual databases use the relational model and schemes are coded using SQL language. The relationships defined in the conceptual schema are mainly transformed into foreign keys. Another possible transformation is to merge entities part of the relation. Identifiers are transformed into primary key and some constraints are added to guarantee the new schema represents exactly the same thing than the conceptual schema.

2.4.2 Drawing the conceptual schema in a tool

Designing a database with a CASE tool is different from one program to another. In some cases, we have to enter all data when the conceptual schema is drawn. Users choose which transformation has to be applied on each kind of relationship. Some other tools ask when the transformation's process is launched which transformation has to be applied on every kind of relationship.

Nevertheless, whatever the used tool, designing a database in a tool has to follow the two same steps. Firstly, developer has to enter all data and to build his conceptual schema. Developers need to specify every information used by the tool to transform a conceptual schema into logical schema without any ambiguities. This step is based on an identification of the needs. Secondly, developers effectively apply the transformations on the schema.

2.4.3 Transforming from conceptual schema into relational model schema

Transformation is a strong theory and should be totally automatized. But in order to automatize the transformation process, user has to add enough data and has to have a high knowledge of the tool and transformations' theory. To help the user to apply the transformations, some CASE tools like ER-win prefer to unautomatize the process and to ask the users for transformation that has to be applied on each construct or relationship using graphical wizards.

From our point of view, the transformation process has to be totally automatized. Conceptual schema are compound by many entities and many relationships. Drawing the conceptual schema and identifying the needs are complex steps. Transformation process needs a global view of the schema to be applied properly. Graphical wizards hide the schema and only show the problematic relationships. Users should be able to define which transformation shall be used to transform each relationship.

2.4.4 Analyzing the result

For each studied transformation, we write the result of the transformation in a table. This table records the name of the transformation, troubles we met when we drew the conceptual schema, troubles we met when we transformed the conceptual schema into the relational model and finally a small note records additional information. Result tables are listed in annexe B. For each line of the result table, we analyse the problem. Using these conclusions, we are able to demonstrate if CASE tools are in adequacy with practice based examples or not.

The next chapter presents for each studied transformation the result of the assessment. The assessment has two goals: firstly we check if the tools were able to transform the schema without asking any new information when the transformation process is launched, secondly we check if the relational model and the produced SQL code are correct.

2.4.5 Commercial CASE tools used

We decided to assess three different tools, with different characteristics.

ER-win version 4.1 uses two different models for logical and physical notation: IDEF1x [IDE,] and Information Engineering [James and Finkelstein, 1981]. For the moment, this tool is developed by Computer and Associate. The purpose of the tool is only database design. The transformation process is divided in two steps: first user draws conceptual schema and decides which transformation has to be applied on the sub-type relationship and secondly, a graphical wizard helps user to transform the conceptual schema. There is no language to create our own rules. The system is based on two synchronized repositories. If a modification is made in one model, this modification is automatically reflected on the other model.

DBmain is a forty persons/year project of the university of Namur directed by Jean-Luc Hainaut. This tool uses the ERA notation and is able to create, store and transform conceptual and logical schemes. It provides too a strong system of transformation rules with the possibility to script the transformation process. Its repository can be updated to be able to store new information or new models. It includes too a proprietary language, Voyage 2, to extend the functionalities of the tool. We use the version 6.5.

Rational Rose is our last tool. Part of the rational process, it uses UML for modeling language. Unlike DB-main and ER-win, the purpose of the tool is not only database design but the whole development of an application. Rose uses two separate repositories without link between them. When a schema is exported in the data modeler, modification of the conceptual schema does not change anything in the logical schema. So, each modification of the conceptual schema induces the regeneration of the logical schema.

For each tool, we explain the way we use it in the annexe A. The purpose of this annexe is not to explain how to use a tool, but to explain how we use each tool. The distinction is important, we want to provide every information in order to demonstrate we are systematic in our method of assessment. Using these information, everybody could reapply our assessment and produce the same results.

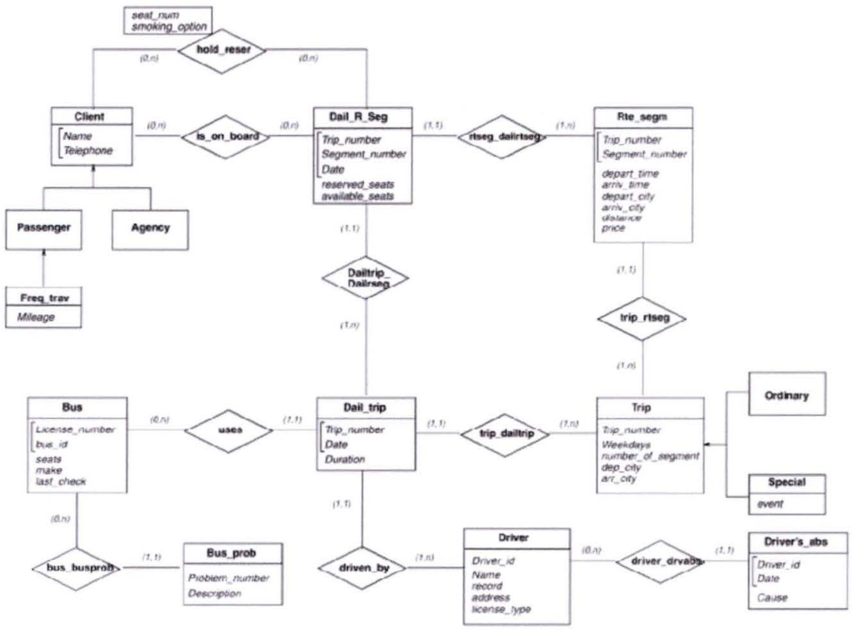


Figure 2.6: Bus example (Kolp 2000).

Chapter 3

Transformation rules and commercial CASE tools

3.1 Classical transformation rules

3.1.1 What do we call Classical transformation rules?

Transformation based approach in database design is considered now by number of authors as a good practice to transform an abstract specification into a correct and efficient database structure. With the analysis of database requirements, developer builds a conceptual schema. This conceptual schema is subsequently transformed into logical schema and implemented in the final system. A transformation rule is a correctness preserver operator, that means the schema resulting of application of the transformation rules on a conceptual diagram expresses the same thing than the original diagram.

Between conceptual and logical schemes, transformations have to be applied. In the DB-theory, the transformation concept can be defined as follows:

> *A Schema transformation is an operator that applies on a construct C of a schema S, and that replaces it with other constructs C', leading to new schema S'. C' is the target of source construct C through T, i.e. $C' = T(C)$.*
>
> *(..)*

To define transformations more precisely, we need a second mapping t, that specifies how valid instances of construct C are translated into C' instances: if c is an instance of C, then $c' = t(c)$ is an instance of C'.
[J-L. Hainaut and Roland, 1996]

A special class of transformations is semantics-preserving. Through the transformation, no semantical information is lost. Using a semantics-preserving transformation T, and a schema R, we obtain by the application of T on R a new schema R', i.e. $R' = T(R)$. Each instance of R should be recovered from an instance of R' using an algebraic or procedural operators.

A higher class of transformations is symmetrically reversible. Every instance of C can be expressed in C' using mapping t and each instance of C' can be expressed in C using the opposite mapping t'.

In order to limit the scope of this work, we choose to divide symmetrically reversible transformations into four new classes, Zero to many, Zero to one, Many to many and Is-a. Each one denotes a set of transformations with common characteristics. They work on relationships with the same maximal cardinalities, and, through a same class, variations are made on the minimal cardinalities. Each relationship could be transformed using different rules. We decide to transform relationships using only transformation rules found in the selected practice based example presented in the previous chapter.

Identifying versus non identifying relationship

In classes Zero to many and Zero to one, the relationship could be identifying or non identifying. That means, considering the relationship R between the entity Parent and Child, if R is an identifying relationship, an instance of Child is identified by an instance of Parent and by zero or more of its attributes. Figure 3.1 illustrates an example of identifying relationship's instances. A and B are linked by a Zero to many identifying relationship R.

Each CASE tool uses its own representation for an identifying relationship. DB-main adds the name of the relationship into the identifier of the entity, in opposite to ER-win and Rational Rose which draw the identifying relationship with a continuous line instead of a dotted line.

These different views have some advantages or disadvantages: the DB-main system forces user to build completely the identifier of an entity in one action, by selecting all attributes and relationships which are part of the identifier. This system produces for us better schemes, user does not make mistake when he creates the identifier.

The ER-win and Rational Rose system could cause mistakes in the schema. For example, a child entity should have only one zero to many identifying relationship as identifier. With only one zero to many relationship as identifier, all instances of the child entity must reference a different instance of the parent entity. This constraint matches against the cardinalities of the relationship.

3.1.2 Zero to many relationship

Identifying relationship

A zero to many identifying relationship links two entities A and B. Each instance of A can be referenced by zero, one or many instance(s) of B and each instance of B must reference one and only one instance of A. Figure 3.2 illustrates allowed and disallowed instances.

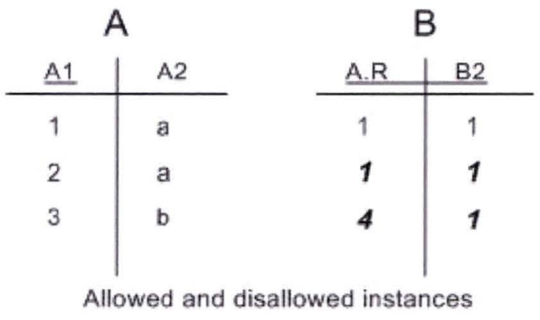
Identifying means this relation is a part of the B's identifier with other attributes or relationships. For example, if the B's identifier is made up of the attribute $b1$, for all instances $I1$ and $I2$, if $I1.b1 = I2.b1$ then $I1.R$ must reference another instance than $I2.R$.

The usual transformation 3.3 to implement this relationship into relational model consists in, for all attributes of A's identifier, adding these attributes into entity B referencing the entity A. These new attributes are mandatory and are part of the primary key group with other B's attributes.

Non-Identifying relationship

A zero to many non identifying optional relationship links two entities A and B. Each instance of A can be referenced by zero, one or many instance(s) of B and each instance of B can reference zero or one instance of A.

The usual transformation 3.4 used to implement this relationship into relational model consist in, for all attributes of A's identifier, adding these



A

A1	A2
1	a
2	a
3	b

B

A.R	B2
1	1
1	***1***
4	***1***

Allowed and disallowed instances

Figure 3.1: Identifying relationship instances example

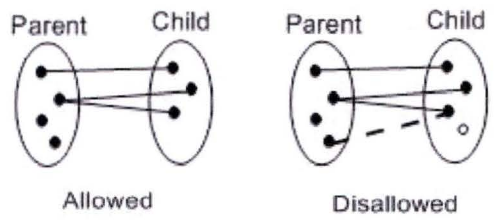


Figure 3.2: Zero to many's instances example

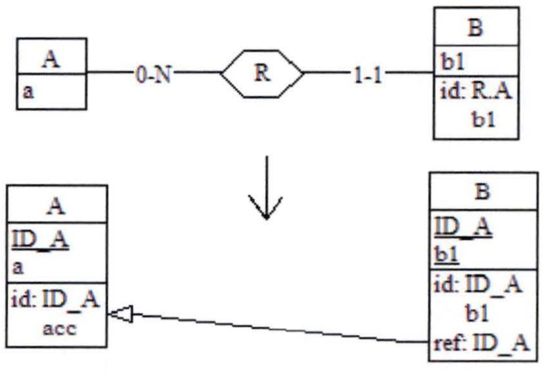


Figure 3.3: Zero to many identifying relationship

attributes into entity B referencing the entity A. These new attributes are optional.

Variations

Some variations on minimal cardinalities can be made and combined. The non identifying relationship can be mandatory instead of optional. In this case, each instance of A can be referenced by zero, one or many instance(s) of B and each instance of B must reference one and only one instance of A. The usual transformation used to implement this relationship into relational model is similar to the optional relationship, but the attributes members of the foreign key group are mandatory.

Both relationship (identifying or non identifying) can be mandatory for the parent entity. That means each instance of A can be referenced by one or many instance(s) of B. Implementing this constraint directly in the relational model is impossible. The only way is by adding a constraint in the target database. In SQL, it can be made using a trigger or a check.

3.1.3 zero to one relationship

Identifying relationship

A zero to one identifying relationship links two entities A and B. Each instance of A can be referenced to zero or one instance of B and each instance of B must reference one and only one instance of A. Furthermore, the relation is a part of B's identifier. For example, if the B's identifier is made up of the attribute $b1$, for all instances $I1$ and $I2$, $I1.R$ must reference another instance than $I2.R$. Figure 3.5 illustrates allowed and disallowed instances.

The usual transformation 3.6 used to implement this relationship into relational model consist in, for all attributes of A's identifier, adding these attributes in entity B referencing the entity A. These attributes are mandatory and are added to the B's primary key. Furthermore, the group of attributes constituting the foreign key is unique. That means if the primary key of A entity is made up of two attributes $a1$ and $a2$, the union of $B.a1$

and $B.a2$ referencing each instance of A entity must be different for each instance of B.

Non-Identifying relationship

A zero to one non-identifying relationship links two entities A and B. Each instance of A can be referenced by zero or one instance of B and each instance of B can reference zero or one instance of A.

The usual transformation 3.7 to implement this relationship into relational model consists in, for all attributes of A's primary key, adding these attributes in entity B referencing the entity A. These attributes are optional and the group of attributes constituting the foreign key referencing entity A is unique.

Variations

Variations could be made on the minimal cardinalities. The non-identifying relationship can be mandatory for the child entity. In this case, attributes part of the foreign key are mandatory. For both relationships, the minimal cardinalities for the parent entity can be one and only one. This constraint can not be expressed directly in the relational model and must be implemented by a trigger in SQL to guarantee each instance of A is referenced by an instance of B.

Variations can be made by the way that we transform the relationship. Usually, the studied example transformed the relationship using a foreign key, but merging the child entity with the parent entity could be, in some cases, more efficient. It is the case when the parent entity must have one and only one child which references each of its instances. Merging the entities consist in transferring all attributes into the parent entity and adding constraints on the attributes, depending of cardinalities of the original relationship.

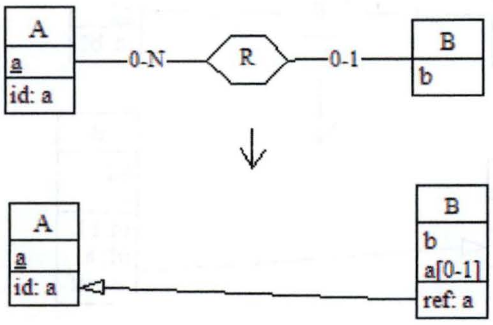


Figure 3.4: Zero to many non identifying relationship

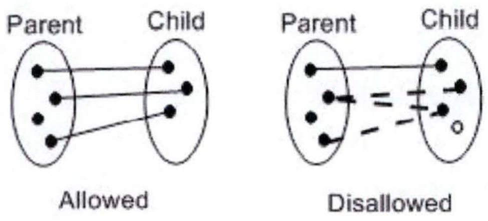


Figure 3.5: Zero to one's instance example

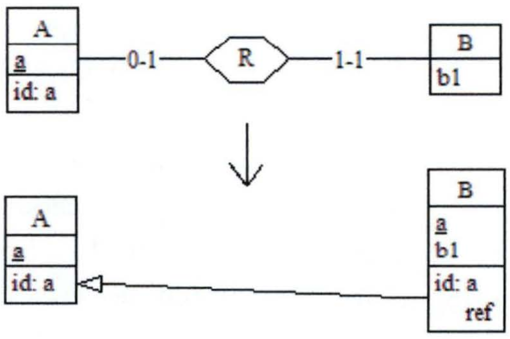


Figure 3.6: Zero to one identifying relationship

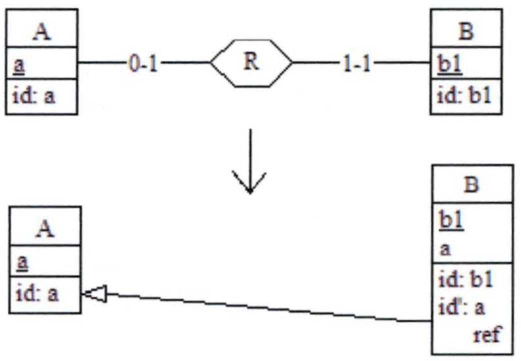


Figure 3.7: Zero to one non-identifying relationship

3.1.4 Many to many relationship

Relationship

Many to many relationship links two entities A and B. Each instance of A references zero, one or many instance(s) of B and each instance of B references zero, one or many instance(s) of A. Figure 3.8 illustrates the allowed and disallowed instances for a many to many relationship and one of its variation.

It is impossible to implement directly this relationship using foreign keys. The most usual implementation illustrates in figure 3.9 is the addition of a entity R. This entity is linked with two zero to many relationships to entities A and B.

Variations

Some variations can be made on minimal cardinalities inducing new constraints. The minimal cardinalities, zero at the outset, mean each linked entity can reference zero, one or many instance(s) of the other entity and vice versa. But in some cases, this minimal cardinality is one, that means each linked entity can reference one or many instance(s) of the other entity.

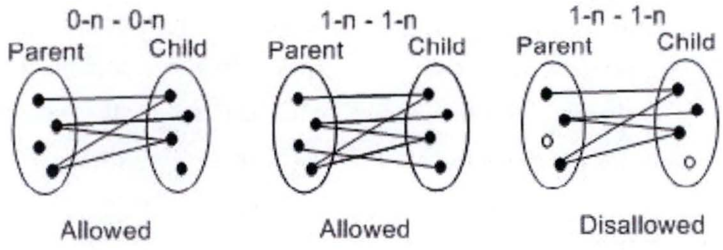


Figure 3.8: Many to many's instances example

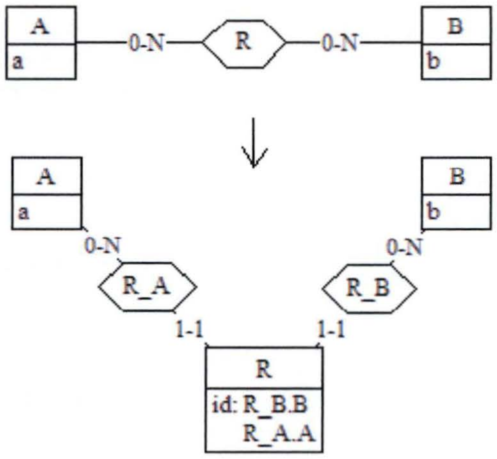


Figure 3.9: Many to many relationship's transformation

3.1.5 Is-a relationship

Is-a relationship is probably one of the most studied and one of the most complicated relation we can meet in a conceptual schema. Is-a relationship can be divided into four categories [J-L. Hainaut and Roland, 1996]. Firstly, the relation can be total or partial. In a total Is-a relationship, each instance of the parent entity must be referenced by an instance of at least one of its children. Secondly, the relation can be disjoint or overlapping. In a disjoint Is-a relationship, each instance of the parent entity can be referenced by one and only one instance of one of its child. That means, if the parent entity A has two children B and C, if an instance of B references the instance $a1$ of A, there is only one instance of B which references $a1$ and no instance of C references $a1$.

