Journal of Universal Computer Science, vol. 10, no. 4 (2004), 428-453
submitted: 17/10/03, accepted: 2/2/04, appeared: 28/4/04 © J.UCS

Two Experiences in Software Dynamics

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

Jennifer Pérez
(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 that provides breakthroughs for legacy systems recovery (RELS) and for data migration (ADAM). To recover a legacy system, we use an algebraic approach by using algebras in order to represent the models and manipulate them. RELS also generates automatically 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 O-O 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. Our past experience in both projects has guided us towards the model management research field. We present a case study to illustrate our proposal.

Key Words: data reverse engineering, rewriting rules, data migration, migration patterns
Categories: D.2.7, D.2.9, E.2, H.1.0, H.2.4

1 Introduction

Information systems are inherently dynamic. One reason for an information system to change is the inaccuracy of its requirements specification. This inaccuracy is usually caused by misunderstandings between the user and the system analyst, inexperience of the analyst or the imprecise knowledge of the user. Other reasons for variability are changes in the requirements of a software application, adaptation to new technologies, the satisfaction of new standards, 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 over the past few years in order to cope with the problem and to reduce the costs. A great deal of

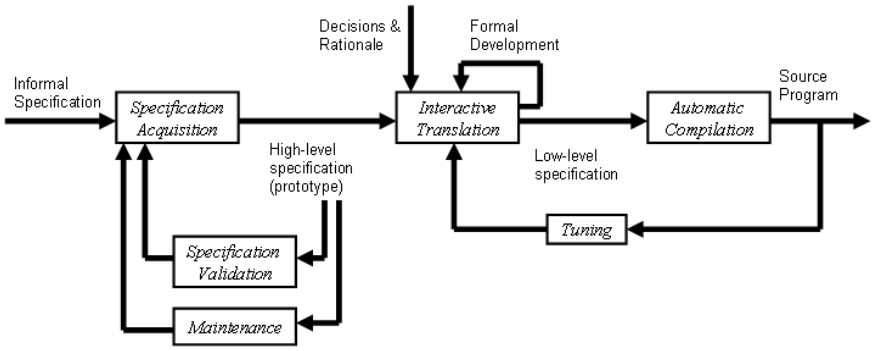


Figure 1: Paradigm of automatic prototyping

work has been done in this area, and it has focused mainly on the automatic software development approach to improve the time and cost invested in the life cycle of a software system. Our work is based on the paradigm of automatic prototyping proposed in [Balzer, 1985] [see Fig. 1].

Existing CASE tools are able to generate applications from specifications. They are commonly referred to as model compilers, and they use visual models from which the application code and the database schema can be generated automatically from the conceptual schema of a system. The automatic generation can be complete, e.g., Oblog Case [Sernadas, 1994], or partial, e.g., Rational Rose [Rational], System Architect [SystemArchitect], Together [TogetherSoft]. However, 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, and gather much knowledge in their databases. Nevertheless, as they become obsolete they become more difficult to maintain. Such systems are called legacy systems and their adaptation to new technologies or even the introduction of changes involves much effort.

Another problem related to the dynamic behavior of an information system concerns the addition of new requirements during the life cycle. Using one of the above-mentioned tools, the change may be applied to the model and the new application and its database are regenerated automatically. Consequently, we have two conceptual schemas, i.e., the original and the evolved one, and 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 into 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. The rest is organized as follows: first, we present a motivating scenario involving both problems: a legacy system recovery and its evolution [see Section 2]; we then present

the legacy system recovery tool [see Section 3] and the data migration tool [see Section 4], along with some experimental results [see Section 5]; we report on the related work [see Section 6], and present some conclusions and future work directions [see Section 7].

