Two experiences in software dynamics[♦]

Artur Boronat
(Polytechnic University of Valencia, Spain
aboronat@dsic.upv.es)

Jennifer Perez
(Polytechnic University of Valencia, Spain
jeperez@dsic.upv.es)

José Á. Carsí
(Polytechnic University of Valencia, Spain
pcarsi@dsic.upv.es)

Isidro Ramos
(Polytechnic University of Valencia, Spain
iramos@dsic.upv.es)

Abstract: This paper presents an outline of a formal model management framework developed in two projects, which provide breakthroughs for legacy systems recovery (RELS) and for data migration (ADAM). To recover a legacy system, we follow an algebraic approach by using algebras in order to represent the models and manipulate them. RELS also automatically generates a data migration plan that specifies a data transfer process to save all the legacy knowledge in the new recovered database. The data migration solution is also introduced as a support for the OO conceptual schemas evolution where their persistent layers are stored by means of relational databases, in the ADAM tool. Contents and structure of the data migration plans are specified using an abstract data migration language. The high abstraction level of this language allows us to be independent from the underlying DBMS technology. Our past experience in both projects has guided us towards the model management research field. We present a case study that illustrates the application of both tools.

Keywords: data reverse engineering, rewriting rules, data migration, migration patterns
Categories: D.2.7 [Software Engineering]: Maintenance, D.2.9 [Software Engineering]: Management, E.2 [Data Storage Representations], H.1.0 [Models and principles]: General, H.2.4 [Database Management]: Systems, I.1.1 [Algebraic Manipulation]: Expressions and Their Representation

1 Introduction

Information systems are dynamic by nature. One reason an information system could change is the inaccuracy of its requirements specification. This inaccuracy is usually produced by a misunderstanding between the user and the system analyst, inexperience of the analyst or the imprecise knowledge of the user. Other reasons for this variable behavior could be changes in the requirements of a software application, adaptation to new technologies or even the satisfaction of new standards. The consequence of these facts is the continuous evolution of the system from the beginning of its deployment. This evolution is necessary in order to achieve the information system that the user wants. Other factors that produce continuous changes

[♦] This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

in information systems are the high level of competitiveness in the market place and their volatile business rules.
Statistics given regarding the time invested and the cost of people involved in the maintenance process are 80% of the total expense of software development [Yourdon 1996]. This fact has intensified interest in software evolution research in the past few years in order to cope with the problem and to reduce the costs.
A great deal of work has been done in this area. It has focused mainly on the automatic software development approach to improve the time and cost invested in the life cycle of the software system. Our work is based on the paradigm of automatic prototyping proposed by Balzer [Balzer 1985] (see Figure 1).

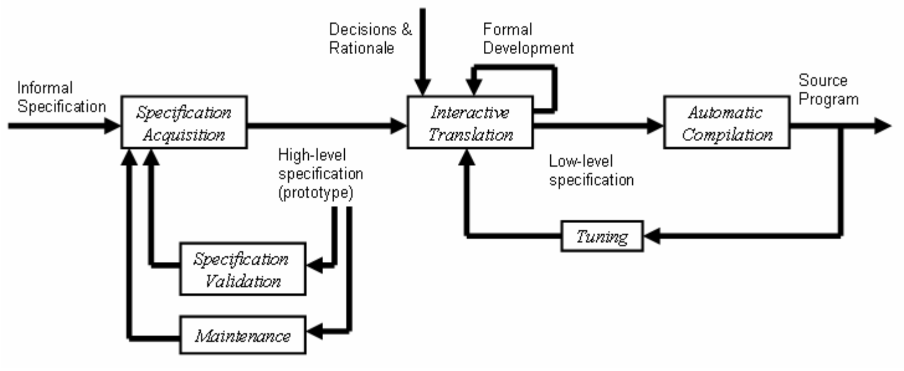


Figure 1: Paradigm of automatic prototyping

Existing CASE tools are able to generate applications from the systems specifications. These CASE tools are called model compilers. They use models to abstract a reality with some graphical notation and automatically generate the application code and the database schema from the conceptual schema of an information system. The automatic generation can be complete, such as Oblog Case ([Sernadas 1994]), or partial, such as Rational Rose ([Rational]), System Architect ([SystemArchitect]), Together ([TogetherSoft]) and others.
None of these tools provides full support for the volatile nature of an information system. Technologies used for software development become obsolete, when new technologies providing new and better features appear. However, software products are kept in use as long as they are useful and efficient, accumulating a lot of knowledge in their databases. Nevertheless, as they become obsolete they become more difficult to maintain. Systems of this kind are called legacy systems and their adaptation to new technologies or even the introduction of changes involves great economical efforts.
Another problem related to the dynamic behavior of an information system concerns to the addition of new requirements during the life cycle of the software product. Using one of the tools mentioned above, the change may be applied to the model and the new application and its database are automatically generated. Consequently, we have two conceptual schemas, the original schema and the evolved one, which contains the new features of the system. Additionally, we obtain two databases, the one corresponding to the first conceptual schema, which stores all the knowledge that

the original application has produced while it has been working, and the new, generated one, which remains empty. In this case, the problem solution focuses on finding a way to migrate the information produced by the first application to the new database.

In this paper, we present a solution for both model management problems. We explain how to solve them by means of two tools that use algebraic formalisms and pattern techniques: the legacy system recovery tool and the data migration tool.

In the next section, we present a motivating scenario involving both problems: a legacy system recovery and its evolution applying some changes. Section 3 presents the legacy system recovery tool and how it can resolve the first problem of the example shown in section 2. Section 4 presents the data migration tool and how it resolves the second problem of the problem. Section 5 discusses related works highlighting their differences with our tools. Finally, we present conclusions and future work.

2 Motivating Scenario

To promote the use of the two tools that we present in this paper, we will use a motivating scenario that is illustrated in Figure 2 and exemplifies the recovery of a legacy system and its later evolution.

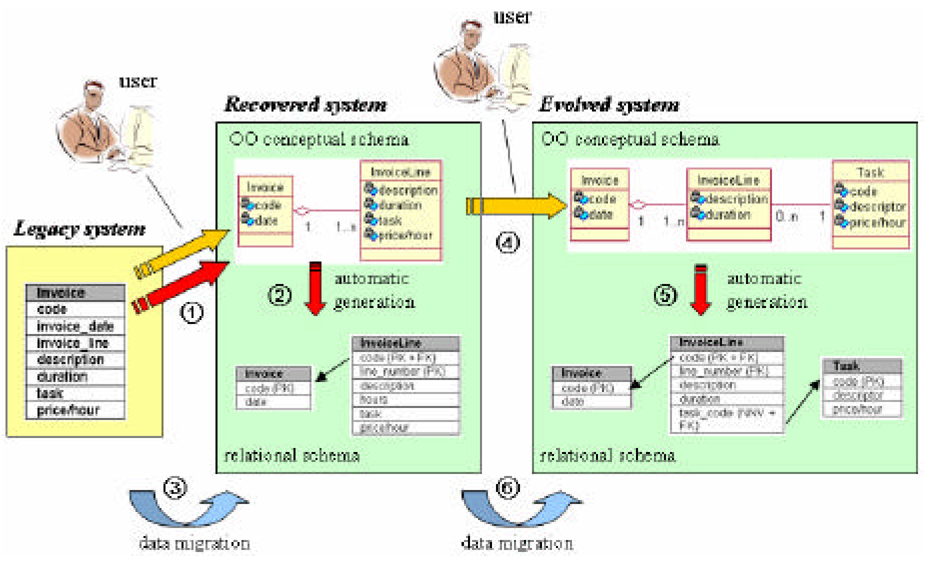


Figure 2: Motivating scenario

Consider a car maintenance company that has worked for a long time for a large car dealership. The maintenance company works with an old C application where the information is stored in a simple relational database that does not even consider integrity constraints. The car dealership has recently acquired the car maintenance company and its president has decided to migrate the old application to a new OO

technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language although the database layer remains a relational database. This time, new integrity constraints are provided in the new relational database in order to improve maintainability.
Consider the part of the legacy system that stores information about invoices in the example. Each invoice contains data about the task performed in a specific period of time and at a specific price.
To recover the legacy system, a designer has to build a semantically equivalent OO conceptual schema that captures the semantics that is disorganized in the legacy system. This task is usually done manually and involves high development costs. What is worse is that the human factor does not guarantee an error-free process to obtain a correct OO conceptual schema. Step (1) constitutes a manual, reverse-engineering process where the designer detects that the legacy table can be broken down into two classes: one containing the information about a performed task during a period of time and another one representing the collection of performed tasks for a specific customer, i.e., the *InvoiceLine* class and the *Invoice* class. In this step, works like those by [Hainaut 1996], [Premerlani 1994] and [Ramanathan 1996] can be applied to obtain OO models from relational schemas.
Once the OO model is complete, the relational database has to be generated (2). Here the designer can use many CASE tools such as Rational Rose, Together or System Architect in order to generate the new relational schema automatically. Despite obtaining a relational schema, these tools do not take into account legacy data.
The cumulative experience of the maintenance company is collected in its database and, it is expected to be preserved in the new database (3). Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user defined scripts which can be executed on the database. Although ETL tools provide friendly interfaces to migrate data between databases, DB administrators must write migration code manually, and this is very costly in terms of people and time.
Once the OO conceptual schema has been obtained from the legacy database and its data has been migrated to the new database, a design problem is detected in the generated OO schema: information about the same tasks is repeated and appears in several instances of the *InvoiceLine* class. Then, the designer decides to change the OO conceptual schema (4) breaking the *InvoiceLine* class down into two classes: a new class *Task* representing a task with information such as the price per hour and the *InvoiceLine* class that keeps the remaining information. The change is applied to the conceptual schema and a new empty database (5) is generated by means of the same CASE tool used in step (2). The data migration problem comes up again (6). Although ETL tools can be used in the same way as in step (3), this situation differs from the first case in the sense that the data migration process has to be specified. Now the original OO conceptual schema must be compared with the evolved one in order to obtain mappings between their respective databases. Thus, the designer that has applied changes to the original OO conceptual schema must supervise the data migration process in order to provide knowledge about the system, and it becomes a very complex solution.