Mixing up these two variations, we have a Total-Disjoint relationship, a Partial-Disjoint relationship, a Total-Overlapping relationship and a Partial-Overlapping relationship. Each variation can be transformed in different way into the relational model. Conceptual schemes is illustrated by figure 3.10, examples of logical transformation is illustrated by figure 3.11.

Partial Disjunctive relationship

An is-a disjunctive relationship is a relation between a super-type entity and one or more subtype(s). Disjunctive relationship means no parent entity can have the same value as any B or C entity, and so on for B and C.

There is two ways to implement this relationship. First, child entities reference the parent entity using a unique foreign key. A constraint must be added in the relational model to guarantee every instance of every child references a different instance of the parent entity. Such a constraint can not be implemented directly in the relational model and needs a trigger to be implemented in SQL.

The other way to implement this relationship is to merge every child into the parent entity. The child's attributes are grouped into optional group and a constraint is added to guarantee the original is-a constraint is respected. Such a constraint can not be implemented directly in the relational model and needs too a trigger to be implemented in SQL.

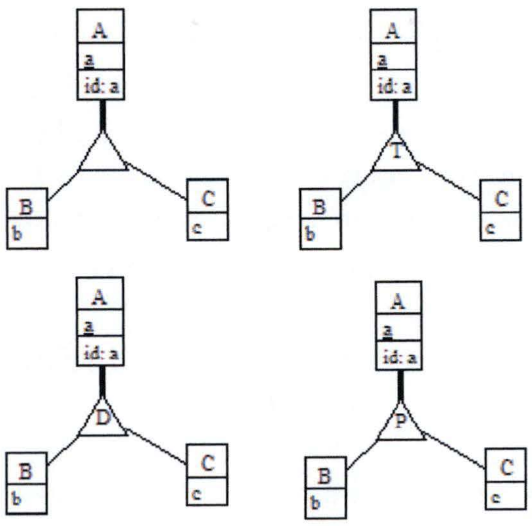


Figure 3.10: Type of Is-A conceptual schemes

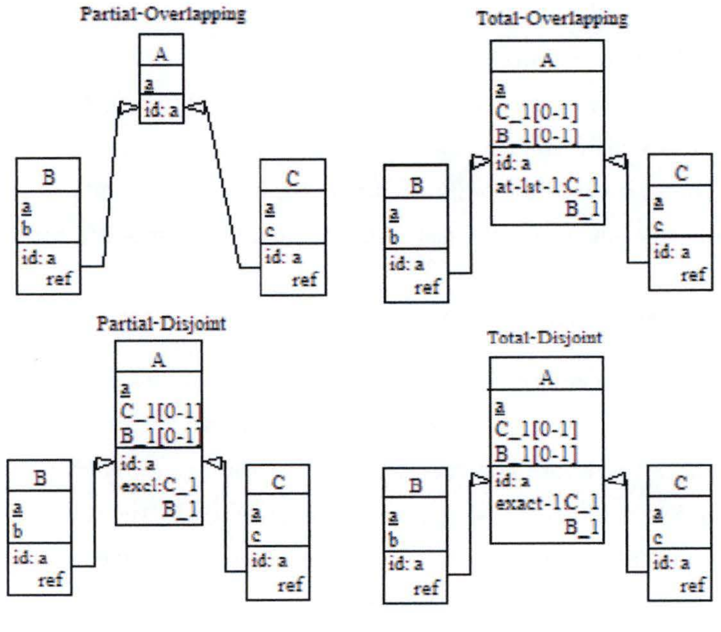


Figure 3.11: Type of Is-A logical schemes

Partial Overlapping relationship

An Is-a partial overlapping relationship is similar to the previous relationship but there is no constraint between the child entities.

There is two ways to implement this relationship. First, child entities reference the parent entity using a unique foreign key. The other way to implements this relationship is to merge every child into the parent entity. The child's attributes are grouped into optional group.

3.2 Uncommon and practice based transformation rules

3.2.1 What is an uncommon and practice based transformation rule?

Theoretical transformations try to cover every case which developer could meet when he develops a database. But these transformations are not always in adequacy with the real life. To demonstrate this hypothesis, we have read some practice based schemes and analyse the applied transformations.

This analysis shows three categories of transformations that are not studied in the literature or not implemented in CASE tools. We do not claim these categories cover each possible case of uncommon transformations, but we claim there is some uncommon transformations. If actual CASE tools are not able to transform schemes rightly, it could cause some problems.

3.2.2 Limitation due to the conceptual models

Limitation can appear due to the conceptual models. As we explained before, the transformation process has to be applied automatically. Users should be able to choose the transformation to apply on each relationship and eventually to add additional informations needed by the transformation.

The biological example illustrates this problem. The schema was drawn using ER-win. This program has the particularity to be able to mix the conceptual and logical models. User is able to see the migrated attributes when he draws the conceptual schema and is able to change the name of these attributes. Furthermore, if entities A and B have only one attribute ID as identifier, and if entity C is linked to A and B with a zero to many relationship, ER-win adds one attribute ID in the entity C to implement this relation and this attribute is added to two different foreign key groups. This example is illustrated by schema 3.12, schema 3.13 and SQL code 3.1. This is a characteristic of the semantic used in the model IDEF1x.

Using this special notations, the authors of the example express a new constraint. Conceptually, this constraint can be described as follows:

> Each instance of Pairwise Similarity Hit, Query Set A Member and Query Set B Member must reference an instance of Identification Set. Each instance of Pairwise Similarity Hit can reference an instance of Query Set A Member and Query Set B Member. If an instance *psh* of Pairwise Similarity Hit references an instance *qsa* of Query Set A Member and *qsb* of Query Set B Member then *qsa*, *qsb* and *psh* references the same instance *is* of Identification Set.

The proposed implementation 3.14 consists in adding only one attribute Set ID in the entity Pairwise Similarity Hit to implements the three foreign keys.

To transform correctly this schema, the user has to be able to choose the name of attributes implementing the relationship.

3.2.3 Unstudied transformation rules

An unstudied transformation rule is a variation of an existing transformation (in our example, a temporal is-a relationship).

Our example is based on a is-a partial overlapping relationship with a temporal aspect. A temporal database records present and previous states of the application domain. To achieve this goal, each modification of data must be recorded with timestamps.

> *If an entity type is temporal, then, for each entity that existed or still exists, the birth and death instants (if any) are known (valid time), and/or the recording (in the database) and erasing*

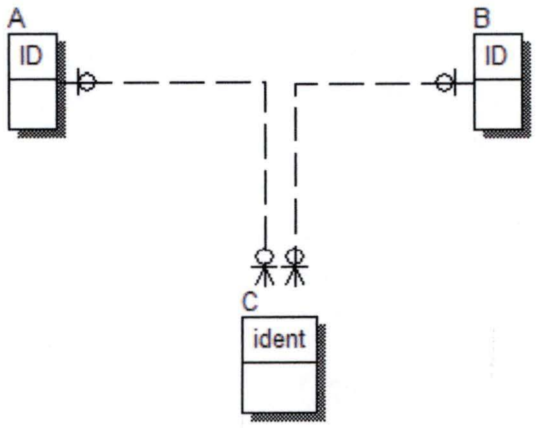


Figure 3.12: ERwin's Conceptual Schema

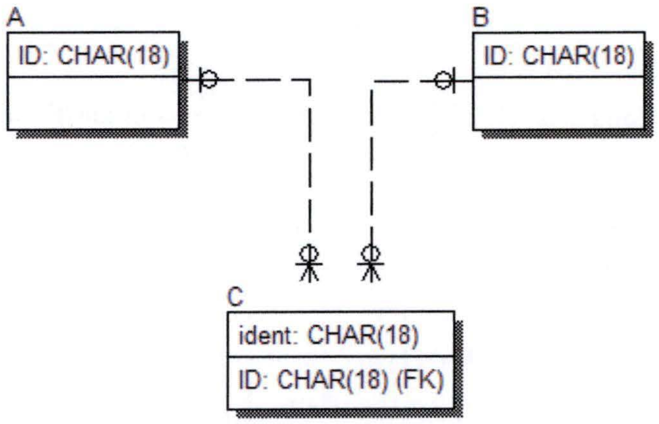


Figure 3.13: ERwin's Logical Schema

```
CREATE TABLE A (ID CHAR(18) NOT NULL PRIMARY KEY (ID));
CREATE TABLE B (ID CHAR(18) NOT NULL PRIMARY KEY (ID));
CREATE TABLE C (
ident CHAR(18) NOT NULL,
ID CHAR(18) NULL
PRIMARY KEY (ident)
FOREIGN KEY (ID) REFERENCES A
FOREIGN KEY (ID) REFERENCES B);
```

Table 3.1: ERwin's SQL code

instants (transaction time) are known. This information is implicit and is not part of the attributes of the entity type. If an attribute is temporal, then all the values associated with an entity are known, together with the instants at which each value was (is) active. The instants are from the valid and/or transaction time dimensions according to the time-tag of the attribute. If a relationship type is temporal, then the birth and death instants are known. The two time dimensions are allowed, according to the time-tag. [Detienne and Hainaut, 2001]

The article does not specify if we are in valid or transaction time but the transaction time is useless in this domain. For each modification, we need a timestamp to record when the new data has been inserted in the database. Each instance of 1D Survey Point, 2D Survey Point or 3D Survey Point must reference a parent instance in Survey Point. A parent instance can be referenced by one or more instance(s) of its children.

Common implementation of temporal databases uses two timestamps to store the date of an instance (starting and ending date). Between these two dates, nothing changes in the instance. For example, using a database recording every information about workers, the temporal table stores the name, the address and the department of the worker. If one of these informations changes, a new instance is created, the lasted old instance receives the date of today as the ending date and the new instance receives the date of today as starting date.

In our example, such a reasoning is false. To position the beamline, the same points are measured repeatedly many times by many different methods. Between two dates, the value of the lasted recorded instance is different and could change. We need to take a picture at regular moment to see the evolution of the domain. We need one timestamps to record the date of each instance. Figure 3.15 and figure 3.16 illustrate the applied transformation.

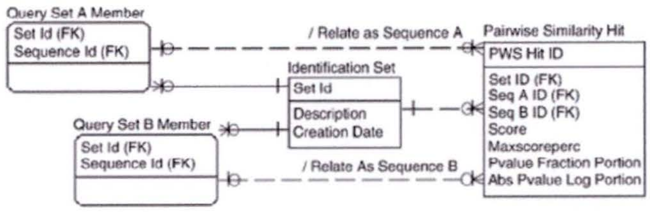


Figure 3.14: Biological example

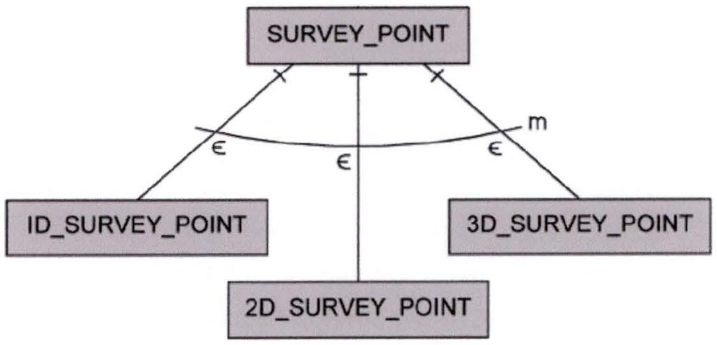


Figure 3.15: Is-a temporal entity relationship diagram

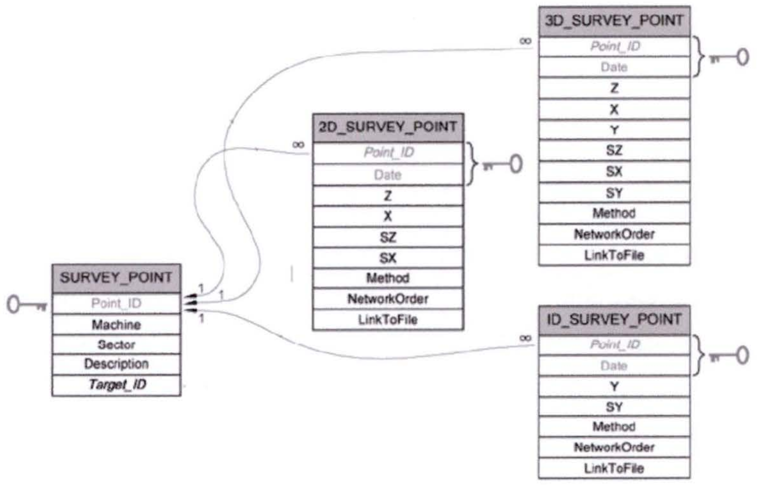


Figure 3.16: Is-a temporal relational model diagram

3.2.4 Information lost

Until now, all applied transformations have to be symmetrically reversible. But sometimes, this constraint is too strong and the developer accepts to loose semantical information when he applies the transformation rules on the conceptual schema. For example, the lost constraint could be implemented directly in the software and does not need to be implemented in the database.

Our example is a legacy problem. The original schema, made for an old database engine (paradox) was optimized and transformed to be usable on the final system. The original conceptual schema was kept but the logical schema is lost. The first idea to get the logical schema was to retro-engineer the paradox database to reuse this database schema [Lundell and Lings, 1999], but no tool at that time was able to do this action correctly.

This test consists in merging four entities: Comment, IndividualComment, MiscComment and MainActivity. No constraint is added to respect the cardinalities of the original relationships, every constraint is already implemented in tools using the database. Figure 3.17 and figure 3.18 illustrate the applied transformation.

3.3 CASE tools

3.3.1 What is a repository?

Purpose of CASE tool is helping user to conceive a software. It provides tools to share data between developers, notations to describe the behavior of the application, notations to describe the classes, the database, etc. To achieve this goal, CASE tool uses different notations, different models. These models are the description of the models. They are the underlying notations. We call these notations the meta-model. It contains the type definitions for the different data items used in the models. It describes what is an entity or an attribute, relationship between an entity and its attributes etc. But describing an application is useless if tool does not provide anything to store this description. Case tools need a kind of database able to store models without loosing information. We call this database the repository.

The meta-model describes every characteristics of every items that user

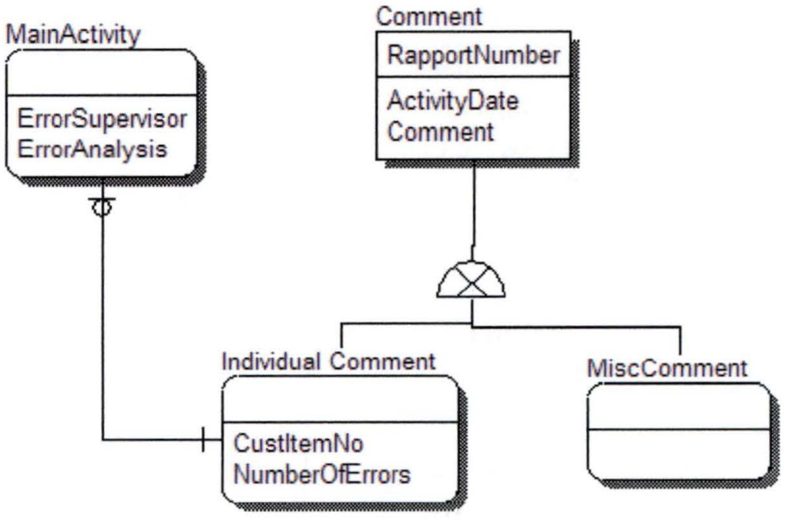


Figure 3.17: Legacy Entity-Relation model

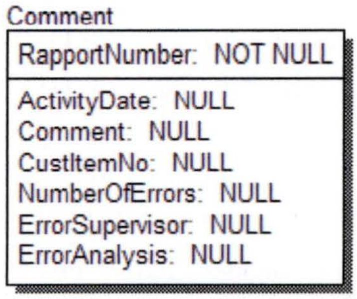


Figure 3.18: Legacy Relational Model with semantical informations lost

could use to describe his model. A repository has a meta model to describe the various types of information which it can store. These types of information can be high level concepts common to every model (an attribute has a name) or can be specific to a tool and is stored for internal reason by the tool. A repository is a database, it must provide access' method to retrieve stored data easily, probably support data versions and security restriction as a classical database.

In our work, we limit the usage of the repository to its storing purposes. We do not study the meta data exchange, the security restriction or the versioning system.

3.3.2 Transformation rules in CASE tools

The first step of the conception of a database consists in collecting the requirements from the relevant parts of an organization. It forms a set of functional requirements affecting the database application, a set of database requirements affecting the design of the database. These requirements are used to form a conceptual schema of the system. Ideally, the conceptual schema does not contain implementation details, and can therefore often be understood by less technically oriented users. This conceptual schema is mapped to a logical schema. This schema contains every implementation details.

Mapping the conceptual schema to the logical schema is made using transformation rules, as we already explain in the previous sections. CASE tools provide system to help this mapping, by implementing transformation rules. The transformation rules, using the conceptual schema stored in the repository, create the logical schema and store it into the repository. Ideally, this action is automatic, user does not have to do anything. Furthermore, the schemes have to be synchronized, if a modification is made in the conceptual schema, this modification must be reflected on the logical schema.

In this work, we pay attention to two characteristics of transformation rules: rules must be applied automatically, using every information stored in the repository by the user and must provide the right result regarding transformation's theory.

Chapter 4

CASE tools evaluation

4.1 Introduction

Three CASE tools based on different meta models were used for this evaluation. DB-main and ER-win are tools on ER model. Their purpose is only databases design. Rational Rose is based on UML. Its purpose is the whole cycle of development of an application. It implements the Rational Process' methodology.

These programs have different ways to transform a conceptual schema into a logical schema. DB-main is able to mix the relational notations with the ER notations. To transform schemes, developer can transform by hand every relationship and therefore choose the best transformation rule for each relationship. Users could too build a script to transform automatically every relationship meeting a specific precondition.

ER-win works with a system of double linked repositories. Every modification made on a schema in the ER model is automatically reflected on the schema in the relational model. Nevertheless, two exceptions are made to transform Many to many relationships and Is-a relationships. Firstly, the user can decide if subtypes have to be merged with their super type. Secondly, a wizard helps the user to choose which rule has to be applied on Many to many relationships and on non merged Is-a relationships. Other relationships are transformed automatically using foreign keys. The transformation process is so divided into two independent phases.

Transformation process in Rational Rose is totally automatic. The application adds automatically a technical identifier in each transformed entity. All relationships are transformed using foreign keys. Rose is not able to merge entities. There is no link between a conceptual schema and its transformed version. If a modification is made, the whole schema must be reexported.

4.2 Classical transformation rules

4.2.1 Identifying vs non identifying relationship

ER-win and Rose use a different notation to make the difference between an identifying relationship (continuous line) and a non identifying relationship (doted line). ER-win automatically disallows cardinalities that are against the identifying concept. An identifying relationship can not be optional for the child entity. Rose allows controversial cardinalities and is able to produce an *optional primary key* as illustrates in figure 4.1 and table 4.1.

DB-main does not use a different notation to make the difference between identifying and non identifying relationship. A relationship can be a part of the identifier of an entity. DB-main checks the cardinalities and does not allow controversial cardinalities. Furthermore, DB-main does not allow an identifier conflicting with the cardinalities of the relationship. That means a zero to many relationship can not be the only identifier of an entity and a zero to one relationship must be the unique identifier of an entity.