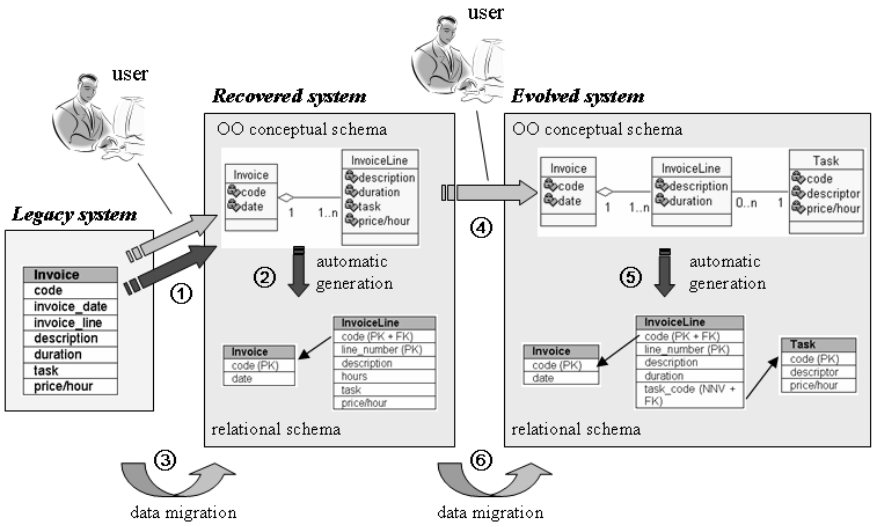


Figure 2: Motivating scenario

2 Motivating Scenario

[Fig. 2] presents a motivating scenario and exemplifies the recovery of a legacy system and its later evolution. It is a car maintenance company that has been working for a large car dealership. The maintenance company works with an old application in which the information is stored in a simple relational database that does not take integrity constraints into account. The car dealership has recently acquired the car maintenance company and they have decided to migrate the old application so that it is O-O, although the database layer will remain relational. 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 O-O conceptual schema that captures the semantics in the legacy system. This task is usually done manually, which is prone to human errors and amounts to high development costs. The first step is a manual, reverse-engineering process in which the designer detects that the legacy table can be divided into two classes: one containing the information about a task to be performed, and another that represents the collection of tasks performed for a specific customer, i.e., the `InvoiceLine` class and the `Invoice` class. (The results in [Hainaut, 1996], [Premerlani, 1994] or

[Ramanathan, 1996] can be applied here.) Once the model is complete, the relational database has to be generated. Here the designer can use a CASE tool to generate the new relational schema automatically. Despite obtaining a relational schema, these tools do not take into account legacy data.

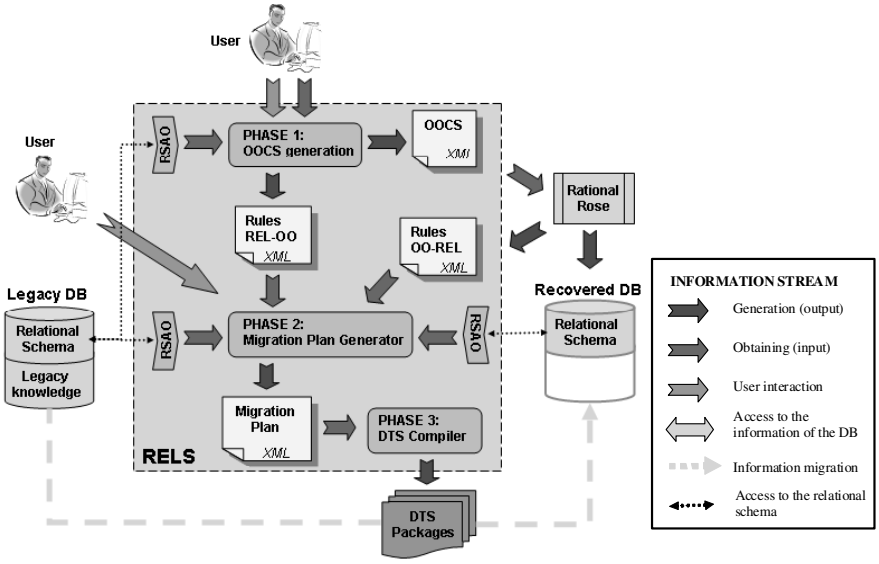


Figure 3: Legacy database recovery process in the RELS tool

The experience of the maintenance company is stored in its database and, it is expected to be preserved in the new database. Several DBMS allow for data migration using their own 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, DB administrators must write migration code manually, which is error-prone and costly.

Once the O-O 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 resulting schema: information about the same tasks is repeated and appears in several instances of the `InvoiceLine` class. Then, the designer decides to change the O-O conceptual schema by dividing the `InvoiceLine` class into two classes: a new class called `Task` that represents 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 is generated by means of the previous CASE tool. The data migration problem comes up again. Although ETL tools can be used as before, this situation differs from the first case since the data migration process has to be specified. Now, the original O-O 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 O-O 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 easily and efficiently.

3 RELS: Reverse Engineering of Legacy Systems

Legacy systems can be defined informally as "software we do not know what to do with, but still performs 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. The following features characterize them:

- Software architecture based on obsolete technology that may have been patched to adapt to new requirements, which complicates maintenance.
- Poor, complex documentation that prevents effective maintenance, making it necessary to check the source code to understand the functionality.
- 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 change continuously. There are two main approaches to performing changes in these systems. On the one hand, the legacy system can be patched. 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 re-trained. On the other hand, the whole system can be developed with a new technology taking advantage of all of its features. Both approaches imply a high cost, but we prefer the second option because the former delays the transition to into a new technology only temporarily, which makes maintenance harder each time the system is changed.