In the following sections, we present two tools that provide a solution for both problems using formalisms and pattern techniques. This solution consists of an automated process that backs up the designer's work in an easy and efficient manner.

3 RELS: Reverse Engineering of Legacy Systems

Legacy systems can be defined informally as "software we do not know what to do with, but it is still performing a useful job" [Ward 1995]. They are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. They are characterized by the following features:

1. Software architecture based on obsolete technology that has probably been patched in order to adapt to new changes in requirements. This fact complicates the maintenance of the application.
2. Poor, complex documentation that prevents effective maintenance or software updating, making it necessary to check the source code to understand the functionality of the system.
3. Cumulative experience working with the system that has filled its database with information that is significant for the company.

As in all complex systems with a medium life cycle, the requirements for this kind of applications go on changing at the same rate as technology does. There are two main approaches to performing changes in these systems. On the one hand, there is the patching of the legacy system that has obsolete technology code. The disadvantages to this approach are that the technology does not consider new features to improve either code reuse, quality or documentation generation, and that the staff that will develop the new part of the system needs to be trained. On the other hand, the whole system can be developed with a new technology taking advantage of all its features. Both approaches imply a high cost, but we prefer the second option because the first one only temporarily delays the translation into a new technology, making maintenance harder and harder each time the system is changed.

The tool RELS (Reverse Engineering of Legacy Systems) provides a solution to this problem by applying the second approach to the structure of an application. It uses a reengineering process to rebuild the legacy system into a semantically equivalent one with a new technology. This process is composed of two steps:

1. A data reverse engineering process that extracts an abstract description from the legacy system database in order to know its structure and its behavior. Changes can be applied to it in order to adapt the systems to new requirements or to new technologies, such as the change between the structured paradigm and the OO paradigm. Our tool recovers a legacy database obtaining the static component of an equivalent OO conceptual schema using formal methods.
2. A forward engineering process that generates the software application (its structure in our case) based on a specific technology from the abstract description extracted from the legacy system. We use the Rose Data Modeler add-in [Boggs 2002] of the Rational Rose tool case to generate a new relational schema from the OO conceptual schema.

Our tool also allows for data migration from the legacy database to the recently generated one, keeping the knowledge stored in the old database. The data reverse engineering process and the data migration process followed by our approach will reduce the time invested and the number of people involved in the data evolution process. This optimization is reached by the automatic tasks that are performed by the tool in three phases. Despite the fact that these tasks are performed automatically, the results can be freely modified by the analyst. In this case, the process is semi-automatic. The tool performs the following three phases in order to reach this goal, as shown in Figure 3:

1. A UML conceptual schema is obtained by applying a data reverse engineering process in order to recover a relational legacy database. Both relational and UML conceptual schemas are represented as terms of an algebra and the correspondences between terms are specified using term rewriting rules.
2. The rewriting rules applied in the first phase and the patterns used by the Rose Data Modeler add-in to generate the new relational schema are used to describe a data migration plan which is specified using a declarative language.
3. The data migration plan is compiled into DTS[1] packages whose execution automatically migrates data from the legacy database to the new one.

In this section, we present the three phases in more detail, illustrating their application by means of the example of the motivating scenario.

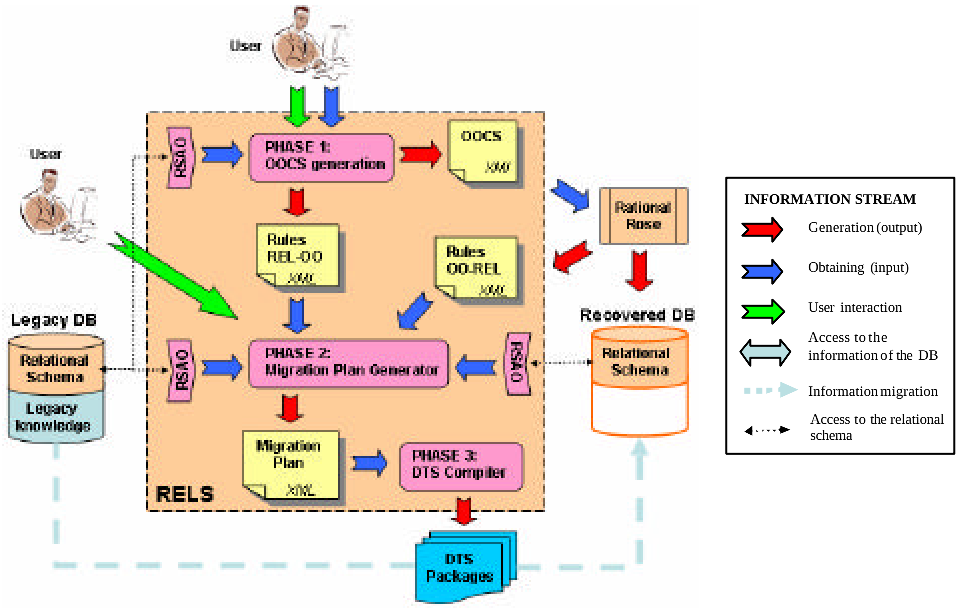


Figure 3: Legacy database recovery process in the RELS tool

[1] Data Transformation Services (DTS) is a technology of SQL Server that allows the transfer of data among heterogeneous databases.

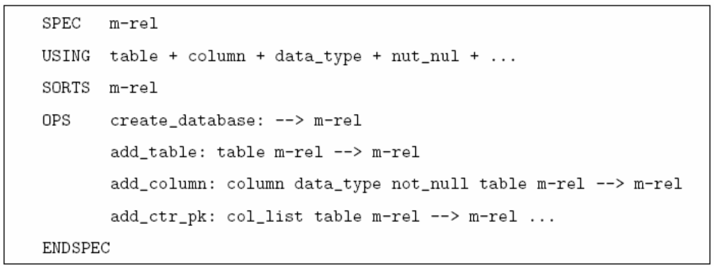
3.1 Data Reverse Engineering Phase

The data reverse engineering phase takes the relational model of the legacy database as input and generates an object-oriented model which is equivalent to the previous relational one. These models are represented as terms of ADTs (Abstract Data Types) that are related by means of rewriting rules. These rules are applied to the term that represents a relational schema by a TRS (Terms Rewriting System) which obtains an OO conceptual schema. The following sections describe the term rewriting mechanism that translates a conceptual schema from the relational model to the OO model, and the whole process that constitutes this phase.

3.1.2 Term Rewriting Mechanism

An ADT is composed by a group of specification modules. Each one of them provides a set of operations (constructors and functions) to define terms and axioms to establish relations among these terms, such as order properties. These axioms specify how an algebraic term is correctly defined.
Therefore, we define two ADT in order to represent both relational and OO conceptual schemas:

a) A relational conceptual schema is represented as an algebraic term that is based on the syntactic and semantic rules provided by the relational ADT. This ADT consists of several specification modules, each of which is related to a relevant element of the relational model. The relational model specification module (m-rel in Table 1) is the core module that provides rules to define a relational conceptual schema term. In this ADT, a constructor is used to define an element of the relational model as a term, for instance a table. Additionally, axioms are used to specify the natural composition order between elements of the relational model; for instance, tables are formed by columns, indicating a compositional relation between the specification modules of the relational ADT.



```
SPEC    m-rel
USING   table + column + data_type + nut_nul + ...
SORTS   m-rel
OPS     create_database: --> m-rel
        add_table: table m-rel --> m-rel
        add_column: column data_type not_null table m-rel --> m-rel
        add_ctr_pk: col_list table m-rel --> m-rel ...
ENDSPEC
```

Table 1: Relational specification module

b) The OO model specification module (m-oo in Table 2) is the core module of the OO ADT. It is composed of other modules that represent elements of the OO model, such as a class or an aggregation, and it expresses how to generate terms that combine the rules of its components. Thus, it provides the rules to generate OO conceptual schema terms.

```
SPEC    m-oo
USING   class + aggregation + specialization + ...
SORTS   m-oo
OPS     create_schema: --> m-oo
        add_class: class m-oo --> m-oo
        add_identif: attr_list class m-oo --> m-oo
        add_ctt_att: attribute data_type bool class m-oo --> m-oo
        add_vbl_att: attribute data_type bool class m-oo --> m-oo
        add_aggregation: aggregation class class rol rol nat nat
                         nat nat bool bool bool bool m-oo --> m-oo ...
ENDSPEC
```