4.2.2 Zero to many relationship

Identifying and non-identifying relationship

Every tool was able to transform zero to many identifying and non-identifying relationship. In all cases the transformation and the produced SQL code are correct.

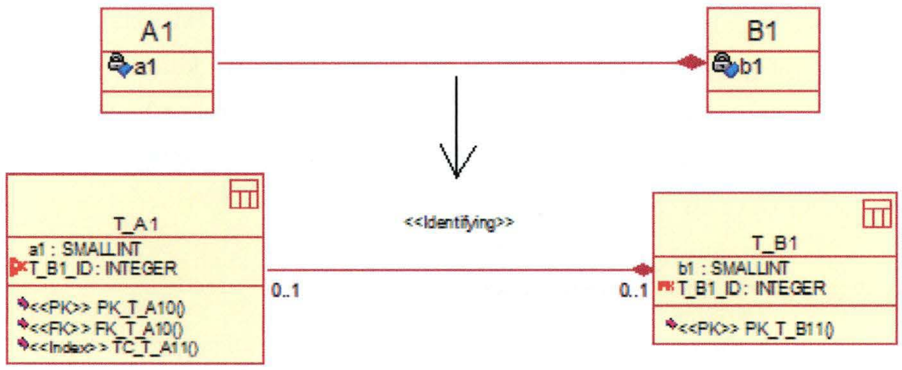


Figure 4.1: Transformation of an optional identifying relationship in Rational Rose

```
CREATE TABLE T_A1 (a1 SMALLINT NOT NULL, T_B1_ID INTEGER,
PRIMARY KEY (T_B1_ID) );
CREATE TABLE T_B1 (b1 SMALLINT NOT NULL, T_B1_ID INTEGER
NOT NULL, PRIMARY KEY (T_B1_ID));
```

Table 4.1: Rational Rose and optional primary key

Variations

Almost variations of zero to many relationship do not cause any troubles. Nevertheless, troubles appear when the relationship is mandatory for the parent entity. Rational Rose and ER-win transform correctly this relationship into the relational model, using a notation to specify this relationship is mandatory but this constraint does not appear in the produced SQL code. DB-main transforms the relationship correctly, using its own notation to represent the constraint ('equ' next to the foreign key group) and adds a trigger in the SQL code.

4.2.3 Zero to one relationship

Main characteristic

To implement a zero to one relationship, tools have to add a constraint of uniqueness on the foreign key implementing the relationship. Rational Rose and DB-main automatically add this unique constraint and produce the right SQL code.

An important problem illustrated by figure 4.2 and table 4.2 appears using ER-win. There is no difference between a zero to one relationship and a zero to many relationship. To guarantee the constraint of uniqueness, user has to add in the conceptual schema an alternate key group. This group contains only the attribute member of the foreign key. The problem is double, firstly the conceptual schema does not have to show migrated attributes and secondly the information about the uniqueness is already present in the cardinalities of the relationship and this alternate key group is redundant.

Variations

The variations on the minimal cardinalities are similar to the variations of zero to many relationship. The problem that occurs when the relationship is mandatory for the parent entity appears again in Rational Rose and ER-win. No constraint is added in the SQL code.

Foreign key is the only way to implement zero to one relationship in

ERwin and in Rational Rose. These tools can not merge entities linked by such a relationship.

DB-main supports merging of entities and puts the moved attribute together in an optional group. All attributes become optional and a constraint guarantees that if an attribute member of the group get a value, other attributes member of the group can not be *null*.

4.2.4 Many to many relationship

Rose, ER-win and DB-main do not have any problems to transform this relationship. Anyway, a problem appears again in ER-win. This tool does not support a many to many relationship mandatory for one or both entities. Only the 0-N - 0-N is supported by this tool.

Rational Rose ignores totally the difference between 1-N - 1-N cardinalities and 0-N - 0-N cardinalities. The result of the transformation is the same for both relationships. Rose creates a new table to implement the relationship and links this new table to the entities using 0-1 - 0-N relationships as illustrated in figure 4.3.

Another problem appears in Rational Rose when two many to many relationship link the same entities. For a unknown reason, the program implements both relationship using only one table. This transformation is probably an optimization, each relationship does 'the same thing'. Nevertheless, this optimization is wrong because the purpose of these relationships is different.

4.2.5 Is-A relationship

Partial Overlapping relationship

Partial overlapping Is-a relationship is supported by every tool. The transformation, using an unique foreign key, is correct and the produced SQL code does not suffer of any problems.

Rational Rose does not support merging of entities. ER-win can merge two entities linked by an partial overlapping is-a relationship but does not add any constraints on the merged attributes. These attributes have to be optional and have to be put together in a group to guarantee that if one

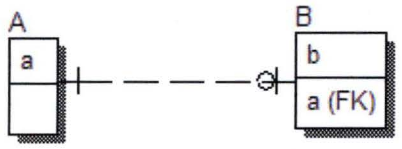


Figure 4.2: Zero to one relationship in ER-win

```
CREATE TABLE A (a CHAR(18) NOT NULL,
PRIMARY KEY (a));
CREATE TABLE B (b CHAR(18) NOT NULL, a CHAR(18) NOT NULL,
PRIMARY KEY (b), FOREIGN KEY (a));
```

Table 4.2: SQL code for a zero to one relationship in ER-win

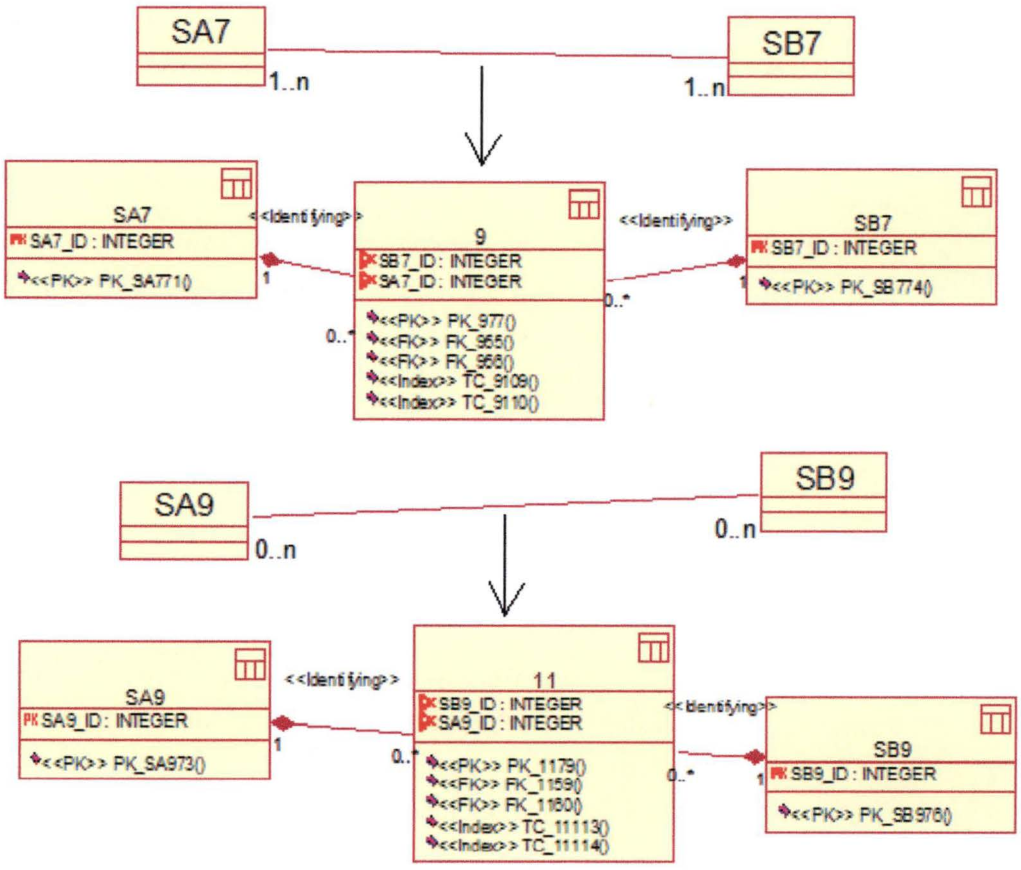


Figure 4.3: Many to many in Rational Rose

of these attributes gets a value, the other must not be *null*. DB-main can merge entities and adds the needed constraint on the merged attributes.

Partial Disjoint relationship

Partial Disjoint relationship is supported only by ER-win and DB-main. In both tools, the relationship could be transformed into an unique foreign key or could be merged into an unique table. In both cases, a constraint must guarantee the uniqueness of each instance.

Nevertheless, even if ER-win uses a different notation in the conceptual model to make the difference between an overlapping and a disjoint relationship, these two relationships are transformed using the same rule and the produced SQL code is the same. The disjoint constraint is not respected.

DB-main is able to transform and to merge partial disjoint relationship and produces the right relational model and SQL code.

4.3 Uncommon and practice based rules

4.3.1 Limitation due to the conceptual models

The original schema was modeled using ER-win. ER-win implements IDEF1x and according to the specification of this model:

> *A migrated attribute may be part of more than one foreign key provided that the attribute always has the same value for these foreign keys in any given instance of the entity. A role name may be assigned for this migrated attribute.*
> [IDE,]

DB-main does not allow to choose the name of the migrated attributes. Furthermore, the logical schema could be drawn directly in the relational model but this schema could not be retro-engineered to the ER model due to its special foreign keys. The conceptual model is not able to store our schema rightly due to lacks in the used ER model.

Rational Rose does not allow to choose the name of the migrated attributes. Furthermore, it does not allow to choose the attributes identifying an entity. This tool automatically adds a technical identifier. It was impossible to model this schema rightly.

4.3.2 Unstudied transformation rules

Designing a temporal database is only possible in DB-main. ER-win and Rational Rose totally ignore this concept. Furthermore, using ER-win, if the attribute Date is added by hand in the logical schema, the attribute is automatically added in the conceptual schema as an element of the primary key. This new attribute induces a contradiction between the primary key groups and the Is-a relationship.

Three temporal relationships are supported by DB-main: valid time, transaction time and both. In valid time, user has to insert the timestamps into each instance. In transaction time, user does not have to care about the time, the system fills the timestamps automatically.

The example is valid time. To implement a valid time entity, DB-main adds two attributes: starting and ending date. Between this two dates, nothing has changed in the instance. As we already explained, we need only one attribute to implement the time in our example, because of each instance represents a snapshot of the position of a Survey Point. A *snapshot* transformation is not supported by DB-main but could be added using the proprietary language Voyager 2. Nevertheless, this language is not a transformation oriented language, it is not its purpose. We decided not to explore this way.

4.3.3 Information lost

Rational Rose can not merge entities. Furthermore, it is impossible to define the transformation by hand. The test is canceled with this tool for these reasons.

ER-win can merge entities for sub/super type relationship, but can not merge entities linked by a 0-1 - 1-1 relationship. Using a sub/super type relationship instead of the 0-1 - 1-1, the transformation is made possible. We already showed, ER-win does not add any constraints when it merges sub type entities with its super type entity. In this example, this lack makes the transformation possible. But user of the tool is not aware that semantical information have been lost.

DB-main can merge entities linked by a Is-a relationship and a zero to one relationship. Nevertheless, the transformation is not possible because DB-main does not support to loose semantical informations. When we merge entities MainActivity and IndividualComment, the tool adds a coexistence constraint on attribute moved from MainActivity entity. Because of this constraint, it does not accept to merge the new entity IndividualComment with MiscComment and Comment. Indeed, this constraint would be lost. The transformation could be made by deleting by hand the constraint but no automation is possible.

4.4 Conclusion

Using our practice based examples, we were able to demonstrate some lacks in transformation rules implemented in studied CASE tools.

Firstly, lacks appear in the implementation of classical transformation rules. ER-win and Rational Rose drop systematically some constraints. ER-win does not implement rightly transformation rules even if these rules are well defined in the theory (zero to one relationship). Rose does some optimizations on the schema without asking anything to the user of the tool and without providing any way to disable them.

Secondly, uncommon transformation rules have a mixed result, depending on the used tool. Rose and DB-main were not able to solve any of these problems and ER-win was able to solve two of them. The biological example was already conceptualized using ER-win and the special constraints can be expressed with this tool due to its management of the foreign in the conceptual schema. The legacy example was solved because of the tool looses some semantical informations, but this loss was made without preventing the user.

Thirdly, ER-win and Rational Rose every classical transformation rule (merging entities linked by a Is-a relationship or a zero to one relationship).

In short, important differences of quality appears between studied tools. Rational Rose is the poorest tool for database design. It does not support important concepts as entity's identifier, entities' merging and disjoint Is-a relationship. ERwin is better but some lacks appear especially concerning the constraint added in SQL code to respect relationships' cardinalities. The best results were obtained with DB-main. Theory is well implemented, constraints are added in SQL code to respect cardinalities but this tool is not able to apply our uncommon rules.

Chapter 5

Using a novel rule approach for expressing transformation rules

5.1 Introduction

As we defined before, a CASE tool helps the developer to design its application. To achieve this goal, a tool provides models (UML, ERA...) and developer designs its application using these models. Models defining other models are called meta-model.

We can divide models in three levels: meta-metamodel, metamodel and model. Meta-metamodel is the higher level. It allows us to specify a metamodel. Metamodel is the second highest level. It is the model of the models. In this level, we define for example what is an entity type, a relationship, an attribute, etc. We define too the link between them. For example, an entity can have zero, one or more attributes. An attribute must belong to an entity. A relationship links an entity to another. Using these information, we can too define the structure to store models (the repository's structure). The lowest level is the model. It is a description of the user data.

The proof of the concept tool [Guvstavsson, 2003] is a meta-case tool (tool able to build its meta-model) based on a repository (formalized in UML) and an active rules system. The purpose of this project is to share conceptual schema with its transformation rules. To achieve this goal, the author defines a transformation based language, OCL+, to implement easily transformation rules. This language is an extension of the Object Constraint Language defined by the OMG group to describe constraints in UML. Schemes are exchanged using XML Metadata Interchange Language (XMI), a standardized language able to export metadata information. So, to export a schema (user model) from a tool to another one, we need to export the meta-model, the rules used to transform the schema and the schema itself.

5.2 The repository system

5.2.1 UML as Repository language

UML, for Unified Modeling Language, is considered now as a standard for modeling applications. UML offers several diagrams for separating concerns of different system views. The same conceptual framework and the same notation can be used from specification through design to implementation. Furthermore, UML is not a proprietary and closed language but is open and fully extensible. If we need something else that is not present in UML, we can easily change the UML specifications in order to add it.

In our work, we will use UML as meta-metamodel. We are particularly interested in the class diagram to build our repository. A class diagram is composed by three main components: class, binary association and generalization.

Class

A class is symbolized by a rectangle divided into three fields. The first field contains the class' name, the second field contains the class' properties and the third field contains definition of methods that are applicable in the class. Each property has a name and a type of data. A property can have an optional symbol representing its visibility (public, private or protected). Each method has parameters and a return data type. A method has a symbol representing its visibility too.

Binary association

A binary association is represented by a line linking two classes. A recursive binary association is an association where both end lines are the same class. At each end point, we find the role of the association and its multiplicity. On the line's center, we find the name of the association. The multiplicity can be * (zero or more), 1..* (one or more), 0..1 (zero or one) or 1 (one).

Generalization and specialization

Generalization is represented by a triangle connected to the supertype. Each subtype is linked to the triangle by a line. A subtype is by definition derived from the supertype. There are four kinds of generalization: overlapping, disjoint, complete or incomplete.

Using UML as meta-metamodel, we are able to build a metamodel and to use it in the proof of the concept tool. Metamodel could be a subset of UML or any other existing model. In our example, we will always use a subset of ER and relational models. We considerer UML is not complete enough to design database. The unexistance of identifier in entities is the main reason.

5.2.2 A basic repository example

Our repository illustrated by figure 5.1 is divided into two parts: on the top of the dotted line we have the conceptual part and on underneath we have the logical part.

The conceptual section is made up of three entities. The entity Attributes stores every information about attributes. The entity Entities stores every information about entities. An entity can have zero, one or more attribute(s). This relation is represented by the relationship AttrToEnt. The entity ERRelationship stores every information about relationships between entities. A relationship has a cardinality, this cardinality represent the maximal number of instances that can be referenced by another instance. The minimal cardinalities are always 0 in this example. A relationship links two entities, the parent entity is linked with the relation FromRel and the child entity is linked with the relation ToRel.

The logical section is made up of three entities. The entity column stores information about the columns of a tables. The entity Tables stores information about the tables. A table can have zero, one or more column(s), this relation is represented by the relationship In. The entity ForeignKeys stores information about table referencing another table. A foreign key links two tables, the referenced table uses the relation FromTable and the referencing table uses the relation ToTable and is compounded by one or more column(s) using the relation MemberOf.

Now, our repository is composed of the modeling of two simple models. We need to link these models by relationships. A table is the implementation of an entity, this relation is expressed by ImplementsTable. A column is the implementation of an attribute, this relation is expressed by ImplementsAttr and can be the implementation of a foreign key by the relation MemberOf defined above. Finally, a relationship is implemented by a foreign key, this relation is expressed by ImplementsRel.

5.3 Transformation rules

5.3.1 Introducing OCL and OCL+ as transformation rules

An UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.
[OMG, 2003]

OCL, Object Constraint Language, helps the developer to fill this gap. OCL can be used to specify invariants on classes and types in the class model, invariants for stereotypes, to describe pre and post-conditions on

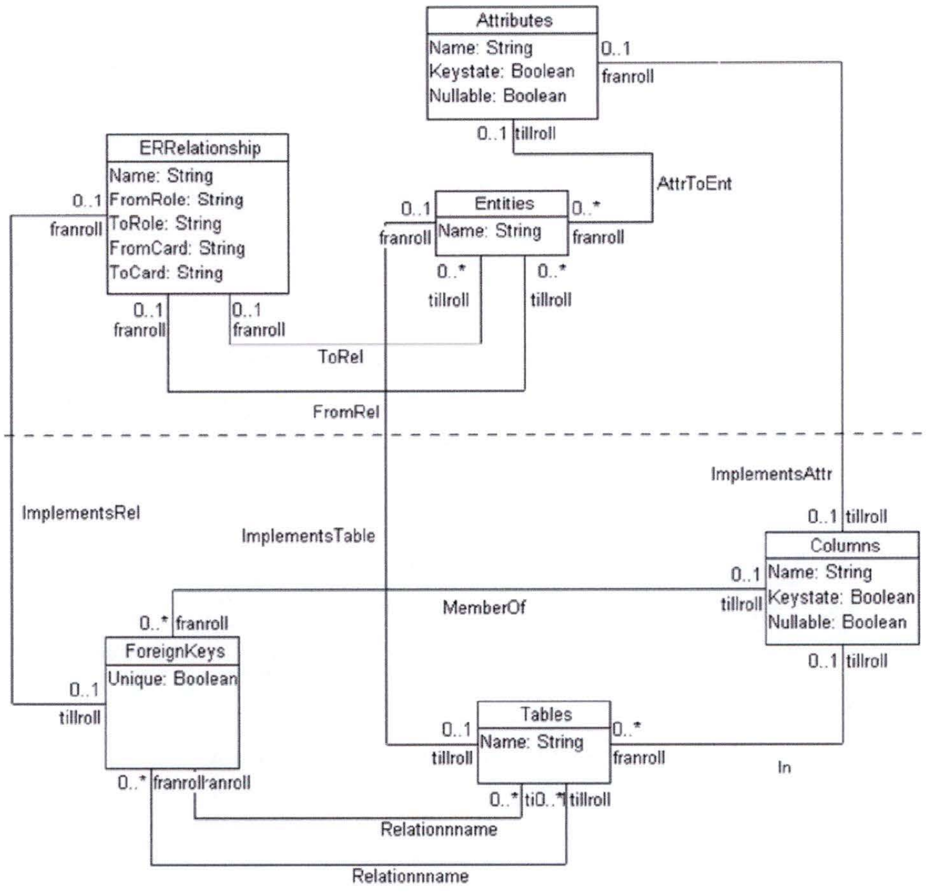


Figure 5.1: A simple repository

methods and metadata operations and to describe guards. An OCL rule illustrated by the table 5.1 is composed by three fields: the context class, the pre-condition and the post-condition. For all instances of the context class, if the pre-condition is respected, the post-condition must be respected too. OCL is a no side effect language. No action can change the system when a rules is applied.

context Typename::operationName(param1 : *Type1*, ...): *ReturnType*
pre parameterOk: param1 = ...
post resultOk: result = ...

