Co-transformations in Database Applications Evolution

Anthony Cleve and Jean-Luc Hainaut

Laboratory of Database Applications Engineering
University of Namur, Belgium
21 rue Grandgagnage 5000 Namur
{acl, jlh}@info.fundp.ac.be

Abstract. The paper adresses the problem of consistency preservation in data intensive applications evolution. When the database structure evolves, the application programs must be changed to interface with the new schema. The latter modification can prove very complex, error prone and time consuming. We describe a comprehensive transformation/generative approach according to which automated program transformation can be derived from schema transformation. The proposal is illustrated in the particular context of database reengineering, for which a specific methodology and a prototype tool are presented. Some results of two case studies are described.

1 Introduction

Software evolution consists in keeping a software system up-to-date and responsive to ever changing business and technological requirements. This paper focuses on the evolution of complex database applications, that is, data intensive software systems comprising a database. Database migration, database merging and database restructuring are popular evolution scenarios that involve not only changing the data components of applications, but also rewriting some parts of the programs themselves, even when no functional change occurs. In general, such evolution patterns induce the modification of three mutually dependent system components, namely the data structures (i.e., the *schema*), the data instances and the application programs [11]. When the system evolves, the consistency that exists between these three artifacts must be preserved.

In this paper, we focus on the consistency relationship that holds between the application programs and their database schema. We assume that the evolution process starts with a schema modification, potentially challenging this consistency. Our main question is the following: how can a change in a database schema be propagated to the application programs manipulating its data instances? Considering that any schema change can be modelled by a *transformation* (a rewriting rule that replaces a data structure with another one), we can formulate our question more precisely. The question can now be expressed

R. Lämmel, J. Saraiva, and J. Visser (Eds.): GTTSE 2005, LNCS 4143, pp. 409–421, 2006.

as follows: how can a schema transformation be propagated to the application programs that manipulate the data stored in the database? Through this transformational approach, database applications evolution will be modelled by the derivation of program transformations from schema transformations, that is, *co-transformations*.

The paper is structured as follows. Section 2 defines the concept of schema transformation. The way program transformations are derived from schema transformations is discussed in Section 3. In Section 4, we illustrate this general approach in a particular evolution context, i.e., data reengineering. Section 5 gives an overview of a tool architecture that support the whole process. In Section 6 first experiments are presented. We discuss related work in Section 7, while Section 8 concludes the paper.

2 Schema Tranformations

2.1 Definition

A schema transformation consists in deriving a target schema S' from a source schema S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty) [7]. C (resp. C') is empty when the transformation consists in adding (resp. removing) a construct. More formally, a transformation Σ is defined as a couple of mappings $<T, t>$ such that : $C' = T(C)$ and $c' = t(c)$, where c is any instance of C and c' the corresponding instance of C'. *Structural mapping* T explains how to modify the schema while *instance mapping* t states how to compute the instance set of C' from instances of C.

2.2 Semantics Preservation

The notion of semantics of a schema has no generally agreed upon definition. We assume that the semantics of a schema S_1 includes the semantics of another schema S_2 iff the application domain described by S_2 is a part of the domain represented by S_1. We can distinguish three schema transformation categories:

- T^+ collects the transformations that augment the semantics of the schema (e.g., adding an entity type).
- T^- includes the transformations that decrease the semantics of the schema (e.g., removing an attribute).
- $T^=$ is the category of transformations that leave the semantics of the schema unchanged (e.g., transforming a relationship type into a foreign key).

The transformations of category $T^=$ are called *reversible* or *semantics-preserving*. If a transformation is reversible, then the source schema can be replaced with the target one without loss of information. We refer to [6] for a more detailled analysis of semantics preservation in schema transformations.

2.3 Examples

Fig. 1[1] graphically illustrates the structural mapping T_1 of the transformation of a multivalued, compound attribute into an entity type and a relationship type R. Fig. 2 depicts the structural mapping T_2 of the transformation of a one-to-many relationship type R into a foreign key. Both transformations can be proved to be semantics-preserving.