Table 2: OO specification module

Once we can define conceptual schemas as terms of both relational and OO ADTs, we relate each element of the relational ADT with different elements of the OO ADT that are semantically equivalent.
An ADT consists of specification modules, and a module can be formed by other modules, as we have seen before in the cases of m-rel and m-oo modules. Hence, a new ADT is defined to relate elements of both relational and OO ADTs. The rules of the m-rel-oo relate the operators of the specific ADTs so that a term of the relational ADT is translated into a term of the OO ADT. These rewriting rules, which represent the correspondences between elements of the relational model and elements of the OO model, are automatically applied by a term rewriting system (TRS). Our TRS is finite and non-confluent, because we can obtain several OO terms from a relational term, i.e. several possible representations.
When the TRS applies the rewriting rules of the m-rel-oo specification module to a relational term, subterms of both relational and OO ADTs coexist in the intermediate terms that belong to m-rel-oo ADT. At the end of the rewriting process, the entire term belongs to the OO ADT. For further work, see [Pérez 2002c].
The rewriting process is automatic but the user can validate whether the applied rules are the most suitable, because an element of the relational model could be represented by several elements of the OO model. The tool supports decisions of this kind by providing the user with a set of potential rewriting rules that are syntactically correct in each rewriting step so that the user can choose one. In order to reduce the interactions required by the user, we have taken into account the criterion that legacy databases were usually designed to improve the efficiency of access to the data.

3.1.2 Data Reverse Engineering Process

The input of this phase is a relational schema of a legacy database and is the input to the main tool. This phase generates two XML documents as output: one representing the generated OO conceptual schema in XMI format ([OMG]) and another one that contains the rewriting rules applied to obtain the final OO conceptual schema. In this phase the process that produces these outputs follows three steps:

a) The reading of the relational schema of the legacy database. The access to the relational schema is performed by means of an API, named RSAO in Figure 3, in order to access heterogeneous databases. This phase builds a term of the defined relational ADT that represents the relational schema obtained from the legacy database. It also considers features of the old DBMS or other repository forms which do not allow for the definition of constraints (either integrity or reference constraints). These constraints, implemented by code in the best case scenario, are not explicitly defined in the legacy database structure. Thus, user interaction may be necessary to provide additional information to obtain a complete relational conceptual schema. This extra information is added to the relational term obtained from the relational database by means of a graphical interface that hides the algebraic formalisms to the user.
b) Translation of the relational term into an OO term by means of the rewriting rules described in the previous section. Here the user may decide to apply other rewriting rule than the default rules chosen by the TRS in order to generate a more semantically accurate OO term.
c) Storage of the OO term as an OO conceptual schema following the UML notation by means of the XMI format. Storing the conceptual schema with this format allows the import of the generated OO conceptual schema from most of the CASE tools that manage UML diagrams, such as Rational Rose. The applied rewriting rules in the translation process are written down to a XML document that will be used in the second phase.

3.2 Relational Migration Plan Generator

This phase generates a migration plan that specifies what information must be copied from the legacy database to the database of the new OO application. Its inputs are two XML documents that contain the mappings:

– between elements of the legacy relational schema and elements of the recently generated OO conceptual schema.
– between elements of this OO conceptual schema and elements of the new relational schema generated by the Rose Data Modeler add-in.

The migration plan generator applies a set of patterns to the input correspondences and produces a migration plan that is specified using a relational declarative language indicating what information has to be copied from the legacy database to the new one. The use of a declarative language provides independence from the specific DBMSs used for supporting the databases. Additionally, this phase checks the constraints of the target database in order to avoid constraint violation.

3.2.1. Relational Migration Plan

A relational migration plan specifies the actions that must be performed in order to copy data from the legacy database to the new database, generated from the recovered

OO conceptual schema. The migration plan consists of a set of *migration modules*. There exists one migration module for each specific table of the target database. Therefore, a migration module assigns a view over the legacy database to a target table indicating where to find the source data.
To specify the data copy process, a migration module contains a set of mappings between columns of the source view and the target table. Those mappings constitute the *migration expressions* that can be used in a migration plan and they are specified by means of the relational declarative language.
The automatic generation of the migration plan considers its structure and its contents. Thus, two kinds of patterns are used: migration patterns and migration expression patterns. Migration patterns generate the structure of the migration plan following the ordering of the tables of the new database. Migration expression patterns generate their content by means of migration expressions that represent mappings between columns of the relational tables.
The migration plan generator gets the applied rewriting rules of the reengineering process from the two input XML documents, one from the data reverse engineering phase and another one from the Rose Data Modeler. These rules provide enough information to determine how many migration modules are needed and which tables of the legacy database form the source view for each module. Thus, the generator constructs the migration modules by applying the migration patterns. Then, it applies the migration expression patterns in order to link attributes of the source view with attributes of the target table in each migration module, reflecting the generation process followed to obtain the target database. Once the migration plan is finished, the generator writes it in an XML document, ready to be compiled into a specific technology in order to perform the data migration.

3.2.2. Constraint checking

Referential and integrity constraints in the target database involve an added problem to the migration process because the legacy database is not supposed to support them. The migration plan generator checks these constraints in order to avoid errors during the data migration execution. To obtain the metainformation required about referential and integrity constraints of a database in order to generate the data migration plan, we focus on the standard SQL99 [Türker 2001]. Relational DBMS that are compliant with SQL99 provide a set of tables with each database that contains information about its relational schema. We have developed an API that accesses these tables in any DBMS by means of the OLEDB interface [Lee 2002]. This API is based on the ADOX object model [Sussman 1999] and provides a way to explore the metadata that forms a relational schema.
The migration order is a sequence in which the tables of the target database must be filled to avoid violations of any of its referential constraints. In the example in Figure 2, there is a foreign key of the *InvoiceLine* table to the Invoice table. If the migration process fills the *InvoiceLine* table first, the underlying referential constraint to the foreign key would be violated for all the copied tuples. The correct migration order for these tables is then the *Invoice* table first and the *InvoiceLine* table afterwards.
The generator of the migration plan considers a correct migration order by analyzing the relational schema of the database as if it were a directed pseudo-graph in which

the tables constitute the nodes and the foreign keys become the arcs. In this step, the generator considers foreign keys to the same table (loops), several foreign keys between two tables (parallel arcs) and cycles among several tables. This order becomes the sequence in which the modules of the migration plan must be performed (each module affects a table of the target database).
Additionally, the migration plan generator also considers the integrity constraints of the legacy and the target database, because, in the first phase, the analyst might complete the relational schema manually. Thus, the generated schema might contain some constraints that are not considered in the legacy database. A simple not null constraint over a column that is added to the new database can provoke an error if there is a tuple that has a null value for its source column in the legacy data.
The generator compares the relational schema of the source view and the target table for each migration module. When the tool detects an inconsistency, it proposes several solutions to the user, such as population filters, default values and data transformations.
Therefore, we obtain a migration plan that is not based on any specific DBMS but rather considers the constraints of the target database in order to perform a secure data migration.

3.3 Migration Plan compiler

This phase performs the compilation of the migration plan to a specific technology and its execution in order to carry out the physical data migration. Each DBMS has its own ETL tool that allows for data migration among databases. We use the Data Transformation Services (DTS) of Microsoft SQL Server. This tool allows data migration between heterogeneous relational DBMS by applying data transformation services in order to fulfill target database requirements. The DTS code that performs data migration is structured in DTS tasks. These tasks perform several actions such as data copy between two tables or the execution of SQL commands and the data connection to other databases. These tasks are stored in DTS packages that become the execution units that guide the data migration process.
The compiler receives an XML document that describes the migration plan from the second phase of RELS and obtains a set of DTS packages that are able to perform the specified migration plan between the legacy database and the new one. The compiler parses the migration plan, module by module, generating the structure of a DTS package. For each type of input module, there is a specific pattern that produces a set of DTS tasks. Once the structure of the final DTS packages has been built, the compiler parses the migration expressions of each migration module and generates the contents of the DTS tasks of the corresponding DTS module. These contents perform the connections to the databases and the migration process among the source tables and the target tables.
The migration plan avoids target database constraint violations by means of several solutions; one of them is to ignore inconsistent data. In this way, there may be a lot of legacy information that is not copied to the target database. The execution environment provides an option to migrate the inconsistent data to error tables that have no constraints so that the designer can query them and recover more information

by means of a wizard that applies the solutions of the second phase to these error tables.

add_ctr_pk([code, invoice_line], LegacyInvoice,	add_identif(line_number, InvoiceLine,
add_ctr_unique([code, invoice_line], LegacyInvoice,	add_identif(code, Invoice,
add_column(price/hour, currency, false,	add_unique(line_number, InvoiceLine,
LegacyInvoice,	add_unique(code, Invoice,
add_column(task, string, false, LegacyInvoice,	add_vbl_att(price_hour, currency, true,
add_column(duration, int, false, LegacyInvoice,	add_vbl_att(task, string, true,
add_column(description, string, false,	add_vbl_att(duration, int, true,
LegacyInvoice,	add_vbl_att(description, string, false,
add_column(invoice_line, int, false, LegacyInvoice,	add_ctt_att(line_number, int, true,
add_column(invoice_date, date, false,	add_vbl_att(invoice_date, date, true,
LegacyInvoice,	add_ctt_att(code, int, true,
add_column(code, int, false, LegacyInvoice,	(add_aggregation(agg_Invoice_InvoiceLine, Invoice,
add_table(LegacyInvoice,	InvoiceLine, 1, 1, 1, n, false, true, false, true,
create_database())))))))))))))	add_class(Invoice,
	add_class(InvoiceLine, create_schema())))))))))))))))

a) b)

Table 3.a): Relational term that represents the Invoice table of the legacy database.
Table 3.b): OO term that represents the two aggregated classes of the new OO conceptual schema.