Table 5.1: OCL quick specifications

Type of data

OCL is a strongly typed language. It supports basic type data as integers, floats, booleans and strings. Operations on each type are summarized in the table 5.2.

Type	Values	Operations
Integer	1, -2, 504, ...	*, +, -, /, abs()
Float	1.2, 2.5, 56.89, ...	*, +, -, /, floor()
Boolean	true, false	and, or, xor, not, implies, if-then-else
String	'This is a string'	toUpper(), concat()

Table 5.2: OCL types, values and operations

Operations

Pre and post conditions allow us to make some basic operations such as the addition on integers and floats, boolean operations and some modifications on strings. Theses operations are allowed to check the value of each variable, but can not change the system. There is no possibility to declare new variables or to keep a value between two rules for example. Purpose of an OCL rules is checking if all variables respect the precondition then the produced result has to respect the post condition.

Extending OCL

OCL is an interesting language to express transformation rules. The system of pre and post conditions with a context class defines clearly on which kind of instance the rules have to be applied. But operations in OCL have no side effect and we need, to apply a transformation, to be able to change the data stored in the repository. Furthermore, we don't want to apply rules on a function in a class but on the class itself.

To avoid this problem, some modifications are made. The post condition, now known as *Action*, accept two new instructions: the assignment (:=) and the separator of instructions (;). A new field appears, *Declaration*, to be able to create new values and to declare variables.

Some other modifications are made too to complete the system. We keep the *Context class* and the *Precondition*, now known as *Condition*. *Context class* refers now to an entity in the repository system (in place of a function signature).

With the field *Event*, we are able to define in which case the rule have to be applied when the condition are respected. There is three kinds of event: insert, update and delete. A rules where the field *Event* is defined as Insert will be used when a new tuple is added in the entity defined by the field *Context Class* and if and only if every statement of the *Condition* field is met. A rule must be declared for one and only one event.

To complete the event, three categories are added: normal, internal and collection. A normal event is initiated when a tuple is inserted, updated or deleted by the user. An internal event is an event initiated by another event. A collection event is initiated when an action modifies a collection of data.

An OCL+ rule is defined like this:

Context Class	Name of the entity
Event	type of event
Condition	condition1 [and/or] condition2 [and/or] ..
Declaration	type: variable
Action	action1;action2;..;actionN

The next example presents a rule which automatically add the string "Ent_" to the name of the entity and add an identifier "ID" to the entity.

Context Class	Entity
Event	Insert
Condition	
Declaration	Attributes A

Action	A.Create;
	A.Name:=ID;
	A.Keystate:=true;
	A.Nullable:=false;
	self.name := concat("Ent_", self.name);
	self.AttrToEnt := A

5.3.2 The active repository system

With a repository and a set of rules, we need a system to apply the rules on the components we add in the repository. To reach this goal, the author has decided on an active database systems.

> *Active database systems allow users to create rules specify data manipulation operation to be executed automatically whenever certain events occur or condition are met.*
> [Widom, 1996]

Using this idea, when an user adds an information in the repository, if this information respects the conditions of one rule defined for the used context class, the action of the rule is initiated. This action can initiate some other rules (internal or collection) and these rules can initiate other rules too, in every context class.

The proof of the concept tool admits multiple events, that means the user adds all its information in one time and commit all changes. The system first applies all 'Normal' rules and after applies all internal or collection rules. The order is the order of the rules. If two rules could be applied, the system applied the first one (i.e. the first in the list).

In order to reduce the scope of this work, we just study the event 'Insert'. Actually, 'Update' and 'Delete' events are complex, due to the fact they should change all the database schema. For example, if we allow an user to change the cardinalities of a relationship, this change could induce the deleting of a table (many to many relationship updated to a one to many relationship), could change the primary key of number of table (Identifying relationship to a non identifying relationship), etc. This kind of problem is out of scope and should be the subject of another research.

5.3.3 A basic example of OCL+ rules using the basic repository

Scope of the example

The basic repository supports only entities, attributes and relationships between two entities in the ER part and only tables, foreign keys and columns for the relational part. We need two transformation rules: the first one transforms a zero to many relationship with a foreign key pointing the primary key of the other table and the second one transforms a zero to one relationship, adding a unique foreign key in the child table.

Building the rules

The purpose of a rule is not to replace the developer in the identification of the needs but to help him to automatize the transformation of huge schemes from one model to another. This distinction is important because of the developer has to enter enough information when he builds his conceptual schema to allow the tool to choose the right rule. In the other way, each rule must have a unique condition to be applicable.

Due to these two conditions, we demonstrate the importance of the link between the rules and the repository. The repository has to be complete enough to allow the rules' developer to write unambiguous rules. It must provide attributes to write the condition statement without ambiguity and attributes to execute the action statement without asking to the tool's user any unknown information when the transformation is made.

Firstly, we need a rule to transform an entity to a corresponding table. This transformation is the easiest, the action statement needed is divided in two parts: we create first a new table using the instruction new, secondly we copy all common attributes from the entity to the table (name...).

Context Class	Entities
Event	Insert
Condition	ImplementsTable→isempty
Declaration	Tables Tab
Action	Tab.create;
	Tab.Name := self.Name;
	self.ImplementsTable := Tab;

Secondly, we need a rule to transform each attribute of an entity to a column in the corresponding table. In this rule, we take care the fact an attribute must belong to an entity (see the condition statement).

Context Class	Attributes
Event	Insert
Condition	ImplementsAttr→isempty and AttrToEnt→notempty
Declaration	Columns Col
Action	Col.create;
	Col.Name := self.Name;
	Col.Keytate := self.Keystate;
	Col.Nullable := self.Nullable;
	self.ImplementsAttr := Col;

Thirdly, for each relationship, we build the transformation rule. To make the difference between each kind of relationships, we check the cardinalities in the condition statement. In these rules, we need to create a new instance of ForeignKeys. We need to add in the table implementing the child entity a column for each attribute member of the identifier of the parent entity (i.e. attribute with Keystate = true).

Context Class	ERRelationships
Event	Insert
Condition	ImplementsRel→isempty and FromCard="N" and ToCard="1"
Declaration	ForeignKeys FK, Columns Col
Action	FK.create;
	FK.Unique:=false;
	self.ImplementsRel:=FK;
	self.FromRel.AttrToEnt→reject(Keystate=false)→iterate(
	PK1|
	Col.Create;
	Col.Keystate := false;
	Col.Name := PK1.Name;
	Col.In:=self.ToEntity.ImplementsTable;
	Col.MemberOf:=FK)

Context Class	ERRelationships
Event	Insert
Condition	ImplementsRel→isempty and FromCard="1" and ToCard="1"
Declaration	ForeignKeys FK, Columns Col
Action	FK.create;
	FK.Unique:=true;
	self.ImplementsRel:=FK;
	self.FromRel.AttrToEnt→reject(Keystate=false)→iterate(
	PK1|
	Col.Create;
	Col.Keystate := false;
	Col.Name := PK1.Name;
	Col.In:=self.ToRel.ImplementsTable;
	Col.MemberOf:=FK)

Three remarks have to be made about the OCL+ language and the used tool. First remark, loop, the statement able to examine a collection of instance, is prefixed. That means the condition and the list of instance to be examined is evaluated and created before the entrance of the loop. During the analyze of each instance of a collection, if we add an instance in the collection and if this instance respects the conditions of the loop, this new element will not be analyzed. In our example, we need to analyze every instance referenced by childAttribute of an entity. The condition of the loop is the rejection of every attribute with the Keystate set to false. The first word following *iterate(* is the variable pointing to the current element to analyze.

Second remark, the current version of the tool does not allow to use the conditional instruction (*if-then-else*) in the action statement. For the moment, the only way to do a condition in the action part is dividing the rules in two new rules, using a different condition statement. This solution is not complete enough, and some problems are unsolvable. We analyze this problem in the next chapter. This is a known problem and it will be corrected in the next version of the tool. This problem is not a bug, but the author of the tool considered this instruction as optional.

Third remark, the current version of the tool does not allow to merge two strings. This is a bug and it will be corrected in the next version.

5.3.4 An alternative: Action Semantic

Purpose of Action Semantic

Action Semantic is a language added in UML by the OMG in 2000. The purpose of this language is to fill the gap between high level concept of UML and the low level programming constructs found in the used oriented language. One of the main lacks in UML is the absence of formal and precise foundation for several constructs such as transition guards or method bodies. These lacks cause the impossibility to simulate and validate an architecture.

Action Semantics (AS) was defined by the OMG to specify algorithms in high level. Before AS was included in UML, the only way to specify the behavior of a function was in an uninterpreted string. This solution is problematic because of developers could misinterpreted the string. Furthermore, this string does not help to automate a formal proof of correctness of a problem specification, does not make possible high-fidelity model-based simulation and verification, does not help for the reusability of a component without reading the whole low-level code.

Such precise action specifications, in conjunction with the UML, provide a stronger basis for model design and eventual coding and could support code generation to multiple software platforms. Action Semantics is a formal language, platform independent, strongly typed, able to specify any functionality of a software.

Relying on the fact that UML meta model is itself a UML model, authors [G. Sunye and Jezequel, 2002] show how the AS can be used at the meta model level to help the OO designer carry on activities such as behavior-preserving transformations, design pattern application and design level aspects weaving. This approach of AS is particularly interesting for us for two reasons: the repository used to build OCL+ rules uses UML as meta model and AS can be combined with OCL to verify if a transformation may be applied, as the condition statement of OCL+. Furthermore, the authors distinguish the same two steps in design level activities: identification of the need to apply a given transformation on a UML model in actual transformation of that model, without forgetting the fact the purpose is not to replace the developer in the first step but to automatize the second step.

Rules' example

A quick example helps to compare OCL+ and AS. The purpose of this rule transforms an entity into a table. In OCL+, such a rule is defined like this:

Context Class	Entity
Event	Insert
Condition	ImplementsTable→isempty
Declaration	RelTable RT

Action	RT.create;
	RT.name:=self.name;
	self.ImplementsTable:=RT

In Action Semantic, there is no declaration field, new instances are created directly in the action code. The same rule can be expressed like this:

Class::Entity

Pre:

class.ImplementsTable→isEmpty()

Action:

newTable := RelTable.new

newTable.name:=self.name;

newTable.addAssociationTo(self, 1, 1)

Post:

class.ImplementsTable→notEmpty()

Comparison to OCL+

OLC+ and AS with OCL are very similar, defined in UML and able to realize the same kind of program. Furthermore, their syntax is nearly the same.

But there are two big differences between them. Action Semantic does not define a field *Event*. In our system of active rules, this is really problematic. Without event, we need to reapply all transformations to transform the schema into the other model. Cascading the rules became impossible (we can not make the difference between an internal event and a normal event).

OCL+ replaces the post condition field with an action field. This action field is able to modify the repository. By replacing this post condition, OCL+ prevents the use a tool to check the final result. We can imagine a system where action is realize and post condition is checked to verify if the result of the transformation is correct. Such a system will help to debug huge number of rules easily.

Purpose of these languages is similar but the absence of the *Event* field in AS is too important to be used in the repository system describe above. Furthermore, the absence of post condition field in OCL+ is not important, it just helps the developer to create correct rules and to verify the behavior of these.

Chapter 6

Solving practice base case using OCL+

6.1 Used repository

6.1.1 Introduction

As we explained in the previous chapter, the purpose of a repository is to store every information about a schema and to be able to express every needed constraint. To this purpose, we add a second one: the repository has to store every information to transform a schema from a model into another model without requiring new information. This second purpose leads to add extra information not present in the original models.

To be able to transform a schema, the tool presented in the previous chapter uses a system of linked repositories modeled in UML. The repository models two or more models, each model linked to each other. These links are made using relationships between repository's entities and with transformation rules coded with a transformations oriented language (OCL+).

We divided transformation rules into two different classes: classical transformation rules and uncommon transformation rules. Classical transformations are well defined in the theory and are symmetrically reversible. We criticized CASE tools which are not able to transform rightly relationships using these classical transformation rules. In order to implement these rules, our work was based on [Hainaut, 2002] and [J-L. Hainaut and Roland, 1996].

About uncommon transformation rules, we implement the transformation as presented in the selected papers.

The purpose of this work was not to make a perfect repository able to store every kind of constructs and every possible transformation but to have a repository complete enough to be able to store and to transform our practice based schemes and furthermore to implement all classical transformation rules presented in the previous chapters. Building our repository as and when we need it, we can study the incidence of the completeness of a repository on transformation rules.

This chapter sums up the implementation of the different rules used to solve our practice based schemes. Firstly, we describe the first version of the used repository. This repository was made by Gustavsson in [Guvstavsson, 2003] to solve the real life example of his thesis. This version is illustrated in annexe C. Secondly, we sum up, for each class of transformations, the implemented transformation rules. They are listed in annexe C.

6.1.2 Repository's description

Figure 6.1 represents the first version of the repository. The upper part is the conceptual repository. It is a simplified version of ER model. The lower part is the logical part. It is a modified version of the classical relational model.

Entity Association Repository

The entity EREntity stores every information about entity type, that is its name and if a table has to be created to implement it. The *notable* attribute is used in case of merging of entities. This indicates that no relational table is to be generated for the merged entity.

An entity could have zero, one or more attributes and an attribute is part of zero or one entity. Therefore, EREntity is linked by a *0-1 - 0-** relationship to the entity ERAttribute. This is a choice made by Gustavsson in his first repository. From our point of view, it does not bring any problems, an attribute without entity is present in the repository but has no side effect.

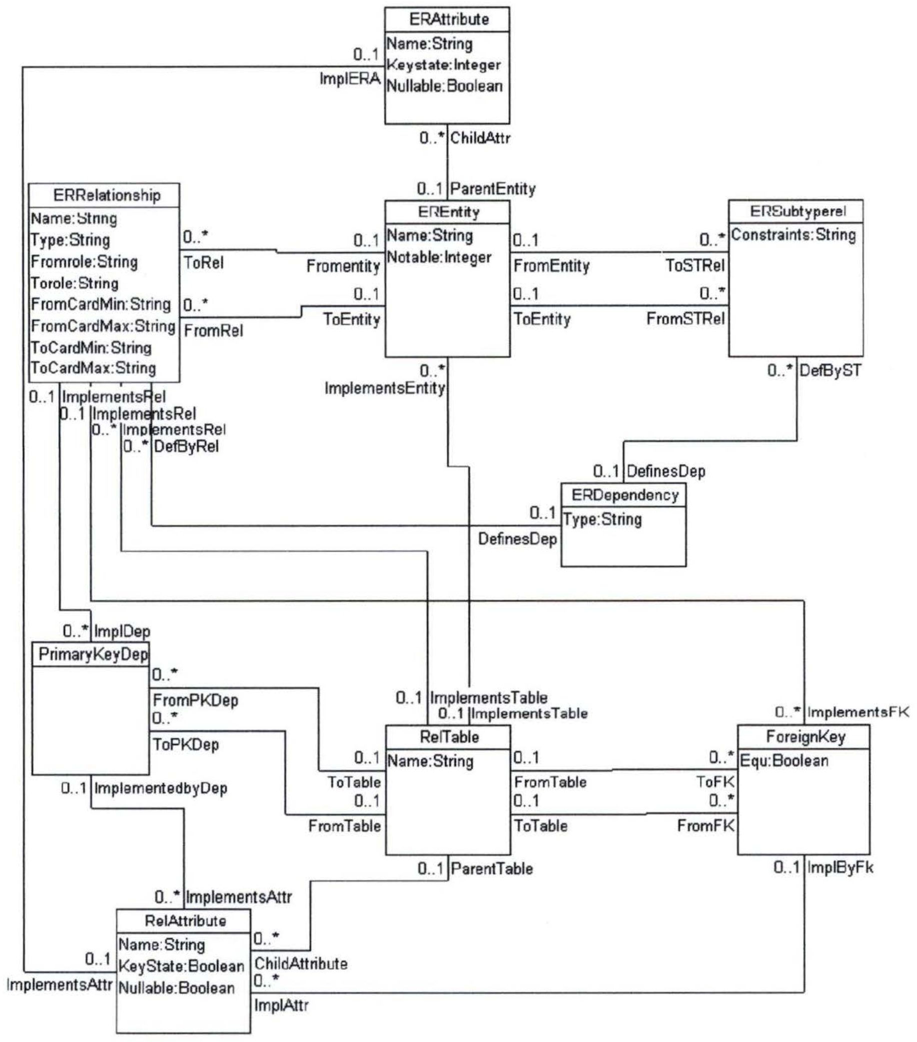


Figure 6.1: Repository version 1

An instance of ERAttribute has three characteristics: *Name*, *Keystate* and *Nullable*. *Keystate* means the instance is an attribute part of the identifier of its entity. *Nullable* set to true means the value of the attribute is optional.

A relationship links two and only two entities (the entities could be the same instance of EREntity) and an entity can be linked to zero, one or many other entities. A relationship links a child entity *to* its parent entity. EREntity is linked by two *0-1 - 0-** relationships to the entity ERRelationship. The parent entity is linked to its child by the relation FromEntity and the child entity is linked to its parent by the relation ToEntity. This notation helps us to distinguish between two parts of a non-symmetrical relationship. In case of a symmetrical relationship like a many to many relationship, this distinction has of course no sens. A relationship is described by eight attributes. *Name* stores the name of the relationship. *Type* stores the type of relationship. We make the choice to use this attribute to distinguish between identifying and non identifying relationships. *Fromrole* and *Torole* give a name for each role played by the relationship. *FromCardMin* and *ToCardMin* store the minimal cardinalities of each side of the relationship and *FromCardMax* and *ToCardMax* store the maximal cardinalities of each side of the relationship. *FromCard* indicates the number of entities which could reference the parent entity and the *ToCard* indicates the number of parent entity which a child could reference.