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 consists of the following steps:

- A data reverse engineering process that extracts an abstract description from the legacy system database to find its structure and its behavior. Changes can be applied to it in order to adapt the systems to new requirements or to new technologies. Our tool recovers a legacy database and obtains the static component of an equivalent O-O conceptual schema using formal methods.
- 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 by Rational [Boggs, 2002].

```
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

Our tool also allows to migrate data from the legacy database to the new one. The data reverse engineering process and the data migration process reduces the time needed and the number of people involved in the data evolution process. This optimization can be achieved due to the tasks that are performed behind the scenes, and it produces results that can be modified by the analyst. In this case, the process is semi-automatic. The tool performs the following phases [see Fig. 3]:

- 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 using an algebra.
- The data migration plan is compiled into DTS[1] packages whose execution migrates data from the legacy database to the new one automatically.
- 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 using a declarative language.

3.1 Data Reverse Engineering Phase

This phase takes the relational model of the legacy database as input and generates an O-O model that is equivalent to the previous one. These models are represented as terms of ADTs (Abstract Data Types) that are related by means of rewriting rules that are applied to the term that represents a relational schema by a TRS (Terms Rewriting System).

Term Rewriting Mechanism: An ADT is composed of a group of specification modules, each of which provides a set of operations (constructors and functions) to define terms and axioms to establish relations among them. Therefore, we define two ADTs in order to represent both relational and O-O conceptual schemas:

- A relational conceptual schema is represented as an algebraic term that is based on the syntactic and semantic rules provided by the relational ADT, which consists of several specification modules, each of which is related to a

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

relevant element of the relational model. The relational model specification module (`m-rel` in [Tab. 1]) is the core module that provides rules to define a relational conceptual schema term. A constructor is used to define an element of the relational model as a term, e.g., a table. Furthermore, axioms are used to specify the natural composition order between elements of the relational model, e.g., tables are composed of columns, indicating a compositional relationship between the specification modules of the relational ADT.

– The O-O model specification module (`m-oo` in [Tab. 2]) is the core module of the O-O ADT. It is composed of other modules that represent elements of the O-O model, e.g., 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 O-O 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: O-O specification module

Once we can define conceptual schemas as terms of both relational and O-O ADTs, we can relate each element of the relational ADT with different elements of the O-O ADT that are semantically equivalent. An ADT consists of specification modules, and a module can be formed by other modules, as we saw in the cases of the `m-rel` and `m-oo` modules. Thus, a new ADT is defined to relate elements of both relational and O-O ADTs. The rules of `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 O-O ADT. These rewriting rules, which represent the correspondences between elements of the relational model and elements of the O-O model, are applied automatically by a term rewriting system. Ours is finite and non-confluent, because we can obtain several O-O terms from the same 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 O-O 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 O-O ADT [Pérez, 2003].

The rewriting process is automatic but the user can validate whether the rules applied are the most suitable, because an element of the relational model might be

represented by several elements of the O-O 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. To reduce the interactions required by the user, we have taken into account the criterion that legacy databases were usually designed with access efficiency in mind.

The Data Reverse Engineering Process: The input of this phase is a relational schema of a legacy database, and it generates two XML documents as output: one that represents the O-O conceptual schema generated, and another that contains the rewriting rules applied to obtain the final O-O conceptual schema. In this phase the process that produces these outputs has three steps:

- Reading the relational schema of the legacy database. The access to the relational schema is performed by means of an API called RSAO in [Fig. 3]. This step 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 that do not allow for the definition of integrity or reference constraints, which are usually built into the business logic of the legacy system. 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 an intuitive graphical interface.
- Translation of the relational term into an O-O term by means of the rewriting rules described in the previous section. The user may decide to apply other rewriting rule than the default rules chosen by the TRS in order to generate an O-O term that is more accurate.
- Storage of the O-O term as an O-O conceptual UML schema in XMI, which allows to read the O-O conceptual schema generated with most CASE tools. The rewriting rules applied in the translation process are written to an 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 new database. Its inputs are two XML documents that contain the mappings between elements of the legacy relational schema and elements of the recently generated O-O conceptual schema, and between the elements of this O-O 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. The use of a declarative language provides independence from the specific DBMSs used to support the databases. Additionally, this phase checks the constraints of the target database in order to avoid constraint violation.

Relational Migration Plan: A relational migration plan specifies the actions that must be performed to copy data from the legacy database to the new one, generated from

the recovered O-O conceptual schema. The migration plan consists of a set of migration modules. There is one such module for each table of the target database that assigns a view over the legacy database to each target table. Migration modules contain a set of mappings between columns of the source view and the target table that constitute the migration expressions that can be used in a migration plan and they are specified by means of the relational declarative language.