3.4 Example

In the example of the motivating scenario, we start from a legacy system whose database consists of a table without any integrity constraint. RELS reads the legacy relational schema, and the designer adds metadata to the relational schema providing information about some integrity constraints. After that the tool generates the relational term that appears in Table 3.a representing the table of the legacy database plus information about the constraints. The rewriting process obtains the OO term that appears in Table 3.b. In this process, the designer has participated because the default rule that applies to a unique table generates a unique class. In this way, the user has chosen one of the rules proposed by the tool generating two aggregated classes.
The OO term that represents the OO conceptual schema is written in an XMI document so that any case tool that manages UML models could open the generated one. In Table 4, we show part of the generated document where there are two classes and the aggregation that relates them. We use Rational Rose to open the generated OO conceptual schema and its Data Modeler add-in to generate the corresponding relational schema. This new relational schema is different from the legacy one because the analyst has provided information about new integrity constraints and has also participated in the rule rewriting process.
Table 5 shows the migration module that specifies the data copy to the Invoice table of the target database. In this step, the tool has detected possible integrity constraint violations due to the not null value constraint over the columns of the table.

```
<UML:Association xmi.id='G.1' name='Invoice_InvoiceLine' visibility='public' isSpecification='false'
isRoot='false' isLeaf='false' isAbstract='false'>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id='G.2' name=''
        visibility='public' isSpecification='false' isNavigable='true' ordering='unordered'
        aggregation='none' targetScope='instance' changeability='changeable'
        type='S.363.1034.45.4'>
                ...
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id='G.3' name='' visibility='public' isSpecification='false'
        isNavigable='true' ordering='unordered' aggregation='aggregate' targetScope='instance'
        changeability='changeable' type='S.363.1034.45.1'>
                ...
        </UML:AssociationEnd>
    </UML:Association.connection>
</UML:Association>
<UML:Class xmi.id='S.363.1034.45.1' name='Invoice' visibility='public' isSpecification='false'
isRoot='true' isLeaf='true' isAbstract='false' isActive='false' namespace='G.0'>
        ...
</UML:Class>
<UML:Class xmi.id='S.363.1034.45.4' name='InvoiceLine' visibility='public' isSpecification='false'
isRoot='true' isLeaf='true' isAbstract='false' isActive='false' namespace='G.0'>
        ...
</UML:Class>
```

Table 4. XMI document that contains information about the generated OO conceptual schema.

Finally, the migration plan is compiled into DTS code by means of the third phase of the tool. Depending on the user's choices the compiler generates one package containing all the migration modules of the plan (compiled mode) or two DTS packages (step-by-step mode). These packages can be opened from the same SQL Server DTS tool. If the migration plan is compiled by means of the step-by-step mode, the DTS packages can be executed one by one, following the migration order specified in the migration plan. The graphical interface of the tool allows querying the data in order to check the migrated information, and it provides several wizards to recover inconsistent data from the error tables generated during the migration process.

4 ADAM: Automatic DAta Migration

Nowadays, CASE tools can evolve applications by modifying their conceptual schema and regenerating the code and the database schema from the modified conceptual schema (see Figure 2). However, these model compilers do not take into account the data stored in the database during the evolution process.
When an information system undergoes an evolution, its conceptual schema is updated and a new schema results. A model compiler generates a new code and a new empty database from the new schema. The structure and the properties of the new database could be different from the old database. As the data remains compliant to the old database schema, the designer must preserve the data of the company by correctly migrating it in order to satisfy the properties of the new database.

The migration task is necessary and is normally done by hand. This task considerably increases the maintenance cost of a software product. For this reason, an important issue is the improvement of the database maintenance process.

```
<migration_plan>
    ...
    <target_table operation="insert">
        <target_name operation="empty">INVOICE</name _destino>
        <target_pk> <pk_field>CODE</pk_field></target_pk>
        <source_table>
            <source_field>LEGACY_INVOICE</source_field>
            <source_pk><pk_field>CODE</pk_field></source_pk>
            <target_field operation="insert">
                <target_name operation="empty">CODE</target_name>
                <source_field>
                    <source_field>LEGACY_INVOICE.CODE</source_field>
                    <condition>UNIQUE</condition>
                    <condition>NOT_NULL_VALUE</condition>
                </source_field>
            </target_field>
            <target_field operation="insert">
                <target_name operation="empty">date</target_name>
                <source_field>
                    <source_field>LEGACY_INVOICE.DATE</source_field>
                </source_field>
            </target_field>
        </source_table>
    </target_table>
    ...
<migration_plan>
```

Table 5. Fragment of the relational migration plan that specifies the data migration to the table Invoice of the target database.

Our proposal is the ADAM tool. The starting point was the work done by Carsí about OASIS reflection [Carsí 1999]. OASIS is a formal language to define conceptual models of object-oriented information systems [Letelier 1998], and it was extended in Carsí's work to support the evolution of models. As a result, the AFTER tool [Carsí 1998] was developed. This is a CASE tool prototype which is based on the logic formalism *Transaction-Frame Logic* [Kifer 1995] and allows the definition, validation and evolution of OASIS models. As the data model of OASIS and UML are basically the same, the solution applied to OASIS models can be applied to UML models.
In ADAM, the data migration process transfers and updates information system data from the old database to the new one. It improves the data maintenance process due to the automatic tasks that constitute the three steps of its migration process (see Figure 4). Despite the fact that these steps are performed automatically, the results can be easily modified by the designer. In this last case, the process will be semi-automatic. These three steps are described in the following subsections.

4.1 Matching between Conceptual Schemas

This phase of the data migration process is necessary to be able to discover the changes that have occurred in the old conceptual schema in order to obtain the new one. The changes can be obtained applying a *matching algorithm* whose result is the set of correspondences between the old and new conceptual schemas. The matching is done automatically [Silva 2002a].

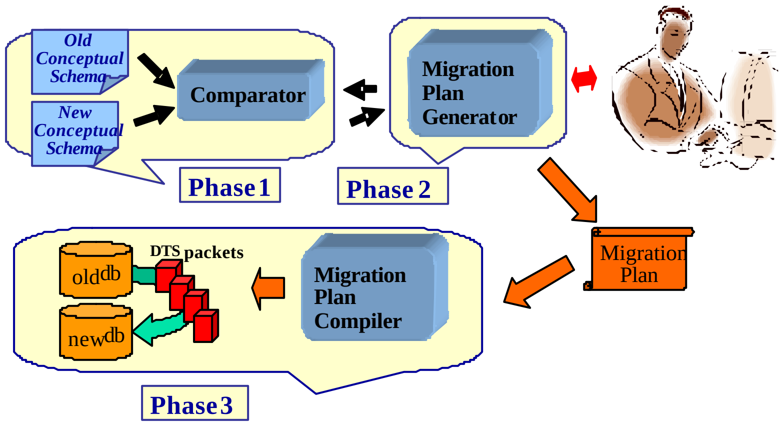


Figure 4. Data migration process of the ADAM tool

A great effort was done to study which kind of algorithms were most appropriate for this kind of problem and finally, a comparison algorithm for the ADAM tool was designed. This algorithm obtains the correspondences between both schemas and shows whether an element is a new one or a modification of an old one. The matching algorithmis based on dynamic programming techniques and graph theory.
In order to apply the algorithm, ADAM represents conceptual schemas as trees breaking cycles in relationships between their elements (see Figure 5). These elements are classified into the following categories: classes, aggregation and association relationships, specialization relationships and attributes. This categorization of the matching space reduces the complexity, thereby reducing the processing time of the algorithm.

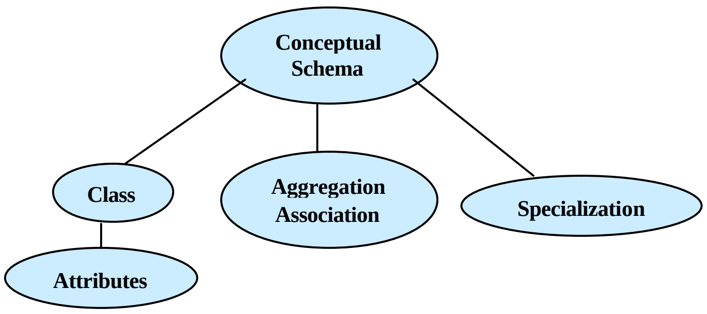


Figure 5. Tree representation of a conceptual schema

The algorithm can also use different *matching criteria* and different combinations of them [Silva 2002b]. The matching criterion allows us to distinguish whether two elements of two different conceptual schemas come from each other. Nowadays, the matching criteria that the algorithm applies are: the object identifier (OID), the name of the element, the number of attributes, the creation date, etc. Depending on the matching criterion selected by the user or the combination of them, the result of the algorithm is different. For this reason, the knowledge of the user about the system helps to select the most convenient criterion.
The matching of a sample of conceptual schemas provided by our industrial partners allowed us to validate the algorithm, achieving a high rate of correct matchings.
As a result of this phase, ADAM produces the correspondences between the elements of the conceptual schemas that have been compared. For instance, in Table 6, we show the correspondences of the algorithm using the *name matching criterion* for the classes of the example, presented in section 2.