ERSubtyperel stores a sub-type relationship between a child (FromEntity) and a parent (ToEntity). An ERSubtyperel relationship is only between two entities. A super-type could have one or more children. To store this information, we use ERDependency. It puts together ERSubtyperel instances and allows to add constraints between these relations (dijunction etc.). A sub-type relationship can be implemented using a zero to one relationship. The relation DefineDep links the relationships implementing a sub-type relation with the sub-type group in the ER model. A sub-type group is implemented using zero, one or more relationships.

The original repository has two other relations *0-1 - 0-** linking ERDependency and EREntity. From our point of view, these relations were useless and redundant with the relation linking ERSubtyperel and EREntity. We choose to delete these two relationships in order to simplify the repository and the rules.

Relational model Repository

RelTable stores information about table. A table can implement zero, one or more entities or can implement a relationship (transformation of a many to many relationship, for example). A table is characterized by a name. The relation between RelTable and EREntity is a *0-1 - 0-** one in the repository of Gustavsson. This relation allows a table to implement many entities and is used when two entities are merged into an unique table. In theory, an entity could be spitted into many tables. Nevertheless, we do not meet this transformation in our practice based schemes. We keep the relation *0-1 - 0-** in order to keep the repository as simple as possible.

A table has zero, one or more attributes, stored in RelAttribute. An attribute has a *name*, a *keystate* (the attribute is member of the primary key group) and can be *nullable* (for an instance of the table X, this attribute can have the value *null*). A RelAttribute is an implementation of an ER attribute or an implementation of a relationship (attribute part of a foreign key).

ForeignKey puts together attributes into a foreign key group, to implement a relationship. If the boolean *Equ* is set to true, each instance of the parent entity has to be referenced by an instance of the child entity.

PrimaryKeyDep puts together the attributes member of a foreign key implementing an identifying relationship. This entity was originally present in the first repository. In order to access to the information as quickly as possible, Gustavsson [Guvstavson, 2003] recommends to build a redundant repository. In a relational schema, the difference between the implementation of an identifying relationship and a non-identifying relationship is the fact that all migrated attributes are members of the primary key group of the child table. In the repository, that means these attributes have the boolean *keystate* set to true. To make the difference between the implementation of an identifying and a non identifying relationship, we have to check the *keystate* of each attribute member of a foreign key. In order to save us from this heavy action, the entity type PrimaryKeyDep was added.

Another usage of PrimaryKeyDep is to puts together the attribute member of the primary key group that are created to implement a kind relationship but that are not in a foreign key group. This special attribute will be studied in the section about unstudied transformation rules.

6.1.3 Lacks

We can already point at some lacks in the repository. It can not represent ternary relationships. A relationship links two and only two entities. This choice was made by Gustavsson to simplify the model. Modeling a repository able to store this kind of construct is possible but no practice based schemes analyzed uses ternary relationships. Furthermore, rules based on a repository able to store ternary (and more) relationships are more complicated and impossible to implement without the conditional instruction. Indeed, with only binary relationships we bypass the absence of the conditional instruction by building a rule for each variation of the cardinalities. With ternary relationship, the number of rules becomes too important to be done.

The ER model does not support attributes and identifiers in a relationship. Even if such a relationship appears in the bus example, we decided not to care about this notation. Rules able to transform such a relationship are easy to implement but without the conditional instruction, we need to divide all rules into two new different variations (with and without attributes).

An attribute in the relational model could take part of one and only one foreign key group. We selected the biological example because of an attribute takes part of many foreign key groups. We will avoid this lacks in the next sections and studying the incidence of the change on the already made rules.

The relational model is not able to add constraints between the attributes and between groups of attributes. This lack will be avoided in the next sections.

6.2 Classical transformation rules

6.2.1 Building the rules

If-then-else problem

As we explain in the previous chapter, the proof of the concept tool does not support the *if-then-else* instruction in the action statement. Due to this fact, we have to build a new rule for each small modification of the cardinalities and for each type of relationships.

The non existence of the *if-then-else* instruction causes another unexpected problem: the debugging of rules is more complicated. Rules are really similar and doing a *copy and past* to create every small variation is probably the best way to create an homogeneous set of rules. But if the starting rule has a mistake, this mistake will be copied on all rules and debugging will be multiplied by the number of variations made from the first rule.

Entities and attributes

Before creating rules to transform relationships, we need some basic rules to transform entities and attributes to the corresponding constructs in the relational model.

A table has to be created if the relation ImplementsTable between ER-Entity and RelTable does not exist. That means this entity is not already implemented. Furthermore, a table must be created if the value of the attribute *notable* is false. The *notable* attribute is used in case of merging of entities. This indicates that no relational table is to be generated for the merged entity. If these two conditions are respected, a table is created and linked to the implemented entity.

```
Declaration:
RelTable RT
Action:
RT.create;
RT.name:=self.name;
self.ImplementsTable:=RT
```

The rule implementing an attribute is similar. We have to check if the relation ImplementsAttr does not exist and if the attribute *notable* of the parent's entity is set to false. Indeed, the *notable* is set to true, this entity will be merged in another. We will create each instance of RelAttribute for each attribute member of this kind of table in the rules that effectively merge the entities. If the condition is respected, an inscance of RelAttribute is created with the same characteristics than the instance of ERAttribute and is linked to the implemented attribute.

Declaration:
RelAttribute RA
Action:
RA.Create; (Instantiation of the RelAttribute)
RA.ImplERA:=self;
RA.Name:=self.Name;
RA.KeyState:=self.KeyState; *(all common characteristic are copied from one model to another)*
RA.ParentTable:=self.ParentEntity.ImplementsTable *(link between ERAttribute and RelAttribute is made)*

6.2.2 Zero to many relationship

Identifying relationship

We choose to use the attribute *type* of ERRelationship to make the difference between a non-identifying and an identifying relationship. We choose the string "I" for an identifying relationship. The condition statement checks the cardinalities of the relationship (both minimal and maximal cardinalities) and the value of the attribute *type*.

The identifying relationship needs to migrate every attribute part of the parent's entity identifier to the table implementing the child entity. A new instance of ForeignKey is created to put together all migrated attributes. This is the foreign key group. As we explain in the previous section, in order to make the difference between a foreign key group implementing an identifying relationship and one implementing a non-identifying relationship, an instance of PrimaryKeyDep is created. This instance puts together the same attributes than the instance of ForeignKey. The instances of PrimaryKey-Dep and ForeignKey are linked to the implemented relationships using the relations ImplementsRel. Finally, all migrated attributes have their *keystate* attribute set to true because they are members of the primary key group and their attribute *nullable* is set to false because the relationship is mandatory.

```
Action:
FK.Create;
FK.Equ:=false;
FK.ImplementsRel:=self; (We link the instance to the imple-
mented relationship)
FK.FromTable:=self.FromEntity.ImplementsTable;
FK.ToTable:=self.ToEntity.ImplementsTable;
Pk.Create;
nal model. This entity stores, for each relatio PK.ToTable:=self.ToEntity.ImplementsTable;
Pk.ImplementsRel:=self; (We link the instance to the imple-
mented relationship)
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1| for each attribute member of the primary key group
of the parent entity, we copy these attributes in the table im-
plementing the child entity. PK1 refer to the current attribute
member of the identifier of the parent entity
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Re.ImplByFk:=FK)
```

Non-identifying relationship

The transformation is nearly the same than the identifying relationship but the value of the attribute *type* is the string "N" in order to make the difference with the identifying relationship.

An instance of ForeignKey is created to put together all migrated attributes but no instance of PrimaryKeyDep is created, this relationship being not part of the identifier of the child entity. The *keystate* of each migrated attribute is set to false. The attribute *nullable* is set to true because the relationship is not mandatory for the child entity.

Variations

An instance of the child entity could be linked to one and only one instance of the parent entity with a non-identifying relationship. In this case, the boolean *nullable* is set to false for each attribute part of the foreign key implementing the relationship.

A zero to many relationship (identifying or non-identifying) can be mandatory for the parent entity. Each instance of the parent entity must be refereced by one or more instances of the child entity. In this case, the boolean *Equ* of the instance of ForeignKey is set to true.

Briefly, a zero to many relationship is implemented by six different rules, depending of their minimal cardinalities. Two of these rules implement identifying relationship. Indeed, an identifying relationship is always mandatory for the child entity but could be optional or mandatory for the parent entity. The four other rules implement the non-identifying relationships.

6.2.3 Zero to one relationship

The zero to one relationship rules are similar to the zero to many rules, but the value of the union of the attributes part of the foreign key must be unique. In order to guarantee this constraint, the repository needs to be able to store constraints between attributes in the relational model. The first version of the repository is not able to do this.

The first idea to implement this constraint was the addition of a new attribute *unique* in the entity type ForeignKey. A foreign key group with the attribute *unique* set to true means the union of the attributes members of this group must be unique. Nevertheless, the implementation of the different kinds of Is-A relationships will require other constraints between attributes and between groups of attributes (unique, coexistence, exactly one, exclusion, etc.). Furthermore, these attributes are not always members of a foreign key group (case of merging for example). The *unique* attribute is not general enough to be used.

In order to solve this problem, an entity AttrConstraint is added in the repository. This entity is linked to RelAttribute by a relation many to many. An attribute can be member of different constraints and a constraint can contain zero, one or more attributes. Furthermore, we add a recursive zero to many relationship to be able to add constraint between group of constraints. For example, an entity A has four optional attributes. These attributes, we are under the next constraints:
($A1$ *IS NOT NULL AND* $A2$ *IS NOT NULL*) *XOR*
($A3$ *IS NOT NULL AND* $A4$ *IS NOT NULL*)
To implement these constraints, three instances of AttrConstraint are created. The two first instances guarantee the coexistence constraints between $A1$ and $A2$ and between $A3$ and $A4$. The third constraint adds the exclusion between the two first groups using the recursive relationship.

An instance of AttrConstraint could be created to implemente a relationship or a constraint of an Is-a relation. This entity is linked to ERRelationship and to ERDependency in order to keep this information.

The other transformation found in our practice based example consists in implementing the zero to one relationship by merging the child entities with its parent entity. All attributes of the child entity are added in the table implementing the parent entity. All mandatory attributes of the child entity become optional and are put together in a coexistence group using the entity AttrConstraint. The optional attributes are simply added into the parent table without any other constraints.

In case of a zero to one non-identifying relationship, the child entity could have an identifier. In this case, a unique constraint must be added to put together the attributes members of the identifier.

Finally, the parent table implements two different entities, this relation is stored with the relation ImplementsTable.

One more time, the absence of the *if-then-else* instruction adds a new problem. An unique constraint must be added if the child entity has an identifier and a coexistence constraint must be added to put together the mandatory attributes of the child entity. We decide to systematically create these two instances of AttrConstraint even if there is no attribute referenced by these constraints. It simply adds some useless data in the repository and building a rule able to clean the repository is not complicated.

6.2.4 Many to many relationship

To implement a many to many relationship, we need to create a new table. This table will be the component implementing the relationship. It is not linked to an entity as other table but to the implemented relationship using the relation ImplementsTable. The identifier of this table is compound by all attributes members of the identifier of both linked entities. Two instances of PrimaryKeyDep and of ForeignKey are created to put together the migrated attributes.

As in the variation of zero to many relationships, attribute *Equ* of the entity ForeignKey has to be set to the right value, corresponding to the type of minimal cardinality for each side of the relationship.

We explained in the previous section that a relationship is oriented from the child entity to the parent entity. This orientation has no sens for a symmetrical relationship such as the many to many. This relation can be mandatory for one entity or for the other (one minimal cardinality is set to 1). We need to build two different rules to implement *0-N - 1-N*, one for each orientation, depending which side of the relationship has the minimal cardinality set to 1.

6.2.5 Is-a relationship

Merging the entities or implementing by foreign keys

All Is-a relationships are stored in the repository using the same entities. ERSubtyperel stores the relation between the parent and the child entities and all relations are put together using an instance of ERDependency. ERDependency defines the type of Is-a relationship.

The first method presented in our practice based schemes to implement an Is-a relationship consists in implementing each relation by a foreign key, as the zero to one relationship. An instance of ERRelationship is created for each ERSubtyperel referenced by the instance of ERDependency. This ERRelationship is a zero to one relationship. The boolean *NoTable* of each child entity is set to false. Each child references the parent entity using a foreign key and all attributes implementing the foreign key are put together under a constraint of uniqueness by an instance of AttrConstraint.

The second method presented in our practice based schemes to implement an Is-a relationship consists in merging the children entities with the parent entity. The boolean *NoTable* of each child entity is set to true. The table implementing the parent entity is linked to all child entities. All attributes of each child become optional and are put together under a constraint of coexistence using entity AttrConstraint defined above.

In both implementations, depending on the constraints between the children, some variations appear.

Overlapped

An overlapped relationship does not require any other constraints between the children, whatever the method to implement this relationship (merging or using foreign keys).

Disjoint

A disjoint relationship requires a constraint between the children, to respect the disjunctive constraint. Using foreign keys to implement this relationship, we add an attribute for each child entities in the parent table. All these attributes are optional and under an exclusive constraint. The instance of AttrConstraint implementing this constraint references the instance of ERDependency.

Merging the entities to implement this relationship requires a exclusive constraint between each coexistence group of attributes. A new instance of AttrConstraint is created and references each constraint of coexistence.

6.3 Practice based rules

6.3.1 Limitation due to the conceptual models

All problems we met until now with OCL+ were produced because of lacks in the repository or because of the absence of the *if-then-else* instruction. Our repository is a light version of the ER model and is intentionally not able to store every type of constraints and every kind of relationships normally allowed by the ER model. But limitations are not always due to our repository but to 'lacks' in the used model itself.

The model used to model the original schema (IDEF1x [IDE,]) allows an attribute to be member of many foreign keys.

To be able to choose the future name of the attribute implementing the foreign key, we need a new entity, an hybrid between the ER model and the relational model. This entity stores, for each relationship, the name of the migrated attribute for each attribute member of the identifier. If an attribute with the same name already exists in the table, no new attribute is added and this existing attribute is added in the foreign key group implementing the current relationship.

The first idea to solve this problem was to read the information stored by AttributRelation. But without the *if-then-else* instruction, it is impossible to check if an attribute with the same name already exists in the table.

The second idea was the addition of a new rule with AttributRelation as context class. When an information is stored in this entity, the rule creates automatically an instance of RelAttribute if no attribute has the same name in the table implementing the child entity. The rules which transform effectively the relationships are changed. They create the instance of ForeignKey and do not create the instance of RelAttribute but make the union between the foreign key group and the previously created attributes.

Condition:
self.UseAttr.ToEntity.ImplementsTable.ChildAttribute→forall(RA|
RA.name <> self.Name)
Action:
RA.Create;

```
RA.ParentTable:=self.UseAttr.ToEntity.ImplementsTable;
RA.ImplERAFK := self;
RA.Name := self.Name;
RA.KeyState := false;
RA.Nullable := true;
```

Another lack in the repository appears during the implementation of these rules. In this example, a relational attribute could be part of zero, one or more foreign key groups. In order to be able to store this information, we have to change the relation bewteen RelAttribute and ForeignKey from a zero to many relationship to a many to many one. This change requires to modify every already implemented rules. Indeed, in previous rule, we linked an instance of RelAttribute and one of ForeignKey using this instruction *Re.ImplByFk:=FK;*. This could not be used anymore due to the used many to many relationship. Both sides of the relationship refers to a collection of objects, we need to use the instruction *union* to add the attributes in the foreign key group.

A third repository is created. This repository is not compatible with the two first repositories used until now.

6.3.2 Unstudied transformation rules

This transformation is an update of the overlapped partial Is-a relationship, studied in the previous section. The difference is we add a new attribute to store the date in all child tables. This attribute is member of the primary key group, has no corresponding attribute in the ER model is not member of the foreign key group implementing the relationships between the parent entity and its children and is added to implement the relationship. In order to keep this information, this attribute is added in the instance of PrimaryKeyDep. It is the only example of an attribute being member of a PrimaryKeyDep without being member of a ForeignKey group.

We simply decide to use the attribute *type* to make the difference between a normal Is-a relationship or a temporal Is-a relationship. The transformation is exactly the same than the overlapping relationship.

6.3.3 Information lost

The purpose of this transformation is to merge entities Comment, IndividualComment, MiscComment and MainActivity. As we explain before, to merge entities IndividualComment and MainActivity, we need to create an optional group of attributes in IndividualComment. But to merge the new entity with its parent, we have to create an optional group and in this group, there is another optional group. This transformation is impossible.

The used transformation does not care about optional groups. Every attribute moved from the merged entities is optional, without any constraints between them. The created rule moves all attributes, by cascading, to the main parent entity. These rules consist in a lighter version of the merging entity in Is-a and in zero to one relationships without implementing any constraints. The tool cascades automatically all transformations. All rules able to solve this example can be found in [Guvstavsson, 2003].

6.4 Conclusion

We demonstrate in this chapter OCL+ gives to the developer a powerful language to build his own transformations for database engineering. We demonstrate too the importance of the completeness of the repository. The second version of the repository, illustrated by figure 6.2, and the differences between this repository and the first version highlight clearly this need of completeness. We show too with the third version 6.3 of the repository that a repository could be totally incompatible with rules even if the change is minimal. We demonstrate finally the importance of hybrid information in a repository to be able to store every needed information to apply completely a transformation.

We demonstrate the proof of the concept tool works but we demonstrate too the importance of the *if-then-else* instruction in the action statement. This instruction was not implemented following an old recommendation in the OCL description from the OMG (this note does not exist anymore). Some rules were impossible to implement due to the absence of this instruction.