```
add_ctr_pk([code, invoice_line], LegacyInvoice,
add_ctr_unique([code, invoice_line],
LegacyInvoice,
add_column(price/hour, currency, false,
LegacyInvoice,
add_column(task, string, false, LegacyInvoice,
add_column(duration, int, false, LegacyInvoice,
add_column(description, string, false,
LegacyInvoice,
add_column(invoice_line, int, false,
LegacyInvoice,
add_column(invoice_date, date, false,
LegacyInvoice,
add_column(code, int, false, LegacyInvoice,
add_table(LegacyInvoice,
create_database())))))))))))))
```

(a)

```
add_identif(line_number, InvoiceLine,
add_identif(code, Invoice,
add_unique(line_number, InvoiceLine,
add_unique(code, Invoice,
add_vbl_att(price_hour, currency, true,
add_vbl_att(task, string, true,
add_vbl_att(duration, int, true,
add_vbl_att(description, string, false,
add_ctt_att(line_number, int, true,
add_vbl_att(invoice_date, date, true,
add_ctt_att(code, int, true,
(add_aggregation(agg_Invoice_InvoiceLine,
Invoice, InvoiceLine, 1, 1, 1, n, false, true, false,
true,
add_class(Invoice,
add_class(InvoiceLine, create_schema())))))))))))))))))
```

(b)

Table 3: A relational term that represents the `Invoice` table of the legacy database (a), and an O-O term that represents the two aggregated classes of the new conceptual schema

The automatic generation of the migration plan takes its structure and its contents into account. Thus, two kinds of patterns are used: migration patterns and migration expression patterns. The migration plan generator gets the rewriting rules applied during the reengineering process from the two input XML documents, one from the data reverse engineering phase and another from the Rose Data Modeler. These rules provide enough information to determine what 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 map the attributes of the source view onto the corresponding attributes of the target table in each migration module Once the migration plan is finished, the generator writes it to an XML file.

Constraint Checking: Referential and integrity constraints in the target database imply new problems to the migration process because the legacy database is not supposed to support them. The migration plan generator checks these constraints to avoid errors during the data migration execution. To obtain the meta-information required about referential and integrity constraints of a database to generate the data migration plan, we focus on SQL99 [Türker, 2001]. Relational DBMS that are compliant with SQL99 associate to each database a set of tables that contain

```
<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 O-O conceptual schema

information about its schema. We have developed an API that reads these tables by means of the OLEDB interface [Lee, 2002].

The migration order is a sequence in which the tables of the target database must be filled to avoid violations of referential constraints. In the example in [Fig. 2], the `InvoiceLine` table has a foreign key into the `Invoice` table. If the migration process fills the `InvoiceLine` table first, the underlying referential constraint to the foreign key would be violated, i.e., the correct migration order `Invoice` first and `InvoiceLine` later. The generator of the migration plan produces a correct migration order by analyzing the relational schema of the database as if it was a directed graph in which the tables are the nodes and the foreign keys are the arcs. The generator takes into account 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. Furthermore, it also considers the integrity constraints of the legacy and the target database, because the analyst might complete the relational schema manually in the first phase. Thus, the schema generated 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 cause an error if there is a tuple that has a `null` value for its source column in the legacy data. Furthermore, 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, i.e., population filters, default values and data transformations.