OLD CONCEPTUAL	NEW CONCEPTUAL
Invoice	Invoice
InvoiceLine	InvoiceLine
Null	Task

Table 6: Correspondences between classes of the recovered and the evolving systems

4.2 Generation of a Data Migration Plan

From the correspondences detected in the first phase of the migration process, the second one automatically generates the first version of a *data migration plan*. A data migration plan is a set of data changes that are specified using "A Data Migration Language" (ADML) [Pérez 2002a]. Its execution allows the transfer of data from an old database to a new database one.
A data migration plan must include all the necessary changes to perform a correct migration in the right order. ADAM structures its data migration plans as a set of the following different elements: migration expressions, changes and modules; where migration modules have higher granularity than module expressions. The definition of these elements is as follows:

- *Migration expressions:* Migration expressions are those expressions that can be specified in a data migration plan. Each type of migration expression has different semantics and follows different syntactic constraints. Examples of migration expressions are: Data Source, Transformation Function, Filters, etc
- *Changes:* A change is the set of migration expressions that specify the updates undergone and the filters applied on the old database instances.
- *Migration modules:* A migration module contains the set of transformations applied to obtain a target element; it has a transactional behavior when it is executed by the ADAM tool. The composition of modules forms a data migration plan. Finally, it is important to note that there is one migration module for each class, aggregation and specialization of the new conceptual schema.

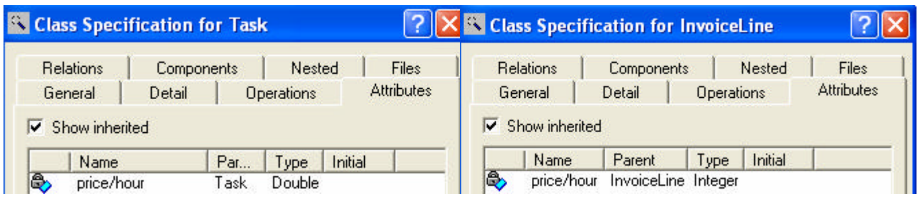
Each one of the changes that can be undergone in the old database is specified using a declarative language. This migration language follows the object-oriented metaphor. The main advantage of ADML is the independence from any DBMS due to its high abstraction level. For this reason, ADAM allows the expression of a migration plan in an easy and user friendly way and it does not need to take into account implementation details.
ADML is declarative and is used to specify all expressions that make up a data migration plan. Object-oriented conceptual schema elements are the data that are managed by the migration language. This language follows the object-oriented model. ADML makes use of path expressions to specify:

– The changes undergone in a conceptual schema element.
– The data belonging to the old conceptual schema.
– The filters that will be applied on the data, if necessary.

4.2.1 Inputs

The ADAM migration plan generation requires several data sources to automatically create the structure and the contents of the plan. The inputs of this process are the following:

– *Correspondences between conceptual schemas* produced by the first phase of the tool.
– *Properties of the elements of the conceptual schemas*. They are necessary in order to know the changes between the elements of a matching and to generate the implied transformations. These transformations must be applied on the data of the old element in order to be compliant with the new element. This information is in the conceptual schemas, where the properties of each element are defined. In our example, we are going to focus on the following correspondence:
  The *Price/hour* attribute of the *Task* class of the new conceptual schema obtains its data from the *Price/hour* attribute of the *InvoiceLine* class of the old conceptual schema. The properties of both attributes are included in the specification of the classes (see Figure 6).
  Figure 5 shows that the prices of the *InvoiceLine* class were in pesetas, and that they must be in euros in the new *Task* class. As a result, the data type of the *price/hour* attribute was *integer* in the old schema and is *double* in the new one. The *price/hour* value must also be converted to the equivalent in euros.



Show inherited ✓

Name	Par...	Type	Initial
price/hour	Task	Double	

Show inherited ✓

Name	Parent	Type	Initial
price/hour	InvoiceLine	Integer	

Figure 6. Price/Hour attribute of Task and InvoiceLine classes

– *Migration order*: The migration order is the sequence in which the data should be migrated. This order preserves the database in consistent states during the migration process. Moreover, this order facilitates the combination of migration modules as well as the migration order between non-related modules. The computation of this order is made by means of an algorithm. The migration order algorithm analyses the structure of the new conceptual schema to obtain the relationships between elements. These relationships imply dependencies determining the order to be followed in the data migration process. The migration order obtained by the algorithm for the new conceptual schema of our example is the following: *Invoice, Task, InvoiceLine.*

4.2.2 Process

The automatic generation of a data migration plan implies generating its structure and its contents. First of all, ADAM generates the structure creating an empty migration module for each element of the new conceptual schema and includes the modules in the data migration plan in the computed migration order. Next, the module content is automatically generated providing the migration expressions of each migration module and including them into their modules. Finally, a complete migration plan results. This automatic and complete generation of the migration plan is performed using two types of patterns: migration patterns and migration expression patterns [Pérez 2002b]. We have specified them using the patterns design criteria proposed by [Alexander 1979] and [Gamma 1994]. They are available in a pattern catalog, where they can be identified by a pattern number (**P-number**) and a title. Each pattern is composed of several sections that give different qualities of the pattern.

– *Migration expression patterns*

There is a pattern for each one of the element properties that can be changed by the schema evolution process and for any of their possible combinations. Each pattern produces a migration expression or a set of migration expressions specifying the correct transformation of data.

The generation of the migration expressions for a new element consists of determining which old element is related to it through mapping and consulting their different properties. Next, it applies the instantiated specific element pattern that specifies the migration expression code for the updated properties, and the resulting migration expressions are generated. Finally, these expressions are included in the new element module.

When the data migration plan is executed, the generated migration expressions of an element will be evaluated and the instances migrated to the new database. An example of a migration expression pattern is the one for an attribute when the “name” and the “data type” properties change (**P-08)** (see Table 7).

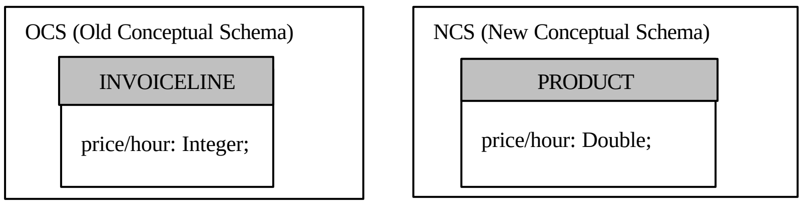
P-08: Pattern for an attribute when the “name”and the “data type” properties change.

Solution

The solution presents the generic migration expressions that specify the attribute changes of “name”, “data type” and “not null value” properties. In this case, as in the **P-04**[2] and **P-08**[3] patterns, it is necessary to perform a type conversion in the transformation function as follows:
old_data_typeTOnew_data_type (old_attribute)
This pattern is a composition of the “name” and the “data type” property patterns (**P-03**[4] and **P-04**). The migrations expressions that express these changes are the following:
Transformation_Function: generic_func‘(‘IDENT_clase‘.‘<IDENT_attr> ‘)‘

Example

The prices of the products were in pesetas, and now they must be in euros. As a result, the data type of the price/hour attribute was *integer* in the old schema and is *double* in the new schema and the price/hour value must be converted to €.



Text Format
Transformation_Function: IntToDouble(OldCS.InvoiceLine.price/hour)
XML Format
<Transformation_Function> IntToDouble(OldCS.InvoiceLine.price/hour)
</Transformation_Function>

Table.7: Solution and Example sections of the migration expression pattern P-08

In our example, ADAM uses the necessary patterns for each one of the established correspondences between the attributes of the new *Task* class and the old *InvoiceLine* class. Moreover, we need take into account that the transformation function generated by the pattern (IntToDouble(OldCS.Product.UnitPrice)) must be modified by the user in order to add the currency conversion function[5]. As a result, the transformation function that will be included in the data migration plan is PtsToEuro(IntToDouble(OldCS.Product.UnitPrice)).

– *Migration patterns*

[2] **P-04:** Pattern for an attribute when the “data type” property change.
[3] **P-08:** Pattern for an attribute when the “name” and the “data type” properties change.
[4] **P-03:** Pattern for an attribute when the “name” property change.
[5] The PtsToEuros conversion fnction is included in the ADAM set of built-in transformation functions.