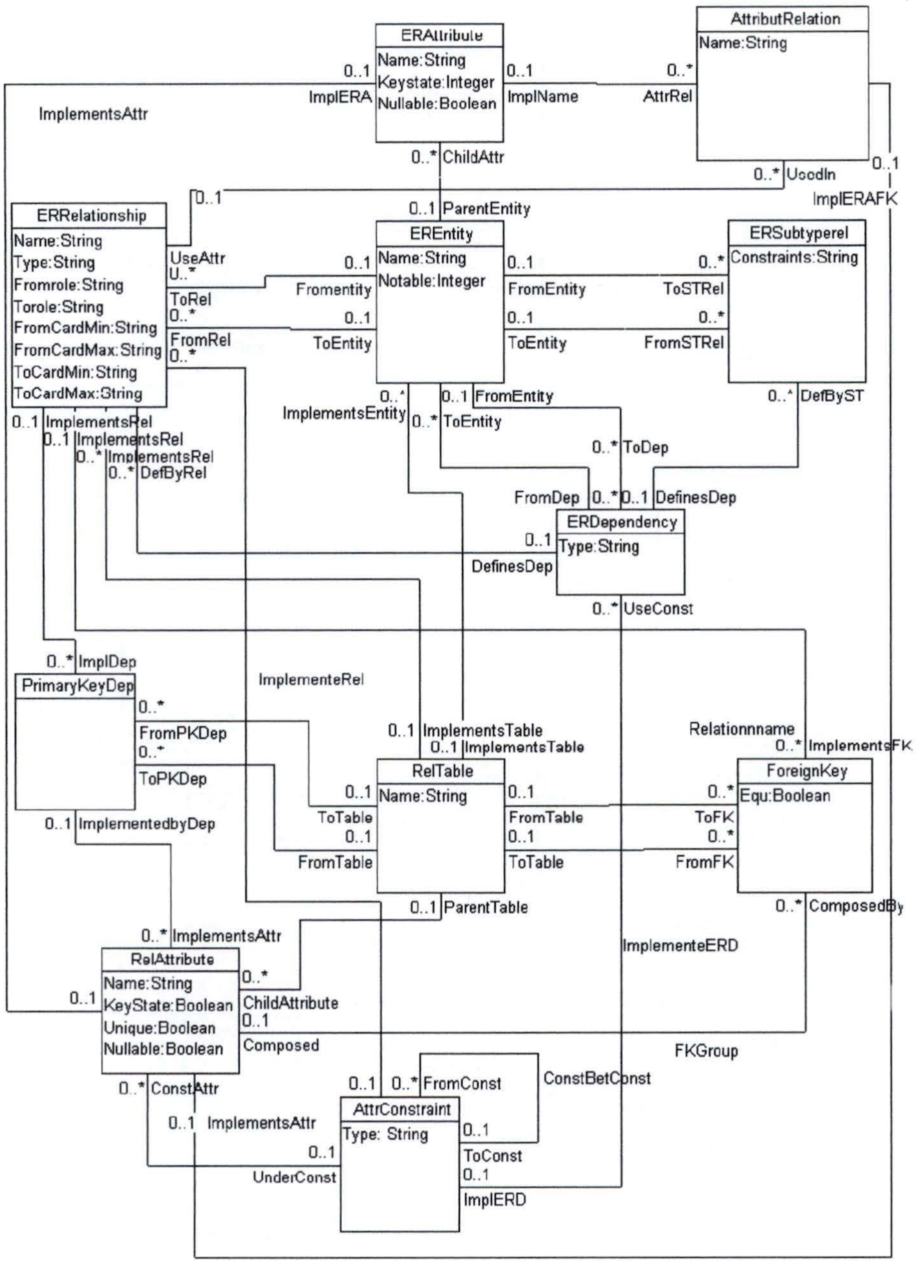


Figure 6.2: Repository version 2

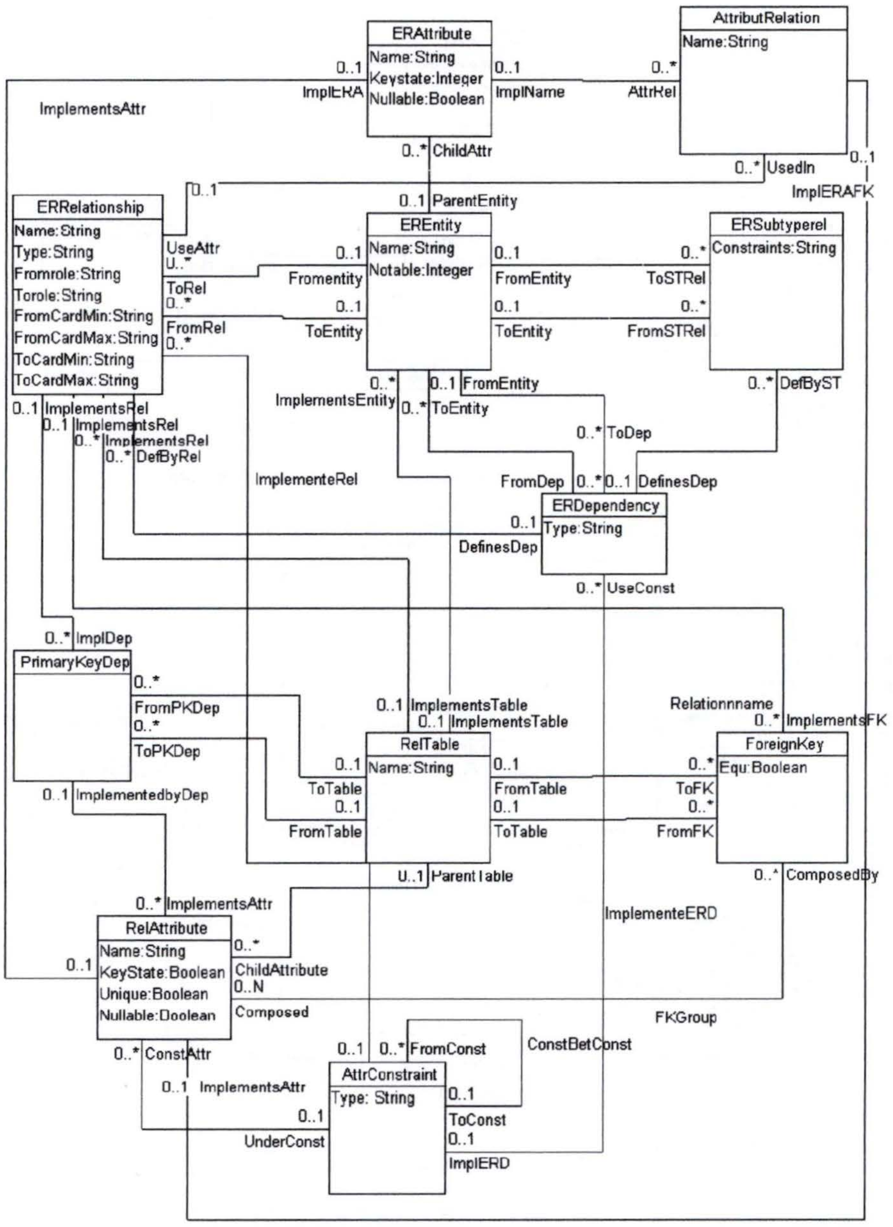


Figure 6.3: Repository version 3

Chapter 7

Conclusion

7.1 Purpose of this document

The purpose of this document was the creation of a set of tests to assess the quality of transformation rules implemented in CASE tools. These tests had to be based on real life examples. To achieve this goal, we have read number of papers and we selected four articles in four different domains each one presenting a solution to a specific problem.

From the analyze of the articles, we extracted firstly classical relationships. These classical relationships were divided into four different classes depending of their maximal cardinalities. Each relationship has one or more transformation rules to be implemented in the relational model. These transformations are well defined in the literature. They are symmetrically reversible, that means no semantical information is lost thought the transformation process.

We extracted secondly three categories of uncommon transformation rules. First category contains relationships with constraints that can not be expressed in the conceptual models. Second category contains relationships that have to be transformed in using an unstudied transformation rule. Last category contains relationships that have to be transformed with a rule that looses semantical information through the transformation process.

The second part of the work was an evaluation of existing CASE tools using the transformations extracted from the analyze of the selected articles.

For each transformation we assess the capability of each tool to apply the rule. In case of problem in the process, we wrote a result table to sum up the difficulties.

This part of the work highlights two main problems. Firstly, the quality of the implementation of classical transformation rules in CASE tools is below what we expected, excepted in DB-main. Some constraints are dropped, some optimizations are made, some semantical information are lost and some concepts are totally ignored (temporal relationship). Secondly, the studied tools are completely static. There is no way to add users' transformation rules.

In order to find a solution to this second problem, and implicitly a solution to the first one too, we have presented the work of Gustavsson [Guvstavsson, 2003] with OCL+. The purpose of his proof of concept tool is to be able to share a conceptual schema with its transformation rules. To reach this goal, the author has defined an extension of OCL (Object Constraint Language) in UML. OCL+ is a transformation oriented language making possible to write our own transformation rules using OCL+ and a repository formalized in UML and with an active database system.

Using OCL+, we were able to build rules to transform our practice based schemes. By implementing our rules, we highlight the importance of completeness of the repository. We were not able to find a lack in OCL+ for the expression of transformation rules. Anyway, two bugs or lacks have been found in the proof of the concept tool: the absence of *if-then-else* statement and the absence of concatenation for string. Nevertheless, we finally showed that the prototype works and is a good solution to solve uncommon design problems.

7.2 Future works

The goal of this work was not to assess every possible transformation rule implemented in CASE tools. We limited our analyze to the rules used in papers presenting database schemes.

Nevertheless, some types of relationships are not supported in most of tested tools. Ternary relationships, relation linking three entities, or attributes in relationships are not supported by ER-win and by Rational Rose for example. We did not find a practice based example that is not a school example using these relationships but, theoretical examples demonstrate such relations should be useful and should be implemented. Transformation of such relationships was not possible due to the absence of the *if-then-else* instruction in the proof of concept tool, but using this instruction, the implementation of such rules should be interesting.

We showed the importance of completeness of the repository. The repository have to be able to store every data needed by a model and every data needed to apply the transformation rules. These data are not part of the model and are used by the tool only for internal reasons. If the repository is not able to store these data, some transformation rules should not be applied. The used repository is based on a small part of ER model. The study of a complete repository able to store every needed information to be able to transform rightly a schema from one model to another should be interesting.

Bibliography

[IDE,] Idef1x: Technical report. `http://www.idet.com/Downloads/pdf/Idef1x.pdf`.

[Butler, 2000] Butler, T. (2000). Transforming information systems development through computer-aided system engineering (case): lessons from practice. *Information Systems Journal*, 10(3):167–193.

[Chen and Carlis, 2003] Chen, J. and Carlis, J. (2003). Genomic data modeling. *Information Systems*, 28:287–310.

[Detienne and Hainaut, 2001] Detienne, V. and Hainaut, J.-L. (2001). Case tool support for temporal database design. In *20th international conference on conceptual modeling (ER 2001)*.

[Dixon, 1992] Dixon, R. (1992). *Winning with CASE: Managing Modern Software Development*. McGraw-Hill, New-York.

[G. Sunye and Jezequel, 2002] G. Sunye, A. L. G. and Jezequel, J.-M. (2002). Using uml action semantics for model execution and transformation. *Information Systems*, 27:445–457.

[Guvstavson, 2003] Guvstavson, H. (2003). Oral discussion.

[Guvstavsson, 2003] Guvstavsson, H. (2003). Maintaining modelling transparency in multi-tool environments through standards based interchange of design transformation. submited to the University of Exeter, UK as thesis for the degree of Doctor of Philosophy in Computer Science.

[Hainaut, 2002] Hainaut, J.-L. (2002). Syllabus of database engineering, university of namur, belgium.

[Hughes and Wood-Harper, 2000] Hughes, J. and Wood-Harper, T. (2000). An empirical model of the information systems development process: a case study of an automotive manufacturer. *Accounting Forum*, 24(4):391–406.

[J-L. Hainaut and Roland, 1996] J-L. Hainaut, J-M Hick, V. E. J. H. and Roland, D. (1996). Understanding the implementation of is-a relation. In *Proc. of the 15th Int. Conf. on ER Approach, Cothbus, Springer-Verlag*, pages 42–57.

[Jackson, 1990] Jackson, M. (1990). Case tools and development methods. *Spurr, K., Layzell, P. (Eds.). CASE on trial. Wiley, Chichester*, pages 95–104.

[James and Finkelstein, 1981] James, M. and Finkelstein, C. (1981). Information engineering: Technical report.

[Kolp and Zimanyi, 2000] Kolp, M. and Zimanyi, E. (2000). Enhanced er to relational mapping and interrelational normalization. *Information and Software Technology*, 42:1057–1073.

[Lundell and Lings, 1999] Lundell, B. and Lings, B. (1999). Method support for developing evaluation frameworks for case tool evaluation. In *Khosrowpour, Mehdi (Ed.) Information Resources Management Association International Conference*, pages 350–358.

[OMG, 2003] OMG, O. M. G. I. (2003). Omg unified modeling language specification. March 2003, Version 1.5.

[Penicka and Friedsam, 2002] Penicka, J. and Friedsam, H. (2002). New database design for the aps survey and alignment data. In *the 7th International Workshop on Accelerator Alignment, SPring-8.*

[Shapiro, 1997] Shapiro, S. (1997). Splitting the difference: the historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, 19(1):20–54.

[Widom, 1996] Widom, J. (1996). The starbust active database rule system. *IEEE transactions on Knowledge and data engineering*, 8(4):583–595.

[Willson, 1998] Willson, S. (1998). Measuring inconsistency in phylogenetic trees. *J. Theor. Biol.*, 190(1):15–36.

Appendix A

Using CASE tools

A.1 Using ERwin

A.1.1 Conceptual phase

Creating entities

An entity in ERwin is represented by a box divided in three parts. The entity's name is written in the upper box, the primary key's attributes are written in the middle box and the other attributes are written in the bottom box.

A weak entity is an entity where the primary key is composed by one or more relation(s). The entity is represented in ERwin by a box with rounded corner.

Creating relationship

ERwin distinguishes between four kind of relationship: sub-category, non identifying relationship, identifying relationship and many to many relationships. The three first relationships link two kind of entities: the parent entity and the child entity. During the creation of a relationship, user clicks first on the parent entity and clicks after on the child entity.

The sub-category relationship is a relation where the parent entity is the super type and the child entity is the sub type. A child entity owns all the characteristics of its super type. According to the IDEF1x [IDE,] definition:

rule A: A category entity can have only one generic entity. That is, it can only be a member of the set of categories for one category cluster.

(..)

rule D: The primary key attribute(s) of a category entity must be the same as the primary key attribute(s) of the generic entity. However, role names may be assigned in the category entity.

(..)

A category entity cannot be a child entity in an identifying connection relationship unless the primary key contributed by the identifying relationship is completely contained within the primary key of the category, while at the same time the category primary key satisfies rule d above.

ERwin distinguishes between two kind of sub-category relationship: exclusive and inclusive. Exclusive sub-category does not allow two children of the same entity reference the same instance of their parent entity. Inclusive sub-category accepts this construct.

Non identifying relationship creates a foreign key in the child attribute. This foreign key isn't a part of the primary key. The group box cardinality contains the cardinality for the child. The nulls group box contains the minimal cardinality for the parent entity. Pay attention to 'zero or one' cardinality, an alternate key have to be had.

The identifying relationship create a foreign key in the child attribute. This foreign key is a part of the primary key. As in the non identifying relationship, the group box cardinality contains the cardinality for the child. The foreign key for this kind of relation can't be null, the nulls group isn't actived. Pay attention to zero or one cardinality, an alternate key have to be had.

The many to many relationship does not allow any configuration. The minimal cardinality is zero for both side of the relation.

An alternate key have to be create if all instances of an attribute or a group of attributes must be unique. During the creation of a relationship, if the maximal cardinality is one (P), a alternate key must be added. We click with the right mouse's right button on the child entity→key group. We click on *New* button, we choose *Alternate key.* The Key is now added, we add now all attribute coming from the foreign key.

A.1.2 Logical phase

The menu *tool→Derive new model* gives us the way to transform logical schema to physical schema .

In the first wizard screen, we choose *Physical* in the *New Type Model* group. We choose too *Oracle 8.* It's normally the default option.

On the second screen, we select *Many-to-many relationship* and *Super-type/subtype* in the *Auto transform logical objects* group.

There is no special action to do in the last screen.

A.1.3 Generating SQL code

The menu *tool→Forward Engineer /Schema Generation* gives us the way to generate SQL code from the physical schema.

In the wizard screen, we change the property of *Referential Integrity* and we check the *Unique (AK)* property.

A.2 Using Rational Rose

A.2.1 Conceptual phase

Rational Rose use UML to represent schemes. UML editor allow to represent classes for a oriented object application. For this reason, it is impossible to create a primary key in a class (normally implicit for each class in oriented object). The conceptual phase consist in creating a new package. This package will contain all our entities.

Creating entities

We create each entity. In entity's option, we select *Persistent.* We add all attributes, even if this attribute is a part of the primary key.

Creating relationship

As ERwin, Rose distinguishes four kinds of relationships: sub-category, non identifying relationships, identifying relationships and many to many relationships. As in ERwin too, after selecting the kind of relationship, user click first on the parent entity and secondly on the child entity. Translated in the relational model, a foreign key is created in the child entity and references the parent entity.

Sub-category relationship is non-disjunctive and partial. Rose does not allow another kind of sub-category.

For three others relationships, after creating the relationship, user defines the cardinality in relationship's option.

A.2.2 Logical phase

Datamodeler in rose is the logical phase. We transform the package to a new datamodel. All persistent entity are added in this new model, and all links are created. A sub-category relationship is transformed to a unique foreign key. For a many to many relationship, a new table is created with two foreign keys referencing the initial entities. For all child entities with one or more identifying relationship or with one sub-category, the foreign key implementing the relationship is part of the primary key. For other entity, a new attribute is added and implemented the primary key.

At this moment, some modification have to be made on the new model. First we delete all automatically added attribute which are not a part of a foreign key. Secondly, we recreate the real primary key group and we add all attribute in this group. If the primary key is compounded by one attribute part of a foreign key and another attribute, we simply add this attribute in the primary key group.

A.2.3 Generating SQL code

We can generate SQL code of a data model with the data model context menu: *Data Modeler→Forward Engineering*.

A.3 Using DB-main

A.3.1 Conceptual phase

Schema creation

An entity is a box, divided in three parts. The first contains the entity's name, the second contains the attribute's list and the last contains information about this attribute (primary key...). To create an entity, we click on the entity's icon and click in the schema. To add attributes in the entity, we click on the attribute's icon, and click on the entity, in the schema.

A relationship is an hexagon. Relationship's name is written inside this hexagon. To create a relationship, we click on the relationship's icon and click in the schema. After that, we click on the link icon, click on the relationship and finally on the entity to link. We redo this operation until all wanted entities have been linked.

An Is-a relationship is represented by a triangle. The bold line links the super type entity and the normal line links the subtype entity. To create an Is-a relationship, we doubleclick on the subtype entity and we add all the super type entity we want.

A.3.2 Logical phase

Individual transformation

DBmain ensures to transform every relationships individually. To transform a relationship, we select first the relation. The menu *Transform→Rel-Type* is now accessible. We use systematically the *→Attribute* transformation.

To Transform an Is-a relationship, we select first the child entity. The menu *Transform-¿Entity Type* is now accessible. An Is-a relationship can be transformed in different ways. We use mainly the *Is-a→Rel-type* function (transform the Is-a to a relationship) and the *split/merge* function (merge the subtype with the super type.

To relation transformation

DBmain ensures to transform the EA schema directly in the relational model. Every relationships are translated with a foreign key, all Is-a relationships are translated to a unique foreign key and all many to many relationships are translated in a table with foreign key's group referencing all entities linked to this relationship. If a relationship was already transformed individually into the relational model as explain upper, this relationship does not change.

Global Transformation

DBmain ensures to make a script to transform the EA schema. This script is made up of one or more action to be made on each object with a certain precondition.

A.3.3 Generating SQL code

The SQL code generation can be made in different way. The menu *File→Generate* ensures the generation in different codes. In all our example, we use the *Academic SQL (check)* function to generate the code.

Appendix B

Testing tools

B.1 Result table

B.1.1 Zero to many non-identifying optional relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.2 Zero to many non-identifying mandatory relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.3 Zero to many identifying relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.4 One to many identifying relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	None
Logical	Ok	none
SQL	Ko	Mandatory constraint on parent entity is not implemented

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ko	Mandatory constraint on parent entity is not implemented

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.5 Zero to one identifying relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	A unique constraint must be added on the foreign key. Normally, we don't have to care about foreign key in the conceptual phase
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.6 Many to many relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.7 Composition by two many to many relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ko	Both relationship are implemented by a unique table
SQL	Ko	NA

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.8 Is-a disjunctive relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	No trigger are created to implement the disjunctive constraint.