3.3 The Migration Plan compiler

This phase compiles the migration plan into a specific technology and executes it. Each DBMS has its own ETL tool that allows to migrate data among databases. We

```
<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

use the Data Transformation Services (DTS) by Microsoft SQL Server, which allows to migrate data between heterogeneous relational DBMS. The DTS code that performs data migration is structured into DTS tasks that perform actions such copying data between two tables, executing SQL commands, or connecting to other databases.

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 and generates 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 is built, the compiler parses the migration expressions of each migration module and generates the contents of the DTS tasks of the corresponding DTS module.

The migration plan avoids target database constraint violations by means of several solutions. One of them is to ignore inconsistent data so that there may be much 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.

3.4 Example

In our motivating scenario, we began with a legacy system whose database consists of a table without any integrity constraint. RELS reads the legacy relational schema, and

the designer adds metadata that provides information about integrity constraints. The tool generates then the relational term in [Tab. 3.a]. The rewriting process obtains the O-O term that appears in [Tab. 3.b]. In this process, the designer has participated because the default rule that applies to a unique table generates a unique class, i.e., the user has chosen one of the rules proposed by the tool and has generated two aggregated classes.

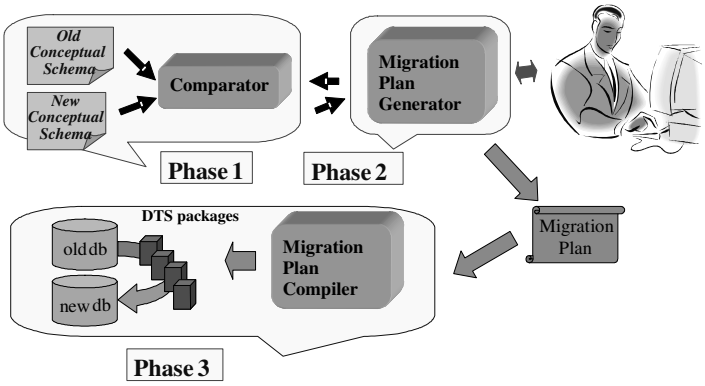


Figure 4: Data migration with ADAM

In [Tab. 4], we show part of the XMI document generated. We use Rational Rose to open this document 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. [Tab. 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 on the columns of the table.

Finally, the migration plan is compiled into DTS code. 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). They 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 after the other. The graphical interface allows to query the data in order to check the information migrated, 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 Fig. 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 new code and a new empty database from the new schema. The structure and the properties of the new database might be different from the older database. As the data remains compliant to the older database schema, the designer must preserve the data of the company by migrating it so that it satisfies the constraints of the new database.

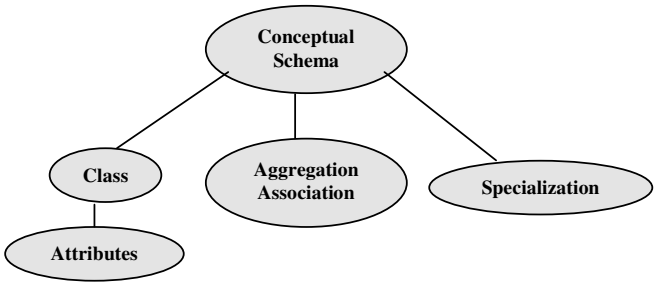


Figure 5: Tree representation of a conceptual schema

The migration task is necessary and is usually performed manually. This task increases considerably the maintenance cost of a software product. For this reason, an important issue is the improvement of the database maintenance process. Our proposal is the ADAM tool. The starting point was the work done by Carsí on OASIS reflection [Carsí, 1999]. OASIS is a formal language to define conceptual models of O-O 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 that builds on 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 as well.

In ADAM, the data migration process transfers and updates information system data from the old database to the new one [see Fig. 4]. Despite ADAM performs its steps automatically, the results can be modified easily, in which case the process is semi-automatic.

4.1 Matching between Conceptual Schemas

This phase is necessary to be able to discover the changes that have occurred in the old conceptual schema. The changes can be obtained by applying a matching algorithm whose result is the set of correspondences between the old and new

OLD CONCEPTUAL	NEW CONCEPTUAL
Invoice	Invoice
InvoiceLine	InvoiceLine
Null	Task

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

conceptual schemas. The matching is done automatically [Silva, 2002a]. Our algorithm obtains the correspondences between both schemas and shows whether an element is new or a modification of an older element. The matching algorithm is based on dynamic programming techniques and graph theory.

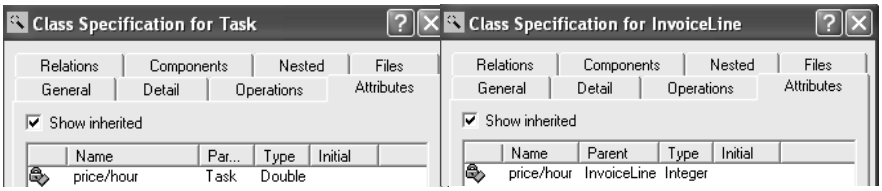


Figure 6: Price/Hour attribute of `Task` and `InvoiceLine` classes

To apply the algorithm, ADAM represents conceptual schemas as trees breaking cycles in relationships between their elements [see Fig. 5]. The nodes are classified into classes, aggregation and association relationships, specialization relationships, and attributes. This categorization of the matching space reduces the complexity, thereby reducing the processing. The algorithm can also use different matching criteria and combinations [Silva, 2002b]. A matching criterion allows 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, or the creation date. 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, and it achieved 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 [Tab. 6], we show the correspondences of the algorithm using the name matching criterion for the classes of the example presented in [Section 2].

4.2 Generation of a Data Migration Plan

From the correspondences detected in the first phase of the migration process, the second can generate the first version of a data migration plan automatically. A data migration plan is a set of data changes that are specified using a data migration language (ADML) [Pérez, 2002a]. 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 migration expressions, changes and modules, namely:

Migration expressions: They can be used in a data migration plan. Each type has different semantics and follows different syntactic constraints. Examples include Data Source, Transformation Function, Filters, and so on.

Changes: A change is the set of migration expressions that specify the updates undergone and the filters applied to the old database instances.

Migration modules: They contain 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 change in the old database is specified using a declarative O-O language. The advantage of ADML is the independence from DBMSs 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. Object-oriented conceptual schema elements are the data that are managed by the migration language that is object-oriented and uses path expressions to specify:

- The changes undergone by a conceptual schema element.
- The data that belongs to the old conceptual schema.
- The filters that will be applied on the data, if necessary.

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

- The correspondences between conceptual schemas produced previously.
- 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 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 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 Fig. 6]. [Fig. 5] shows that the prices of the `InvoiceLine` class were in Spanish Pesetas, and that they must be in Euro in the new `Task` class. As a result, the data type of the `price/hour` attribute was `integer` in the old schema, but `double` in the new one. The `price/hour` value must also be converted to the equivalent in Euro.
- 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 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`.

Process: First, ADAM generates the structure by 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 generated by 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 plan is performed using two types of patterns: migration and migration expressions [Pérez, 2002b]. We have specified them using the patterns design criteria proposed by [Gamma, 1994]. Each pattern is composed of several sections that give different qualities of the pattern.

Migration expression patterns: There is a pattern for each 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 that specify 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 Tab. 7].

In our example, ADAM uses the necessary patterns for each of the correspondences established between the attributes of the new `Task` class and the old `InvoiceLine` class. Moreover, we need to take into account that the transformation function generated by `IntToDouble(OldCS.Product.UnitPrice)` must be modified by the user to add the currency conversion function[2]. As a result, the transformation function that will be included in the data migration plan is `PtsToEuro(IntToDouble(OldCS.Product.UnitPrice)`.

Migration patterns: 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 associated 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

2 The `PtsToEuros` conversion function is included in the ADAM set of built-in transformation functions, and it converts from old Spanish Pesetas into Euro.

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 Tab. 8].

The first version of the data migration plan should be validated by the user after it is generated by ADAM. In addition, the users can modify the plan as needed. ADAM provides a graphical user interface 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 Fig. 7].

Outputs: After applying the necessary patterns to generate the data migration plan automatically, it is written to 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.

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**[1] and **P-08**[1] patterns, it is necessary to perform a type conversion in the transformation function as follows:

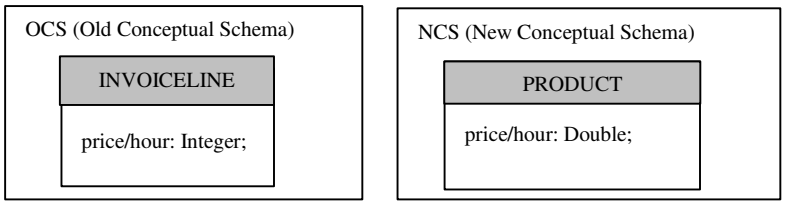
```
old_data_typeTOnew_data_type (old_attribute)
```

This pattern is a composition of the "name" and the "data type" property patterns (**P-03**[1] and **P-04**). The migrations expressions that express these changes are the following:

Transformation_Func.: `generic_func‘(‘IDENT_class‘.‘IDENT_attr‘)‘`

Example

The prices of the products were in Pesetas, and now they must be in Euro. 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 Euro.



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 patter P-08

P-01.Pattern: Elemental class

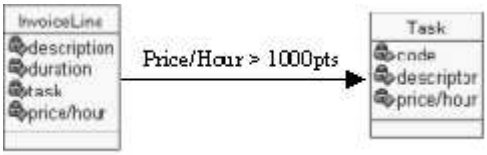
Solution

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 $(S2.C2) = \{y \mid \exists x \in data(S1.C1) \wedge \forall i \vDash_x g_i \; i=1,..,n \wedge ((y.a2 = f_n of_{n-1}...of_1(x.a1) \vee y.a2 = cte) \wedge \forall i \vDash_{x\downarrow a1} gc_i)\}$

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 $(S2.Task) = \{ y \mid \exists x \in data(S1.InvoiceLine) \vDash_x (S1.InvoiceLine.price/hour > 1000 \wedge (y.code = x.task) \wedge (y.descriptor = \text{“ ”}) \wedge y.Price = PtsToEuro(IntToDouble(x.UnitPrice)) \}$

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