For each type of target conceptual schema elements, migration patterns establish the way of migrating data. They also establish the necessary actions to migrate each type of conceptual schema element and the allowed migration expressions for each one. During the design process of a migration pattern, we must take into account the type of the conceptual schema element, because the transformations that may undergo each element are different.
A type of a conceptual schema element can have different associate patterns because there are different properties that influence the migration process. For example, the migration of a specialization relationship is different if its condition is more restrictive or less restrictive than the previous one, or is different because we must apply different types of filters on the data and different migration expressions in a different place. An example of a migration pattern is the pattern of the elemental class (**P-01)** (see Table 8):

P-01.Pattern: Elemental class

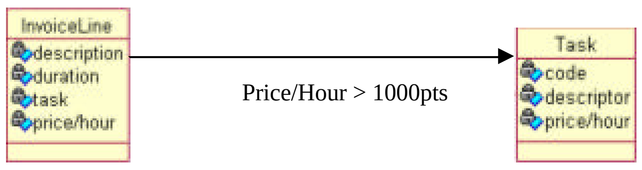
Solution

Let S be a set of schemas, C be an alphabet of classes, A be a set of attributes, G be a set of filters that are applied on old class population, GC be a set of conditions that are applied over old attributes, F be a set of transformation functions, SM be a set of matches between conceptual schemas, CM be a set of matches between classes of new and old conceptual schemas, and AM be a set of matches between attributes of conceptual schemas.

$$S1, S2 \in S \wedge S1.C1, S2.C2 \in C \wedge S1.C1.a1, S2.C2.a2 \in A \wedge f_1, .., f_n \in F \wedge g_1,...,g_n \in G \wedge gc_1,...,gc_n \in GC \wedge SM1 \in SM \wedge CM1 \in CM \wedge AM1 \in CA \wedge SM1.old=S1 \wedge SM1.new=S2 \wedge CM1.old=S1.C1 \wedge CM1.new=S2.C2 \wedge AM1.old=S1.C1.a1 \wedge AM1.new=S2.C2.a2 \rightarrow data(S2.C2) = \{y \mid \exists x \in data(S1.C1) \wedge \forall i\ ?_x\ g_i\ i=1,..,n \wedge ((y.a2 = f_n o f_{n-1}...o f_1(x.a1) \vee y.a2 = cte) \wedge \forall i\ ?_{x\downarrow a1}\ gc_i)\}$$

Example

The Task class of the new conceptual schema obtains its data from the InvoiceLine class of the old conceptual schema. However, the analyst of this system is only interested in the products that have a price that is higher than 1000. Moreover, all its attributes must be migrated with their transformations and conditions.

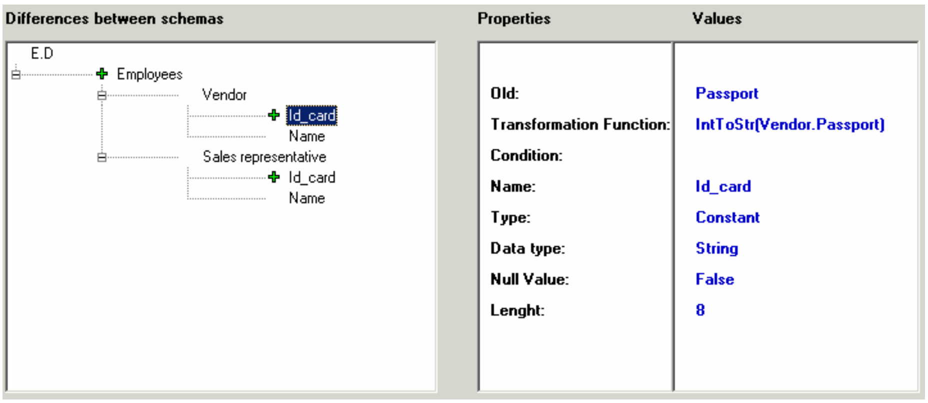


Text Format:
S1, S2 $\in$ S $\wedge$ S1.InvoiceLine, S2.Task $\in$ C $\wedge$ S1.InvoiceLine.task, S1.InvoiceLine.price/hour, S2.Task.code, S2.Task.descriptor, S2.Task.price/hour $\in$ A $\wedge$ IntTODouble, RightTrunc, PtsToEuro $\in$ F $\wedge$ {S1.InvoiceLine.price/hour > 1000pts}$\in$ G $\wedge$ SM1 $\in$ SM $\wedge$ CM1 $\in$ CM $\wedge$ AM1, AM2, AM3.AM4 $\in$ CA $\wedge$ SM1.old=S1 $\wedge$ SM1.new=S2 $\wedge$ CM1.old=S1.InvoiceLine $\wedge$ CM1.new=S2.Task $\wedge$ AM1.old=S1.InvoiceLine.task $\wedge$ AM1.new=S2.Task.code $\wedge$ AM2.old=S1.InvoiceLine.price/hour $\wedge$ AM2.new=S2.Task.price/hour $\wedge$ AM3.new=S2.Task.descriptor $\rightarrow$ data (S2.Task) = { y | $\exists$x $\in$ data(S1.InvoiceLine) $?_x$ (S1.InvoiceLine.price/hour > 1000 $\wedge$ (y.code = x.task) $\wedge$ (y.descriptor = " ") $\wedge$ y.Price = PtsToEuro(IntToDouble(x.UnitPrice)) }

XML Format:
```
<New_Conceptual_Schema>
 <Class>
  <Name> Task </Name>
  <Origin>
   <Name> InvoicedLine </Name>
   <Filtered>
    <Filter> OldCS.CS1.InvoicedLine.price/hour > 1000
    </Filter
    <Attribute>
     <Name> code </Name>
     <OriginAttribute> OldCS.InvoiceLine.Task </OriginAttribute>
     <Transformation_Function> OldCS. InvoiceLine.Task
     </Transformation_Function>
    </Attribute>
    <Attribute>
     <Name> Descriptor </Name>
     <OriginAttribute> Null </OriginAttribute>
     <Transformation_Function> " "
     </Transformation_Function>
    </Attribute>
     <Attribute>
     <Name> price/hour </Name>
     <OriginAttribute> OldCS.InvoiceLine.price/hour </OriginAttribute>
     <Transformation_Function>
            PtsToEuro(IntToDouble(OldCS.InvoiceLine.price/hour))
     </Transformation_Function>
    </Attribute>
  </Class>
</New_ConceptualSchema>
```

Table 8: Solution and Example sections of the migration pattern P-01

The first version of the data migration plan should be validated by the user after it is generated by ADAM. In addition, users can modify the plan if they want. ADAM provides a graphical user interface in order to perform these tasks in an easy, user-friendly way. This interface shows the correspondences between elements using a tree and the differences between them by means of textual expressions, symbols and colors (see Figure 7).



Differences between schemas

```
E.D
└─ ✚ Employees
     ├─ Vendor
     │    ├─ ✚ Id_card
     │    └─ Name
     └─ Sales representative
          ├─ ✚ Id_card
          └─ Name
```

Properties	Values
Old:	Passport
Transformation Function:	IntToStr(Vendor.Passport)
Condition:	
Name:	Id_card
Type:	Constant
Data type:	String
Null Value:	False
Lenght:	8

Figure 7. Graphical representation of the differences between elements

4.2.3 Outputs

After applying the necessary patterns to automatically generate the data migration plan, it is written in an XML document. This format makes the reading and translation of the data migration plan easier. This document makes the second and third phases of ADAM independent from each other.

4.3 Data Migration Plan Compiler

Finally, the third phase of the migration process compiles the data migration plan into code. The code execution migrates data from the old database to the new one [Anaya 2003]. This third phase of the ADAM tool automatically generates the code that a migration tool must manually produce using its script languages. In this way, ADAM considerably reduces the people and time invested in the creation of a migration plan between databases.

In ADAM, the target language was selected taking into account the capability of specifying complex expressions and the possibility of migrating data between heterogeneous databases such as Oracle [Oracle], Access, dBase, SQL Server, etc. SQL was excluded because it does not provide enough expressivity to specify complex expression transformations.

The compilation of the data migration plan produces a set of DTS packages. A DTS package includes a set of connections to the data sources, where data are read and stored, and a set of tasks to migrate the information. To generate the specific DTS packages that perform the data migration, we define a set of semantic correspondences between the object-oriented migration plan and the elements of a DTS package. These correspondences are shown in table 9.

Having taken into account these correspondences, the compiler reads the migration plan and the conceptual schemas to extract the necessary information to automatically generate the DTS packages. This information is stored into intermediate structures of the main memory to be able to query them during the compilation process. The result of this process is a set of DTS packages, whose execution performs the migration of data.

Data Migration Plan	**DTS Code**
Migration Module	Package
Migration Sub module	Task
Data Filter	WHERE condition of task query
Transformation Function	Function specified using the script language and defined at the transformation section of a task
Attribute Condition	Condition specified using the script language and defined at the transformation section of a task

Table 9: Solution and Example sections of the migration pattern P-01

5 Experimental results

Results of both experiences in software dynamics are two tools that involve breakthroughs in legacy system recovery and in data migration. In this section, we indicate how we tested both tools.