- Rational Rose

Phase	Result	Notes
Conceptual	Ko	Impossible to represent a disjunctive subtype
Logical	Ko	Constraint impossible to represent
SQL	Ko	NA

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.9 Is-a non disjunctive relationship

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ok	none
SQL	Ok	none

B.1.10 Limitation in conceptual model

- ERwin

Phase	Result	Notes
Conceptual	Ok	None
Logical	Ok	none
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ko	The constraints can't be expressed Problem with the primary key
Logical	Ok	none
SQL	Ok	none

- DB-main

Phase	Result	Notes
Conceptual	Ko	The constraints can't be expressed
Logical	Ok	none
SQL	Ok	none

B.1.11 *Temporal* Is-a relationship

- ERwin

Phase	Result	Notes
Conceptual	Ko	ER-win doesn't support this entity type.
Logical	Ko	NA
SQL	Ko	NA

- Rational Rose

Phase	Result	Notes
Conceptual	Ko	Rose doesn't support this entity type
Logical	Ko	NA
SQL	Ko	NA

- DB-main

Phase	Result	Notes
Conceptual	Ok	Three kind of temporal entity exist, Hainaut...
Logical	Ko	A temporal entity is implemented with two attributes (beginning date and ending date), we need one attribute
SQL	Ko	NA

B.1.12 Semantical information lost

- ERwin

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ko	Impossible to merge two entities linked by a 0-1 relationship Using a sub-type instead of 0-1 relationship between 'MainActivity' and 'IndivudualComment', merging transformation works, but we lose the 'MiscComment' entity.
SQL	Ok	none

- Rational Rose

Phase	Result	Notes
Conceptual	Ko	Disjunctive constraints can't be expressed
Logical	Ko	Using a non-disjunctive Is-a, Rose is not able to merge two entities
SQL	Ko	NA

- DB-main

Phase	Result	Notes
Conceptual	Ok	none
Logical	Ko	We are able to merge entities 'MainActivity' and 'IndivudualComment' but we are not able to merge the new entity with 'comment' because of semantic information lost.
SQL	Ko	NA

Appendix C

Architecture developed for the ASTRID project

C.1 Repository

Version 1 Original version of the repository

Version 1 First version of the repository

Version 2 Adds the possibility to add constraints on relational attribute.

Version 3 Used in the biological example.

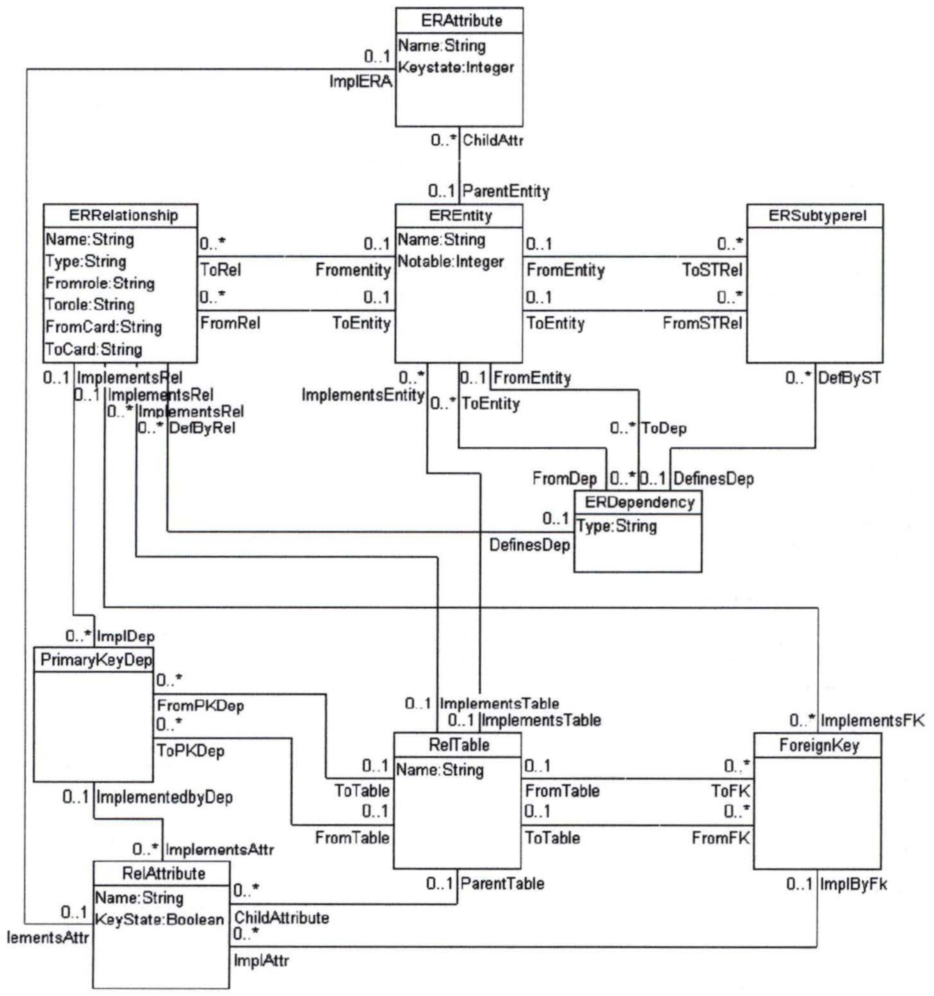


Figure C.1: Original Metamodel

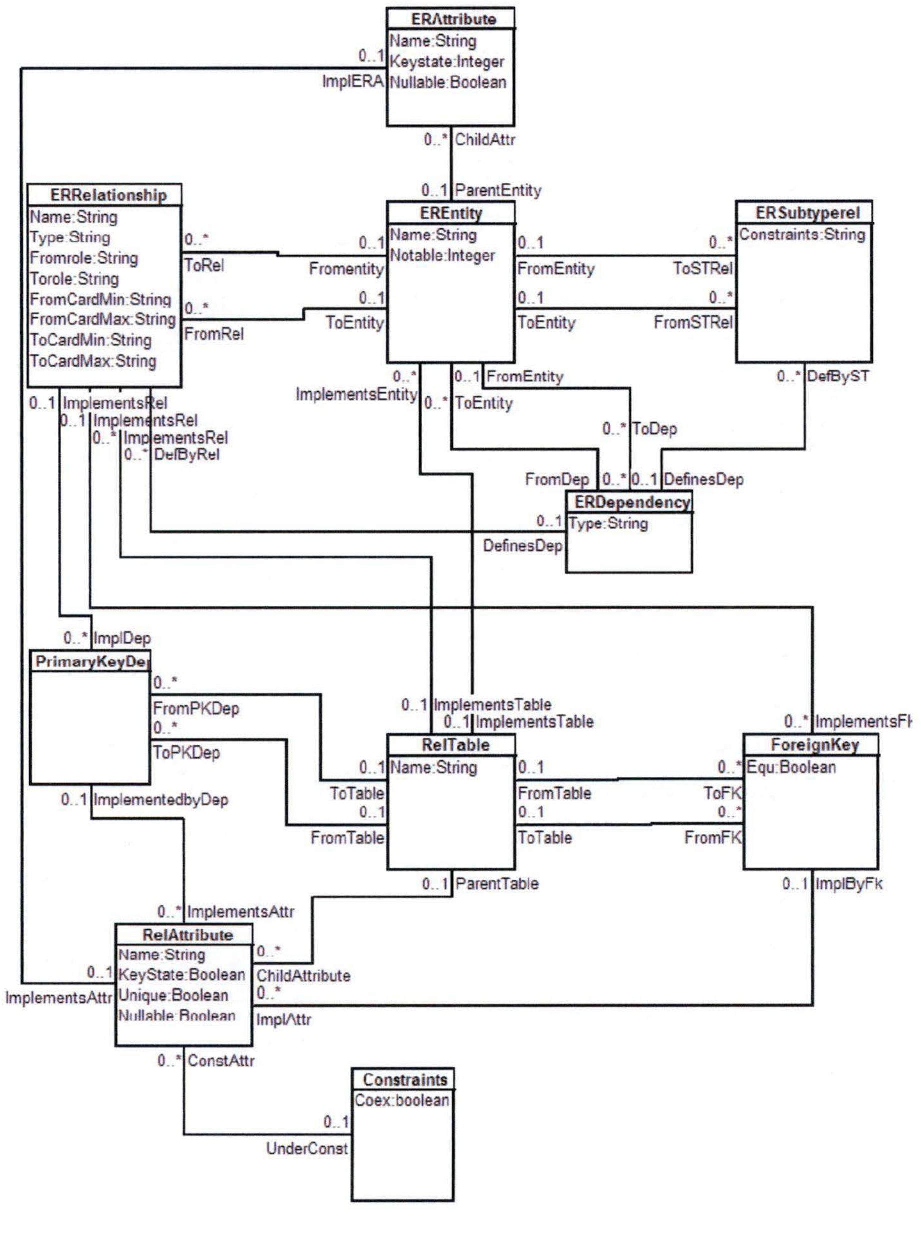


Figure C.2: Metamodel version 1

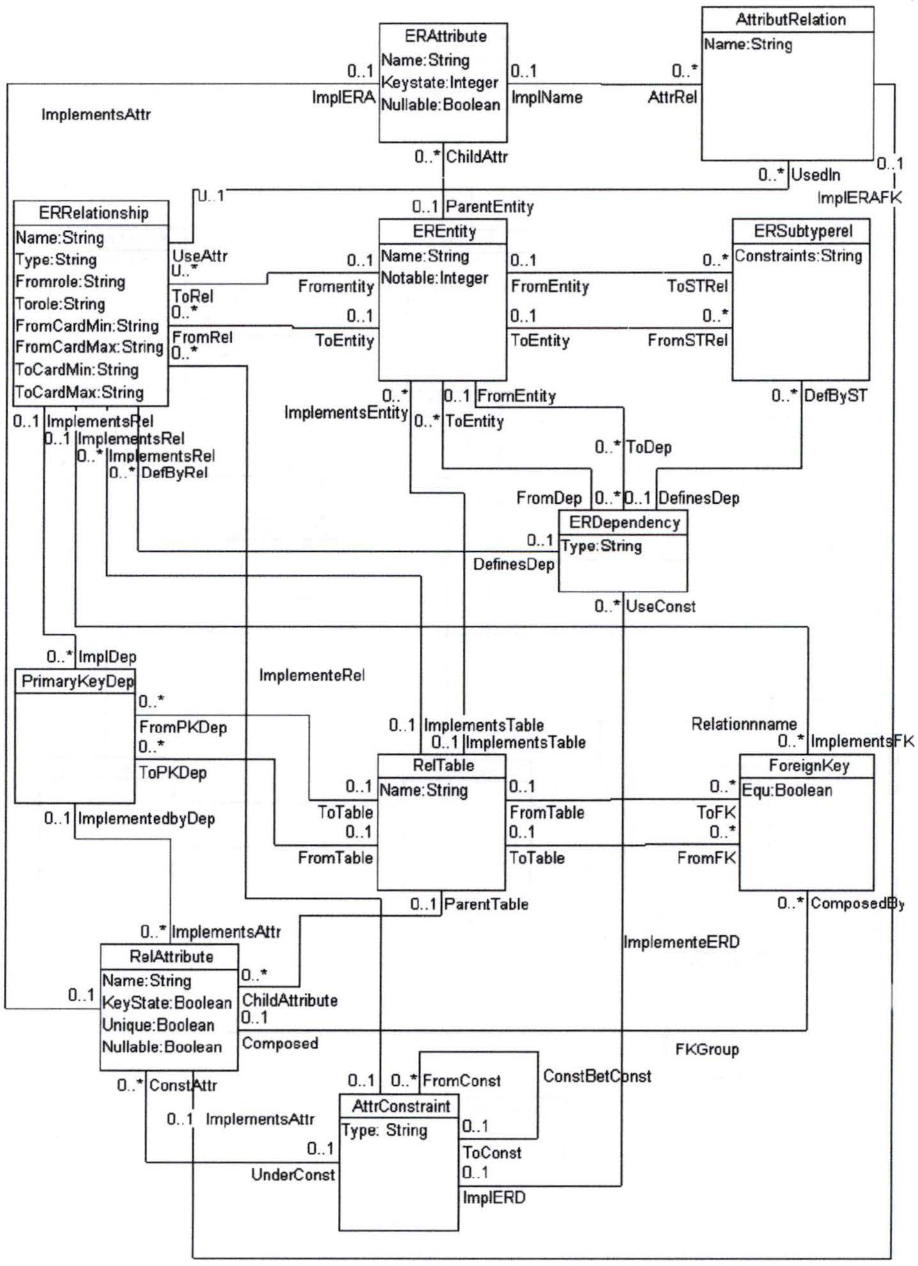


Figure C.3: Metamodel version 2

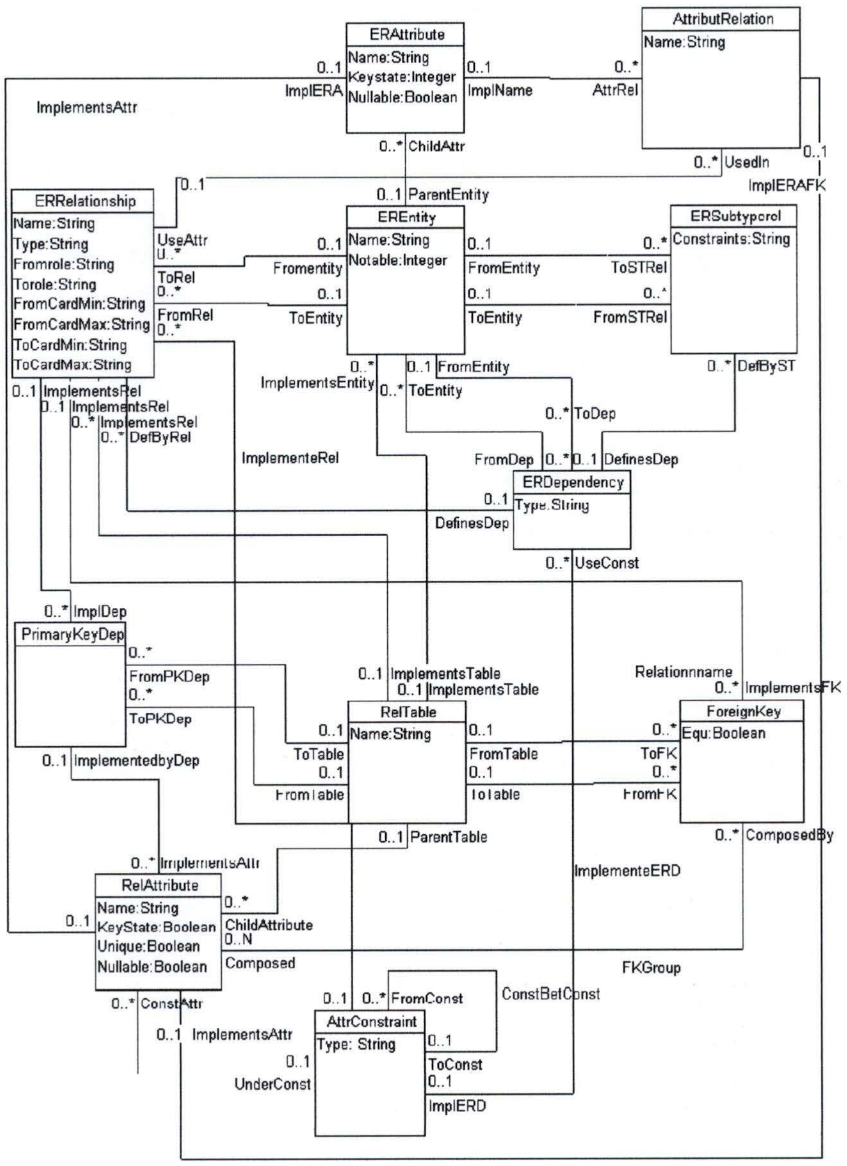


Figure C.4: Metamodel version 3

C.2 Rules

1. Entity to Table

Repository	v. 1
Context Class	EREntity
Event	Insert
Declaration	RelTable RT
Condition	ImplementsTable→isempty and notable=false

Action	RT.create;
	RT.name:=self.name;
	self.ImplementsTable:=RT

2. Attribute to RelAttribute

Repository	v. 1
Context Class	ERAttribute
Event	Insert
Declaration	RelAttribute RA
Condition	ImplementsAttr→isempty and ParentEntity.notable=false

Action	RA.Create;
	RA.ImplERA:=self;
	RA.Name:=self.Name;
	RA.KeyState:=self.KeyState;
	RA.Nullable=self.Nullable;
	RA.ParentTable:=self.ParentEntity.ImplementsTable

3. *0-1 0-N* non identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk

Condition FromCardMin=”0” and ToCardMin=”0” and FromCardMax=”N” and ToCardMax=”1” and Type<>”W”

Action
```
FK.Create;
FK.Equ:=false;
FK.ImplementsRel:=self;
FK.FromTable:=self.FromEntity.ImplementsTable;
FK.ToTable:=self.ToEntity.ImplementsTable;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplByFk:=FK)
```

4. *1-1 0-N* non identifying relationship

Repository v. 1

Context Class ERRelationship

Event Insert

Declaration RelAttribute Re, ForeignKey Fk

Condition FromCardMin=”0” and ToCardMin=”1” and FromCardMax=”N” and ToCardMax=”1” and Type=”N”

Action
```
FK.Create;
FK.Equ:=false;
FK.ImplementsRel:=self;
FK.FromTable:=self.FromEntity.ImplementsTable;
FK.ToTable:=self.ToEntity.ImplementsTable;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplByFk:=FK)
```

5. *0-1 1-N* non identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk
Condition	FromCardMin="1" and ToCardMin="0" and FromCardMax="N" and ToCardMax="1" and Type="N"

Action
```
FK.Create;
FK.Equ:=true;
FK.ImplementsRel:=self;
FK.FromTable:=self.FromEntity.ImplementsTable;
FK.ToTable:=self.ToEntity.ImplementsTable;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplByFk:=FK)
```

6. *1-1 1-N* non identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert

```
 Declaration  RelAttribute Re, ForeignKey Fk
   Condition  FromCardMin="1" and ToCardMin="1" and FromCardMax="N"
              and ToCardMax="1" and Type="N"
      Action  FK.Create;
              FK.Equ:=true;
              FK.ImplementsRel:=self;
              FK.FromTable:=self.FromEntity.ImplementsTable;
              FK.ToTable:=self.ToEntity.ImplementsTable;
              self.FromEntity.ImplementsTable.childAttribute→
              reject(Keystate=false)→
              iterate( PK1|
              Re.Create;
              Re.Keystate:=false;
              Re.Nullable:=false;
              Re.Name:=PK1.name;
              Re.ParentTable:=self.ToEntity.ImplementsTable;
              Re.ImplByFk:=FK)
```