```
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 (cont.): Solution and Example sections of the migration pattern P-01

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 phase generates automatically the code that a migration tool must produce manually using its script languages. Thus, ADAM 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 ability to specify complex expressions and migrate data between heterogeneous databases. 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 [Tab. 9].

5 Experimental Results

In this section, we indicate how we tested our tools on data migration.

5.1 Experimental Results with RELS

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, and DTS, which runs on Windows. As [Fig. 3] shows, the RSAO API reads the meta-information that constitutes the relational schema of the legacy database and structures this information into an XML document that is read by the translation module (phase 1), which uses the MAUDE system and produces two additional XML documents: one that describes the O-O model generated in XMI, and another that specifies the rules applied during the rewriting process.

The XMI document is used by Rational Rose to obtain the O-O conceptual model 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.

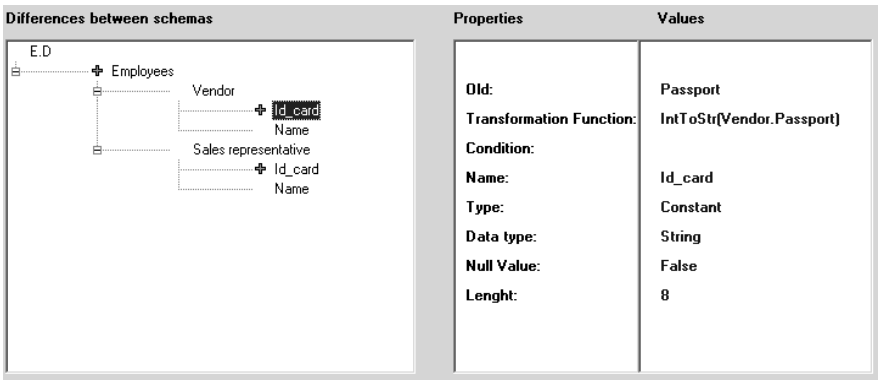


Figure 7: Graphical representation of the differences between elements

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

One of the tests that we applied to the RELS tool was a free accounting application, which stored its information into a relational database. We used RELS to recover this database, obtaining an O-O 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 using a team to build the new database and to migrate the information, which reduces costs in both staff and time.