5.1 Experimental results in the RELS tool

The RELS tool consists of several modules that are communicated by means of XML documents. This modularity has allowed us to use technologies that run on different operating systems, such as MAUDE system, which runs on Linux OS, and DTS, which runs on Windows OS. As Figure 3 shows, the RSAO API reads the metainformation that constitutes the relational schema of the legacy database and structures this information into an XML document. This document is read by the translation module (phase 1), which uses the MAUDE system and produces two more XML documents: one describing the generated OO model in XMI, and another one specifying the rules applied during the rewriting process.
The XMI document is used by Rational Rose to obtain the OO conceptual model in order to generate the relational schema of the target database. The migration plan generator module (phase 2) obtains the XML document that describes the rewriting process followed in phase 1, and obtains the generation rules applied by the Data Modeler add-in of the Rational Rose tool. This module (phase 2) generates an XML document that specifies the data migration plan, which is compiled by the DTS compiler module (phase 3), obtaining the DTS packages, whose execution performs the data migration from the legacy database to the target one.
One of the tests that we applied to the RELS tool was a free accounting application, which stored its information in a relational database. We used RELS to recover this database, obtaining an OO conceptual schema that could be edited in Rational Rose and a new relational database that contained the information of the legacy database. This process was carried out in an almost automatic manner. The user only interacted with RELS to indicate that the table could be broken down into several classes. By

doing so, the RELS tool saved us from having to use a developer team to build the new database and to migrate the information, decreasing costs in both staff and time.

5.2 Experimental results in the ADAM tool

The ADAM tool has a 3-tier architecture: client, server and database. The client layer includes the interface of the ADAM tool. The server layer implements services that provide ADAM in order to manage the data migration process. Finally, the database stores the information about schemas, the matchings between them and data migration plans.
Moreover, ADAM needs a checker of ADML migration expressions in order to syntactically and semantically validate the migration expressions defined by users. The ActiveX checker has been generated using VisualParse ++, and a file of rules has been designed.
The checker is invoked by the server layer using the function *fu_validate(string_formulae, type_formulae)*. Each time that the checker receives an invocation of *fu_validate*, it will reply to the server indicating wether the migration expression is valid and providing the decomposition of the migration expression in a XML syntactic tree. In addition, the checker needs information about the conceptual schema elements that includes the migration expression during its validation process. This information is achieved by querying the server.

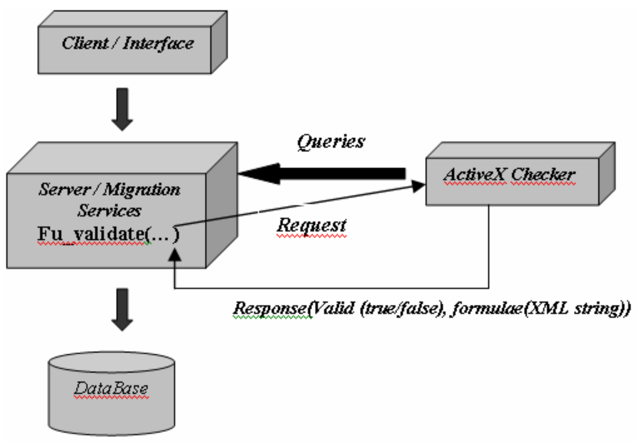

Client / Interface
Queries
Server / Migration Services
Fu_validate(...)
ActiveX Checker
Request
Response(Valid (true/false), formulae(XML string))
DataBase

Figure 8. ADAM architecture

With regard to the server layer, it implements the three phases needed to create a data migration plan (see Figure 4). Input conceptual schemas are XMI documents that are loaded in the database of the ADAM tool. This information is used by a module that implements the comparison algorithm, and the results of the algorithm are stored in the database. Another module implements the generation of the data migration plan. This one uses the data stored in the database and its results are stored in a XML document, which follows a DTD that was designed for this purpose. The structure of these XML documents is briefly presented in the migration patterns (see table 8). Finally, the XML document that stores the data migration plan is used by the last

module that implements the server layer of the tool. This module is the one responsible for compiling this XML document into DTS packages in order to be executed. This execution allows the data migration from the old database to the new one. It is important to keep in mind that XMI documents allow us to manage any kind of conceptual schema that is able to be stored in accordance with this standard, and XML documents allows us to achieve independence among the phases of the ADAM tool.
ADAM tool was tested using several examples. These examples were provided by industrial partners and were used in order to evolve their data. As a result of these experiments, we obtained better results than our industrial partners doing these tasks by hand. Due to the privacy policies of the companies, we cannot publish these results; however, we can advance that one of our analyst who was not familiar with the information system, migrated the database using ADAM for two days and half; whereas the same database had been manually migrated by a databases expert and an expert familiar with the information system for one month.

6 Related work

The work presented in this paper involves data reverse engineering from relational schemas to OO schemas (using a relational persistent layer) and data migration between relational schemas and between relational and OO schemas.
In [Bisbal 1999], a general migration process is broken down into several phases. We compare RELS and ADAM and their application with the tools studied in this survey for each proposed phase. In [Bisbal 1999], the justification phase is when the benefits and risks of recovering a legacy system are discussed. Although there are software quality metrics for estimating the level of technical difficulty involved and there are tools like RENAISSANCE [ESPRIT] to support this task, we have not yet considered them.
An understanding process of the legacy system is necessary in order to know its functionality and how it interacts with its domain. [Müller 2000] presents a roadmap for reveerse engineering research building on the program comprehension theories of the 1980s and the reverse engineering technology of the 1990s. We focus on database understanding tools and we find the DB-Main CASE tool [DB-Main]. This tool applies a data reverse engineering process and recovers the conceptual schema from the logic schema to obtain traceability between different layers of the database, to create new databases in other DBMSs and to reduce the dependence on the technology. [Henrard 2002] describes and analyzes a serie of strategies to migrate data-intensive applications from a legacy data management system to a modern one, basing on DB-Main CASE tool. Rational Rose also obtains OO conceptual schemas from many DBMSs by means of the Data Modeler add-in. However, none of them takes into account legacy data recovery.
RELS does not only the support development of relational schemas by hiding their physical database design, but it also provides an automated translation across ontologies, i.e. the relational and the OO metamodels. In the target system development phase, which is based on three-layer target systems, we produce the persistent layer.

The testing phase ensures that the new recovered system provides the same functionality as the legacy system. This is a complex task that can be supported by a Back-to-Back testing process [Sommerville 1995]. We can shorten this task by generating a relational schema that is semantically equivalent to the legacy database schema, although we allow the introduction of new integrity constraints to the legacy relational schema that remains buried in the code of the old application.

For the migration phase, we contrast the use of our tool with the other approaches to migration presented in [Bisbal 1999]. The Big Bang approach [Bateman 1994], also referred to as the Cold Turkey Strategy [Brodie 1993], involves redeveloping a legacy system from scratch using modern software techniques and hardware of the target environment. This approach was severely criticized in [Brodie 1993]. In [Brodie 1995], Brodie and Stonebraker present their Chicken Little approach. This strategy proposes migration solutions for fully-, semi- and non-decomposable legacy systems by using a set of gateways that allow the recovery of the legacy system in an incremental way. These gateways relate the legacy and recovered databases during the migration process, so that both systems coexist during the migration process, sharing data. Nevertheless, [Wu 1997] presents the Butterfly methodology, which discredits the Chicken Little approach by arguing that the migration process maintained by means of gateways is too complex. Under the Butterfly approach, new subsystems are developed; however they are only taken into production once the whole system is finalized using the Cold Turkey approach. The last phase involves a data migration process eliminating the need for data gateways. RELS follows this last approach providing support for an automated generation of the new database by means of a formal data reverse engineering process. This reduces the fears that a legacy system migration provokes.

RELS and ADAM focuses on the data migration process taking into account heterogeneous relational DBMSs and managing inconsistencies that might be produced during the migration of the legacy data to the new database. Additionally, the high level of user involvement in these approaches is drastically reduced by means of data inconsistency wizards and automated support for schema generation. RELS focuses on relational DBMS, but it can also be applied to COBOL legacy systems whose persistent layer is based on a flat file by interpreting it as a relational table.

Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done by means of SQL statements or user-defined scripts that can be executed on the database. However, these tools do not provide automatic support for the generation of these statements and scripts as the data migration tool does. For this reason, DB administrators must write the migration code by hand.

There are several works that study new algorithms to perform data migration in a more efficient manner, such as [Anderson 2001] and [Khuller 2003]. These works focus on the physical consistency of the data persistence when the physical storage configuration must be changed, whereas we stay at a high logical level.

The most similar approach to ADAM on data migration between relational and OO schemas is the TESS tool [Staund 2000]. It involves an automatic process that is based on schema evolution. TESS uses an intermediate language that is generated from the relational schema code. This is an important difference to our approach, because we deal directly with the OO conceptual schemas, and we do not have to

translate them to an intermediate language. The OO conceptual schemas give us a higher level of abstraction and eliminate the translation process.
The Varlet Database [Jahnke 1998] provides support to transform a relational schema into an OO conceptual schema and migrates the legacy data to the new OO database. However, our approach considers a relational database as the persistence layer of an object society and migrates information to it. Also, in Varlet, the legacy relational schema is enriched with semantic information that is extracted from several sources as the application source code. In our approach, this semantic information is given by the user in an interactive way.