7. *1-1 0-N* identifying relationship

```
  Repository  v. 1
Context Class  ERRelationship
       Event  Insert
 Declaration  RelAttribute Re, ForeignKey Fk, PrimaryKeyDep Pk
   Condition  FromCardMin="0" and ToCardMin="1" and FromCardMax="N"
              and ToCardMax="1" and Type="I"
      Action  FK.Create;
              FK.Equ:=false;
              FK.ImplementsRel:=self;
              FK.FromTable:=self.FromEntity.ImplementsTable;
              FK.ToTable:=self.ToEntity.ImplementsTable;
              Pk.Create;
              Pk.FromTable:=self.FromEntity.ImplementsTable;
```

```
PK.ToTable:=self.ToEntity.ImplementsTable;
Pk.ImplementsRel:=self;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Re.ImplByFk:=FK)
```

8. *1-1 1-N* identifying relationship

```
Repository     v. 1
Context Class  ERRelationship
Event          Insert
Declaration    RelAttribute Re, ForeignKey Fk, PrimaryKeyDep Pk
Condition      FromCardMin="1" and ToCardMin="1" and FromCardMax="N"
               and ToCardMax="1" and Type="I"
Action         FK.Create;
               FK.Equ:=true;
               FK.ImplementsRel:=self;
               FK.FromTable:=self.FromEntity.ImplementsTable;
               FK.ToTable:=self.ToEntity.ImplementsTable;
               Pk.Create;
               Pk.FromTable:=self.FromEntity.ImplementsTable;
               PK.ToTable:=self.ToEntity.ImplementsTable;
               Pk.ImplementsRel:=self;
               self.FromEntity.ImplementsTable.childAttribute→
               reject(Keystate=false)→
               iterate( PK1|
               Re.Create;
```

Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Re.ImplByFk:=FK)

9. *1-1 0-1* non identifying relationship

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action	FK.Create;
	FK.Equ:=false;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;
	AC.Create;
	AC.Type="unique"; self.FromEntity.ImplementsTable.childAttribute→
	reject(Keystate=false)→
	iterate(PK1|
	Re.Create;
	Re.Keystate:=false;
	Re.Nullable:=false;
	Re.Name:=PK1.name;
	Re.ParentTable:=self.ToEntity.ImplementsTable;
	union(AC.underconst(Re)); Re.ImplByFk:=FK)

10. *1-1 1-1* non identifying relationship

Repository	v. 2
Context Class	ERRelationship

Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC
Condition	FromCardMin="1" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action	FK.Create;
	FK.Equ:=true;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;
	AC.Create;
	AC.Type="unique"; self.FromEntity.ImplementsTable.childAttribute→
	reject(Keystate=false)→
	iterate(PK1|
	Re.Create;
	Re.Keystate:=false;
	Re.Nullable:=false;
	Re.Name:=PK1.name;
	Re.ParentTable:=self.ToEntity.ImplementsTable;
	union(AC.underconst(Re)); Re.ImplByFk:=FK)

11. *0-1 0-1* non identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC
Condition	FromCardMin="0" and ToCardMin="0" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action	FK.Create;
	FK.Equ:=false;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;

AC.Create;
AC.Type="unique"; self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate(PK1|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
union(AC.underconst(Re)); Re.ImplByFk:=FK)

12. *1-1 0-1* identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, PrimaryKeyDep Pk, AttrConstraint AC
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="I"
Action	FK.Create;

FK.Equ:=false;
FK.ImplementsRel:=self;
FK.FromTable:=self.FromEntity.ImplementsTable;
FK.ToTable:=self.ToEntity.ImplementsTable;
Pk.Create;
Pk.FromTable:=self.FromEntity.ImplementsTable;
PK.ToTable:=self.ToEntity.ImplementsTable;
Pk.ImplementsRel:=self;
AC.Create;
AC.Type="unique"; self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate(PK1|
Re.Create;

Re.Keystate:=true;

Re.Nullable:=false;

Re.Name:=PK1.name;

Re.ParentTable:=self.ToEntity.ImplementsTable;

Re.ImplementedbyDep:=Pk;

union(AC.underconst(Re)); Re.ImplByFk:=FK)

13. *1-1 1-1* identifying relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, PrimaryKeyDep Pk, AttrConstraint AC
Condition	FromCardMin="1" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="I"

Action FK.Create;

FK.Equ:=true;

FK.ImplementsRel:=self;

FK.FromTable:=self.FromEntity.ImplementsTable;

FK.ToTable:=self.ToEntity.ImplementsTable;

Pk.Create;

Pk.FromTable:=self.FromEntity.ImplementsTable;

PK.ToTable:=self.ToEntity.ImplementsTable;

Pk.ImplementsRel:=self;

AC.Create;

AC.Type="unique"; self.FromEntity.ImplementsTable.childAttribute→

reject(Keystate=false)→

iterate(PK1|

Re.Create;

Re.Keystate:=true;

Re.Nullable:=false;

Re.Name:=PK1.name;

Re.ParentTable:=self.ToEntity.ImplementsTable;

```
Re.ImplementedbyDep:=Pk;
union(AC.underconst(Re)); Re.ImplByFk:=FK)
```

14. merging *1-1 0-1* non identifying relationship

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC1, AttrConstraint AC2
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action
```
AC1.Create;
AC1.Type="coexistence";
AC2.Create;
AC2.Type="unique";
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( Att|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=Att.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
union(AC1.underconst(Re)); union(AC2.underconst(Re)); Re.ImplByFk:=FK);
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=true)→
iterate( Att|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=Att.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
union(AC1.underconst(Re)); Re.ImplByFk:=FK)
```

15. merging *0-1 0-1* non identifying relationship

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC2
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action
```
AC2.Create;
AC2.Type="unique";
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( Att|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=Att.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
union(AC2.underconst(Re)); Re.ImplByFk:=FK); self.FromEntity.ImplementsTable.child
reject(Keystate=true)→
iterate( Att|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=Att.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplByFk:=FK)
```

16. merging *1-1 0-1* identifying relationship

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	RelAttribute Re, ForeignKey Fk, AttrConstraint AC1

Condition FromCardMin="0" and ToCardMin="1" and FromCardMax="1" and ToCardMax="1" and Type="N"

Action AC1.Create;
AC1.Type="coexistence";
AC2.Create;
AC2.Type="unique";
self.FromEntity.ImplementsTable.childAttribute→
iterate(Att|
Re.Create;
Re.Keystate:=false;
Re.Nullable:=true;
Re.Name:=Att.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
union(AC1.underconst(Re)); Re.ImplByFk:=FK);

17. merging *Is-a* disjoint

Repository v. 2

Context Class ERDependency

Event Insert

Declaration RelAttribute Re, ERRelationship ER, AttrConstraint AC1, AttrConstraint AC2

Condition Type="D"

Action AC1.Create;
AC1.Type="exculsion";
self.DefBySTi→
iterate(SubType|
AC2.Create; AC2.Type="coexistence";
Union(AC2.FromConst(AC1));
SubType.FromEntity.Implementstable := SubType.ToEntity.Implementstable;
SubType.FromEntity.Implementstable.childAttribute→
iterate(Attr|
Re.Create;
Re.Keystate:=false;

Re.Nullable:=true;
Re.Name:=Attr.name;
Re.ParentTable:=SubType.ToEntity.Implementstable;
union(AC2.underconst(Re));

18. merging *Is-a* Overlapped

Repository	v. 2
Context Class	ERDependency
Event	Insert
Declaration	RelAttribute Re, ERRelationship ER, AttrConstraint AC2
Condition	Type="D"

Action	AC1.Create;
	self.DefByST→
	iterate(SubType|
	AC2.Create; AC2.Type="coexistence";
	SubType.FromEntity.Implementstable := SubType.ToEntity.Implementstable;
	SubType.FromEntity.Implementstable.childAttribute→
	iterate(Attr|
	Re.Create;
	Re.Keystate:=false;
	Re.Nullable:=true;
	Re.Name:=Attr.name;
	Re.ParentTable:=SubType.ToEntity.Implementstable;
	union(AC2.underconst(Re));

19. *0-N 0-N* relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelTable Rt, PrimaryKeyDep Pk1, PrimaryKeyDep Pk2, ForeignKey Fk1, ForeignKey Fk2, RelAttribute Re
Condition	FromCardMin="0" and ToCardMin="0" and FromCardMax="N" and ToCardMax="N" and Type="N"

```
Action Rt.Create;
       Rt.Name:=self.Name;
       self.ImplementsTable:=Rt;
       Pk1.Create;
       Pk1.ImplementsRel:=self;
       Pk1.FromTable:=self.FromEntity.ImplementsTable;
       Pk1.ToTable:=RT;
       Pk2.Create;
       Pk2.ImplementsRel:=self;
       Pk2.FromTable:=self.ToEntity.ImplementsTable;
       Pk2.ToTable:=Rt;
       Fk1.Create;
       Fk1.Equ:=false;
       Fk1.ImplementsRel:=self;
       Fk1.FromTable:=self.FromEntity.ImplementsTable;
       Fk1.ToTable:=Rt;
       Fk2.Create;
       Fk2.Equ:=false;
       Fk2.ImplementsRel:=self;
       Fk2.FromTable:=self.ToEntity.ImplementsTable;
       Fk2.ToTable:=Rt;
       self.FromEntity.ImplementsTable.childAttribute→
       reject(Keystate=false)→
       iterate( PK|
       Re.Create;
       Re.Keystate:=true;
       Re.Nullable:=false;
       Re.Name:=PK.name;
       Re.ParentTable:=Rt;
       Re.ImplementedbyDep:=Pk1;
       Re.ImplByFk:=Fk1);
       self.ToEntity.ImplementsTable.childAttribute→
       reject(Keystate=false)→
       iterate( PK|
```

Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk2;
Re.ImplByFk:=Fk2)

20. *1-N 1-N* relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelTable Rt, PrimaryKeyDep Pk1, PrimaryKeyDep Pk2, ForeignKey Fk1, ForeignKey Fk2, RelAttribute Re
Condition	FromCardMin="1" and ToCardMin="1" and FromCardMax="N" and ToCardMax="N" and Type="N"
Action	Rt.Create;

Rt.Name:=self.Name;
self.ImplementsTable:=Rt;
Pk1.Create;
Pk1.ImplementsRel:=self;
Pk1.FromTable:=self.FromEntity.ImplementsTable;
Pk1.ToTable:=RT;
Pk2.Create;
Pk2.ImplementsRel:=self;
Pk2.FromTable:=self.ToEntity.ImplementsTable;
Pk2.ToTable:=RT;
Fk1.Create;
Fk1.Equ:=true;
Fk1.ImplementsRel:=self;
Fk1.FromTable:=self.FromEntity.ImplementsTable;
Fk1.ToTable:=RT;
Fk2.Create;

```
Fk2.Equ:=true;
Fk2.ImplementsRel:=self;
Fk2.FromTable:=self.ToEntity.ImplementsTable;
Fk2.ToTable:=RT;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk1;
Re.ImplByFk:=Fk1);
self.ToEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk2;
Re.ImplByFk:=Fk2)
```

21. *0-N - 1-N* relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelTable Rt, PrimaryKeyDep Pk1, PrimaryKeyDep Pk2, ForeignKey Fk1, ForeignKey Fk2, RelAttribute Re
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="N" and ToCardMax="N" and Type="N"

```
Action RT.Create;
        Rt.Name:=ERRelationship.Name;
        self.ImplementsTable:=Rt;
        Pk1.Create;
        Pk1.ImplementsRel:=self;
        Pk1.FromTable:=self.FromEntity.ImplementsTable;
        Pk1.ToTable:=RT;
        Pk2.Create;
        Pk2.ImplementsRel:=self;
        Pk2.FromTable:=self.ToEntity.ImplementsTable;
        Pk2.ToTable:=RT;
        Fk1.Create;
        Fk1.Equ:=false;
        Fk1.ImplementsRel:=self;
        Fk1.FromTable:=self.FromEntity.ImplementsTable;
        Fk1.ToTable:=RT;
        Fk2.Create;
        Fk2.Equ:=true;
        Fk2.ImplementsRel:=self;
        Fk2.FromTable:=self.ToEntity.ImplementsTable;
        Fk2.ToTable:=RT;
        self.FromEntity.ImplementsTable.childAttribute→
        reject(Keystate=false)→
        iterate( PK|
        Re.Create;
        Re.Keystate:=true;
        Re.Nullable:=false;
        Re.Name:=PK.name;
        Re.ParentTable:=Rt;
        Re.ImplementedbyDep:=Pk1;
        Re.ImplByFk:=Fk1);
        self.ToEntity.ImplementsTable.childAttribute→
        reject(Keystate=false)→
        iterate( PK|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk2;
Re.ImplByFk:=Fk2)
```

22. *1-N* - *0-N* relationship

Repository	v. 1
Context Class	ERRelationship
Event	Insert
Declaration	RelTable Rt, PrimaryKeyDep Pk1, PrimaryKeyDep Pk2, ForeignKey Fk1, ForeignKey Fk2, RelAttribute Re
Condition	FromCardMin="1" and ToCardMin="0" and FromCardMax="N" and ToCardMax="N" and Type="N"

Action
```
Rt.Create;
Rt.Name:=self.Name;
self.ImplementsTable:=Rt;
Pk1.Create;
Pk1.ImplementsRel:=self;
Pk1.FromTable:=self.FromEntity.ImplementsTable;
Pk1.ToTable:=RT;
Pk2.Create;
Pk2.ImplementsRel:=self;
Pk2.FromTable:=self.ToEntity.ImplementsTable;
Pk2.ToTable:=RT;
Fk1.Create;
Fk1.Equ:=true;
Fk1.ImplementsRel:=self;
Fk1.FromTable:=self.FromEntity.ImplementsTable;
Fk1.ToTable:=RT;
Fk2.Create;
```

```
Fk2.Equ:=false;
Fk2.ImplementsRel:=self;
Fk2.FromTable:=self.FromEntity.ImplementsTable;
Fk2.ToTable:=Rt;
self.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk1;
Re.ImplByFk:=Fk1);
self.ToEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK.name;
Re.ParentTable:=Rt;
Re.ImplementedbyDep:=Pk2;
Re.ImplByFk:=Fk2)
```

23. *Is-a* disjoint by FK

Repository	v. 2
Context Class	ERDependency
Event	Insert
Declaration	RelAttribute Re, ERRelationship ER, PrimaryKeyDep Pk, ForeignKey Fk, AttrConstraint AC1
Condition	Type="D"
Action	AC1.Create;

```
AC1.Type="exculsion";
self.DefBySTâ
iterate( SubType|
ER.Create;
ER.Name="string";
ER.Type="U";
ER.Fromrole:="string";
ER.Torole:="string";
ER.FromCardMin:="1";
ER.FromCardMax:="1";
ER.ToCardMin:="0";
ER.ToCardMax:="1";
ER.FromEntity:=SubType.FromEntity;
ER.ToEntity:=Subtype.ToEntity;
ER.DefinesDep:=self);
self.DefByRelâ
iterate( Rel|
Fk.Create;
Pk.Create;
Re.Create;
Re.Name:=Re.Fromentity.Name;
union(AC1.ConstAttr(Re);
Rel.FromEntity.ImplementsTable.childAttributeâ
reject(Keystate=false)â
iterate( PK1|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Re.ImplByFk:=FK)
```

24. *Is-a* overlapped by FK

Repository	v. 2
Context Class	ERDependency
Event	Insert
Declaration	RelAttribute Re, ERRelationship ER, PrimaryKeyDep Pk, ForeignKey Fk
Condition	Type="D"
Action	self.DefByST→

```
self.DefByST→
iterate( SubType|
ER.Create;
ER.Name="string";
ER.Type="U";
ER.Fromrole:="string";
ER.Torole:="string";
ER.FromCardMin:="1";
ER.FromCardMax:="1";
ER.ToCardMin:="0";
ER.ToCardMax:="1";
ER.FromEntity:=SubType.FromEntity;
ER.ToEntity:=Subtype.ToEntity;
ER.DefinesDep:=self);
self.DefByRel→
iterate( Rel|
Fk.Create;
Pk.Create;
Rel.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
```

Re.ImplByFk:=FK)

25. InsAttrRel

Repository	v. 2
Context Class	AttributRelation
Event	Insert
Declaration	RelAttribute RA
Condition	self.UseAttr.ToEntity.ImplementsTable.ChildAttribute→forall(RA|
	RA.name <> self.Name)
Action	RA.Create;
	RA.ParentTable:=self.UseAttr.ToEntity.ImplementsTable;
	RA.ImplERAFK := self;
	RA.Name := self.Name;
	RA.KeyState := false;
	RA.Nullable := true;

26. InsERRelni110N

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	ForeignKey FK
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="N"
	and ToCardMax="1" and Type="N" and UsedIn→notempty
Action	FK.Create;
	FK.Equ:=false;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;
	self.UsedIn→iterate(AR|
	AR.Nullable:=false;
	FK.Composed := FK.Composed→union(AR.ImplementAttr.ComposedBy(FK))

27. InsERRelni010N

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	ForeignKey FK
Condition	FromCardMin="0" and ToCardMin="0" and FromCardMax="N" and ToCardMax="1" and Type="N" and UsedIn→notempty

Action	FK.Create;
	FK.Equ:=false;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;
	self.UsedIn→iterate(AR|
	FK.Composed := FK.Composed→union(AR.ImplementAttr.ComposedBy(FK)))

28. InsERReli110N

Repository	v. 2
Context Class	ERRelationship
Event	Insert
Declaration	ForeignKey FK, PrimaryKeyDep PK
Condition	FromCardMin="0" and ToCardMin="1" and FromCardMax="N" and ToCardMax="1" and Type="I" and UsedIn→notempty

Action	FK.Create;
	FK.Equ:=false;
	FK.ImplementsRel:=self;
	FK.FromTable:=self.FromEntity.ImplementsTable;
	FK.ToTable:=self.ToEntity.ImplementsTable;
	Pk.Create;
	Pk.FromTable:=self.FromEntity.ImplementsTable;
	PK.ToTable:=self.ToEntity.ImplementsTable;
	self.UsedIn→iterate(AR|

```
AR.Nullable:=false;
FK.Composed := FK.Composed→union(AR.ImplementAttr.ComposedBy(FK));
PK.ImplementsAttr := PK.ImplementsAttr→union(AR.ImplementedbyDep(PK)))
```

29. Temporal *Is-a* overlapped by FK

Repository	v. 2
Context Class	ERDependency
Event	Insert
Declaration	RelAttribute Re, ERRelationship ER, PrimaryKeyDep Pk, ForeignKey Fk
Condition	Type="TO"
Action	self.DefByST→

```
self.DefByST→
iterate( SubType|
ER.Create;
ER.Name="string";
ER.Type="U";
ER.Fromrole:="string";
ER.Torole:="string";
ER.FromCardMin:="1";
ER.FromCardMax:="1";
ER.ToCardMin:="0";
ER.ToCardMax:="1";
ER.FromEntity:=SubType.FromEntity;
ER.ToEntity:=Subtype.ToEntity;
ER.DefinesDep:=self);
self.DefByRel→
iterate( Rel|
Fk.Create;
Pk.Create;
Re.Create;
Re.Keystate:=true;
Re.Nullalble:=false;
Re.Name:=D̈ate;̈
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Rel.FromEntity.ImplementsTable.childAttribute→
reject(Keystate=false)→
iterate( PK1|
Re.Create;
Re.Keystate:=true;
Re.Nullable:=false;
Re.Name:=PK1.name;
Re.ParentTable:=self.ToEntity.ImplementsTable;
Re.ImplementedbyDep:=Pk;
Re.ImplByFk:=FK)
```