5.2 Experimental Results with ADAM

The ADAM tool has a 3-tier architecture: the client layer includes the interface; the server layer implements services that allow ADAM to manage the data migration process; 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 validate the migration expressions defined by users syntactically and semantically. 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 it is called, it reports if the migration expression is valid and provides the decomposition of the migration expression in a XML tree. In addition, the checker needs information about the conceptual schema elements that includes the migration expression during its validation process. This information is gathered by querying the server.

With regard to the server layer, it implements the three phases needed to create a data migration plan [see Fig. 4]. Input conceptual schemas are XMI documents that are loaded into the database of the ADAM tool. This information is used by a module that implements the comparison algorithm, and the results are stored in the database. Another module implements the generation of the data migration plan. It uses the data stored in the database and its results are stored in an XML document. The structure of these XML documents is briefly presented in the migration patterns [see Tab. 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.

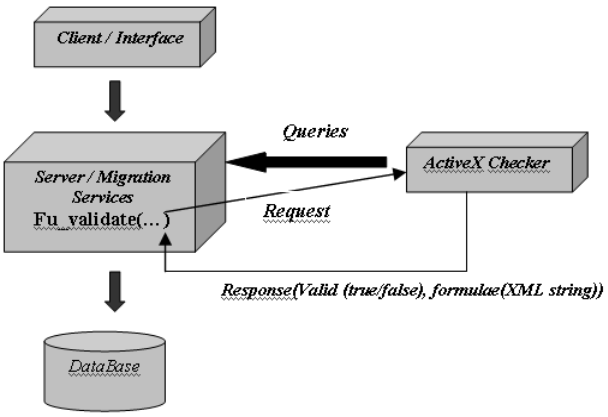


Figure 8: ADAM architecture

ADAM was tested using several examples that were provided by industrial partners and were used in order to evolve their data. We obtained better results than our industrial partners doing these tasks manually. Due to the privacy policies of the companies, we cannot publish these results; however, we can disclose that one of our analyst, who was not familiar with the information system, migrated the database using ADAM for two days and a half, whereas the same database took one month to be migrated manually by a experts who were familiar with the information system.

6 Related Work

In [Bisbal, 1999], a general migration process is split 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, 1996] 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 reverse engineering research that builds on the program comprehension theories of the 80s and the reverse engineering technology of the 90s. We focus on database understanding tools and we found 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 series of strategies to migrate data-intensive applications from a legacy data management system to a modern one that builds on the DB-Main CASE tool. Rational Rose also obtains O-O conceptual schemas from many DBMSs by means of the Data Modeler add-in. However, none of them takes into account legacy data recovery.

Not only does RELS 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 O-O 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 it by generating a relational schema that is semantically equivalent to the legacy database schema.

For the migration phase, we contrast the use of our tool with the other approaches 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 the software and hardware of the target environment. This approach was criticized in [Brodie, 1993], where the authors 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. With the Butterfly approach, new subsystems can be developed; however they are only taken into production once the whole system is finished using the Cold Turkey approach. The last phase involves a data migration process that eliminates the need for data gateways. RELS follows this approach and supports the automated generation of the new database by means of a formal data reverse engineering process.

RELS and ADAM focus on the data migration process, and takes into account heterogeneous relational DBMSs and manages 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 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 manually.

There are several proposals that study new algorithms to perform data migration more efficiently, e.g., [Anderson, 2001] and [Khuller, 2003]. They 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 is TESS [Staund, 2000], which revolves around an automatic process that is based on schema evolution, and it uses an intermediate language that is generated from the relational schema code. This is an important difference with our approach, because we deal directly with the O-O conceptual schemas, and do not have to translate them into an intermediate language. The O-O conceptual schemas give us a higher level of abstraction and eliminate the translation process.

The Varlet Database [Jahnke, 1998] support to transform a relational schema into an O-O conceptual schema and migrates the legacy data to the new O-O database. However, our approach considers a relational database as the persistence layer of an object society and migrates information to it. Furthermore, 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 interactively.

7 Final Remarks and Further Work

This paper reports on two experiences in software evolution that provide support to legacy system recovery and data migration. To recover a legacy system, we use 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, and obtains the term that represents the target O-O 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, to obtain a more accurate result.

The data migration problem is also introduced for the O-O conceptual schemas evolution where persistent layers are formed by relational databases. In this case, a matching process is applied between both O-O models to generate mappings between them that are used in the generation of the data migration plan. The automatic generation process gives us a preliminary 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.