7 Final Remarks and Further Work

This paper presents two experiences in software evolution that provide support to legacy system recovery and data migration.
To recover a legacy system, we follow an algebraic approach by using algebra terms to represent models. RELS provides a data reverse engineering process supported by a Term Rewriting System that applies a set of rewriting rules, obtaining the term that represents the target OO model. RELS also generates a data migration plan that specifies the data copy process to keep all the legacy knowledge in the new recovered application database. This entire process should be checked by a designer who could intervene, if necessary in order to obtain a more accurate result.
The data migration problem is also introduced for the OO conceptual schemas evolution where persistent layers are formed by relational databases. In this case, a matching process is applied between both OO models to generate mappings between them that are used in the generation of the data migration plan. The automatic generation process gives us a first version of a data migration plan that can be modified later by the designer.
The contents and structure of the data migration plan are generated by means of a set of patterns. The high abstraction level of the migration language allows us to be independent from the underlying DBMS.
Our implementation experience in both projects has led us to dealing with generic model management tasks. A model is an abstract representation of reality that enables communication among heterogeneous stakeholders in order to understand each other, i.e. an OO conceptual schema, an interface definition, an XML DTD, or a semantic network.
RELS and ADAM work for several heterogeneous models by means of mappings between them that allows transformations between models of heterogeneous metamodels such as the automatic generation of OO conceptual schemas from relational schemas. Model management aims at solving problems related to model representation and its manipulation. This is done by considering models as first-class citizens that are manipulated by means of abstract operators. This approach permits the automation of model manipulation tasks. Therefore, it completely involves all the tasks carried out in our projects. In future projects, we will propose a model management platform that permits model representation and manipulation using an algebraic approach.

8 References

[Anderson 2001] Anderson E., Hall J., Hartline J., Hobbes M., Karlin A., Saia J., Swaminathan R., Wilkes J.: “An Experimental Study of Data Migration Algorithms”. Workshop on Algorithm Engineering, pages 145-158, 2001.

[Alexander 1979] Christopher Alexander, “The Timeless Way of Building”, Oxford University Press. 1979.

[Anaya 2003] Anaya V.**,** Carsí J.A., Ramos I., “Automatic evolution of database data”, Novática, ISSN: 0211-2124, July 2003 (in Spanish).

[Balzer 1985] Balzer R., “A 15 Year Perspective on Automatic Programming”. IEEE Transactions on Software Engineering, vol.11, num.11, pages 1257-1268, November 1985.

[Bateman 1994] A. Bateman and J. Murphy, "Migration of Legacy Systems"; School of Computer Applications, Dublin City University, Working Paper CA-2894, 1994

[Bisbal 1999] Bisbal, J., Lawless, D., Wu, B., Grimson, J.: “Legacy Information Systems: Issues and Directions”, IEEE Software, September/October 1999.

[Boggs 2002] Boggs, W., Boggs, M.: “Mastering UML with Rational Rose 2002”; Sybex. January 2002.

[Brodie 1993] Brodie M., Stonebraker M., “DARWIN: On the Incremental Migration of Legacy Information Systems”; Technical Report TR-022-10-92-165 GTE Labs Inc., March 1993

[Brodie 1995] Brodie, M. and Stonebraker, M., "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach", Morgan Kaufmann, 1995.

[Carsí 1998] Carsí J.A., Camilleri S., Canós J.H., Ramos I., “Homogeneous graphical user interface to design and use information systems”, JIS’98, III Workshop on Software Engineering, Murcia - Spain, November 1998.

[Carsí 1999] Carsí J.A., “OASIS as conceptual framework to treat the software evolution”, PhD Thesis, Technical University of Valencia, Spain, November 1999 (in Spanish).

[Chaffin 2000] Chaffin M., Knight B., Robinson T., “Professional SQL Server 2000 DTS (Data Transformation Services)”, Wrox, 2000

[DB-Main] DB-Main, http://www.fundp.ac.be/recherche/unites/publications/en/2987.html

[ESPRIT] ESPRIT Project - Lancaster University, “RENAISSANCE Project - Methods & Tools for the evolution and reengineering of legacy systems”; http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenaissanceWeb/, December 1997

[Gamma 1994] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addyson-Wesley.1994.

[Hainaut 1996] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., Hick, J.M.: “Structure Elicitation in Database Reverse Engineering”; WCRE’96 3 (1996), 131-140.

[Henrard 2002] Henrard J., Hick J.M., Thiran P., Hainaut J.L.: “Strategies for Data Reengineering”; WCRE 2002, Working Conference on Reverse Engineering October 29 - November 1, 2002. Richmond, Virginia, USA.

[Jahnke 1998] Jahnke, J.H., Zundorf, A.: “Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment”. Theory and Application of Graph Transformations (TAGT’98), Paderborn, Germany, Technical Report tr-ri-98-201, University of Paderborn, 1998.

[Khuller 2003] Khuller S., Kim Y., Wan Y.: “Algorithms for Data Migration with Cloning”; SIAM Journal on Computing, Volume 33, Number 2, Pages 448 – 461, 2003.

[Kifer 1995] Kifer M., “Deductive and Object Data Languages: A Quest for Integration,” Proc. of the 4th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995.

[Lee 2002] Lee W.: “The Evolution of Data-Access Technologies”. SQL Server Magazine (April 2002). URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsqlmag02/html/TheEvolutionofData-AccessTechnologies.asp

[Letelier 1998] Letelier P., Sánchez P., Ramos I., Pastor O., “OASIS 3.0: A formal Approach for OO Conceptual Modelling". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998. (In Spanish)

[Müller 2000] Müller H. A., Jahnke J. H., Smith D. B., et al.: “Reverse Engineering: A Roadmap”; In A. Finkelstein, editor, The Future of Software Engineering, ACM Press, 2000.

[OMG 2003] OMG: XML Metadata Interchange (XMI), v2.0. Object Management Group, formal/03-05-02 (2003)

[Oracle] Oracle Corporation, *Oracle*, http://www.oracle.com

[Pérez 2002a] Pérez J., Carsí J A., Ramos I., “ADML: A Language for Automatic Generation of Migration Plans”, The First Eurasian Conference on Advances in Information and Communication Technology, Tehran, Iran, octubre 2002 http://www.eurasia-ict.org/ © Springer LNCS vol n.2510

[Pérez 2002b] Pérez J., Carsí J A., Ramos I., “On the implication of application’s requirements changes in the persistence layer: an automatic approach”, Database Maintenance and Reengineering Workshop (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal (Canada), October 1st, 2002, pag. 3-16, ISBN: 84-699-8920-0.

[Pérez 2002c] Pérez, J., Ramos, I., Anaya, V., Cubel, J.M., Domínguez, F., Boronat, A., Carsí, J.A.: “Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas”; Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle Workshop (SET'02), First International Conference on Graph Transformation(ICGT2002) , Barcelona (Spain), 2002.

[Premerlani 1994] Premerlani, W.J., Blaha, M.: “An approach for reverse engineering of relational databases”; Communications of the ACM 37,5 (1994), 42-49.

[Ramanathan 1996] Ramanathan, S., Hodges, J.: “Reverse Engineering Relational Schemas to Object-Oriented Schemas”; Technical Report MSU-960701, Mississippi State University, Mississippi, 1996.

[Rational] Rational Software, “Rational Rose”, http://www.rational.com/products/rose/.

[Sernadas 1994] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994

[Silva 2002a] Silva J.F., Carsí J.A., Ramos I., “An algorithm to compare OO-Conceptual Schemas” Proceeding of the International Conference on Software Maintenance (ICSM’2002), Montreal, Canada, 2002.

[Silva 2002b] Silva, J.F., Carsí, J.A., Ramos, I., “Theoric analyze of the criteria of OO conceptual schemas comparison”, Ingeniería Informática Magazine, ISSN:0717-4197, January, http://www.inf.udec.cl/revista/edicion7/jsilva.htm (in Spanish)

[SQL] Microsoft, *SQL Server*, http://www.microsoft.com/sql

[Sommerville 1995] Sommerville, I., “Software Engineering”, Addison-Wesley, 1995

[SystemArchitect] “System Architect”, http://www.popkin.com/products/sa2001/systemarchitect.htm

[Staund 2000] Staund Lerner, B.: “A Model for Compound Type Changes Encountered in Schema Evolution”; ACM Transactions on Database Systems (TODS) March 2000, Vol. 25 number 1.

[Sussman 1999] Sussman, D.: “ADO 2.1 Programmer's Reference”; Wrox Press Inc; 2nd edition (June 1999).

[TogetherSoft] “TogetherSoft Corporation”, http://www.togethersoft.com/

[Türker 2001] Türker, C., Gertz, M.: “Semantic Integrity Support in SQL-99 and Commercial (Object-)Relational Database Management Systems”. The VLDB Journal — The International Journal on Very Large Data Bases archive. Volume 10 , Issue 4 (December 2001). Pages: 241 - 269.

[Versant] Versant Object Technology, “Versant”, http://www.versant.com/

[Ward 1995] Ward, M.P., Bennett, K.H.: “Formal Methods to Aid the Evolution of Software”; Journal of Software Maintenance: Research and Practice, Vol 7, no 3, May-June 1995, pp 203-219.

[Wu 1997] Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., O'Sullivan, D.: "The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems", in Proceedings of the 3rd 1EEE Conference on Engineering of Complex Computer Systems, Italy, September 1997.

[Yourdon 1996] Yourdon, E.: “Rise and Resurrection of the American Programmer”; Yourdon Press, Upper Saddle River, NJ, 318 pp., 1996.