RELS and ADAM work for several heterogeneous models by means of mappings between them that allows transformations between models of heterogeneous

metamodels. 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.

References

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

[Anaya, 2003] Anaya, V., Carsí, J.A., Ramos, I. (2003). Automatic evolution of database data. *Novática*, 164: 51-55.

[Balzer, 1985] Balzer, R. (1985). A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, 11(11): 1257-1268.

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

[Bisbal, 1999] Bisbal, J., Lawless, D., Wu, B., Grimson, J. (1999). Legacy Information Systems: Issues and Directions. *IEEE Software*. 16(5): 103-111.

[Boggs, 2002] Boggs, W., Boggs, M. (2002). Mastering UML with Rational Rose 2002. Sybex.

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

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

[Carsí, 1998] Carsí, J.A., Camilleri, S., Canós, J.H., Ramos, I. (1998). Homogeneous graphical user interface to design and use information systems. *Proc. JIS'98, III Workshop on Software Engineering*, Murcia, Spain.

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

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

[DB-Main] Hick, J., Englebert, V., Roland, D., Henrad, J., Hainaut, J. The DB-MAIN Database Engineering CASE Tool. http://www.fundp.ac.be/recherche.

[ESPRIT, 1996] Lancaster University (1996). RENAISSANCE Project - Methods & Tools for the evolution and reengineering of legacy systems.

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

[Hainaut, 1996] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., Hick, J.M (1996). Structure Elicitation in Database Reverse Engineering. *Proc. WCRE'96*, pages 131-140.

[Henrard, 2002] Henrard, J., Hick, J.M., Thiran, P., Hainaut, J.L. (2002). Strategies for Data Reengineering. *Proc. WCRE'02*, Richmond, Virginia, USA.

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

[Khuller, 2003] Khuller, S., Kim, Y., Wan, Y. (2003) Algorithms for Data Migration with Cloning. *SIAM Journal on Computing*, 33(2): 448 – 461.

[Kifer, 1995] Kifer, M. (1995). Deductive and Object Data Languages: A Quest for Integration, *Proc. of the DOOD'95*. Singapore.

[Lee, 2002] Lee, W. (2002). The Evolution of Data-Access Technologies. SQL Server Magazine. http://msdn.microsoft.com/library.

[Letelier, 1998] Letelier, P., Sánchez, P., Ramos, I., Pastor, O. (1998). *OASIS 3.0: A formal Approach for O-O Conceptual Modelling*. Universidad Politécnica de Valencia.

[Müller, 2000] Müller, H.A., Jahnke, J.H., Smith, D.B. (2000). *Reverse Engineering: A Roadmap*. In A. Finkelstein, editor, The Future of Software Engineering, ACM Press.

[Pérez, 2002a] Pérez, J., Carsí, J.A., Ramos, I. (2002). ADML: A Language for Automatic Generation of Migration Plans. *Proc. of the First Eurasian Conference on Advances in Information and Communication Technology*. Tehran, Iran. Springer LNCS vol n.2510.

[Pérez, 2002b] Pérez, J., Carsí, J.A., Ramos, I. (2002). On the implication of application's requirements changes in the persistence layer: an automatic approach. *Proc. DBMR'02, IEEE International Conference of Software Maintenance*, pages 3-16. Montreal, Canada.

[Pérez, 2003] Pérez, J., Anaya, V., Cubel, J.M., Ramos, I., Carsí, J.A. (2003). Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. *Electronic Notes in Theoretical Computer Science*, 72(4): 1-10.

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

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

[Rational] Rational Software, Rational Rose, `http://www.rational.com/`.

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

[Silva, 2002a] Silva, J.F., Carsí, J.A., Ramos, I. (2002). An algorithm to compare O-O-Conceptual Schemas. *Proc. of the ICSM'02*, Montreal, Canada.

[Silva, 2002b] Silva, J.F., Carsí, J.A., Ramos, I., Theoric analyze of the criteria of O-O conceptual schemas comparison*, Ingeniería Informática Magazine*, 7: 1-12.

[Sommerville, 1995] Sommerville, I. (1995). *Software Engineering*. Addison-Wesley.

[SystemArchitect] System Architect. http://www.popkin.com.

[Staund, 2000] Staund Lerner, B. (2000). A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Transactions on Database Systems*. 25(1):83-127.

[TogetherSoft] TogetherSoft Corporation. http://www.togethersoft.com.

[Türker, 2001] Türker, C., Gertz, M. (2001). 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*. 10(4): 241 - 269.

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

[Ward, 1995] Ward, M.P., Bennett, K.H. (1995). Formal Methods to Aid the Evolution of Software. *Journal of Software Maintenance: Research and Practice*, 7(3): 203-219.

[Wu, 1997] Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., O'Sullivan, D. (1997). The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems. *Proc. ICECCS'97.* Italy.

[Yourdon, 1996] Yourdon, E. (1996). *Rise and Resurrection of the American Programmer*. Yourdon Press, Upper Saddle River, NJ.