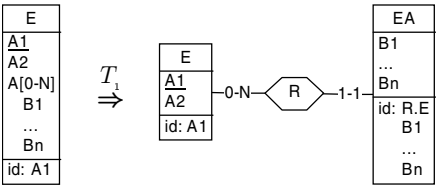

Fig. 1. Structural mapping T_1 of a semantics-preserving schema transformation that transforms a complex attribute A into entity type EA and relationship type R

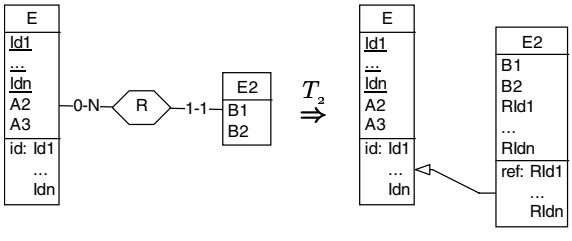

Fig. 2. Structural mapping T_2 of a semantics-preserving schema transformation that transforms a one-to-many relationship type R into a foreign key $RId_1 \cdots RId_n$

3 Deriving Program Transformations

The feasibility of automatically deriving program transformations from a schema transformation depends on the nature of the latter. In the general case, the transformations of T^+ and T^- categories do not allow automatic program modifications. The task remains under the responsibility of the programmer. However, it is generally possible to help him modify the code by automatically locating the program sections where occurrences of modified object types are processed. Program understanding techniques such as pattern searching, dependency graphs and program slicing allow to locate with a good precision the code to be modified [8].

[1] In Fig. 1 and 2, a box represents an entity type (record type or table). The first compartment specifies its names, the second one specifies its attributes (fields or columns) and the third one specifies the keys and other constraints: `id` stands for primary identifier/key; `acc` stands for access key or index; `ref` stands for foreign key. Relationship types between entity types are represented by diamonds.

Semantics-preserving schema transformations ($T^=$) can be propagated to the program level. Indeed, they allow the programming logic to be left unchanged, since the application programs still manipulate the same informational content. Program conversion mainly consists in adapting the related DMS[2] statements to the modified data structure. In order to define DMS-independent transformation rules, we will reason about the following abstract data modification primitives:

- `create` *var* `:=` *rec-name condition*: creates a record of type *rec-name* satisfying *condition*. The variable *var* contains or references the created record for further manipulation.
- `delete` *var condition*: deletes the record *var* if it satisfies *condition*, according to the chosen delete mode.
- `update` *var condition*: updates the record *var* in such a way that it satisfies *condition*.

Fig. 3 presents the correspondences between the above abstract primitives and concrete statements (COBOL, CODASYL and SQL). Access (reading) primitives are both more simple and more complex than modification primitives. In the one hand, the instance mapping states how instances can be extracted from the database according to the new schema. On the other hand, the way currency registers are implemented in various DMS can be quite different. Abstracting them in a DMS-independent manner is too complex to be adressed in this paper. Therefore, we assume, without loss of generality, that propagating schema transformations to reading primitives requires DMS-specific rules.

Abstract	COBOL	CODASYL	SQL
create	`WRITE`	`STORE`	`INSERT`
delete	`DELETE`	`ERASE`	`DELETE`
update	`REWRITE`	`MODIFY`	`UPDATE`

Fig. 3. Correspondences between data modification primitives

The problem translates as follows: given a schema transformation Σ applicable to data construct C, how can it be propagated to the abstract primitives that create, delete and update instances of construct C. Our approach consists in associating with structural mapping T of Σ, in addition to instance mapping t, three modification primitive mappings: t_c (create), t_d (delete) and t_u (update). These three mappings specify how to modify the corresponding primitives when T is applied to the database schema.

Fig. 4 shows mapping t_{1_c} we associate with structural mapping T_1 of Fig. 1. Since attribute A of entity type E has become entity type EA, the way an instance of E is created must be changed. It now involves the creation of EA instances corresponding to the old A instances (in a new loop). The created EA instances must be linked with the instance (e) of E through the relationship

[2] Data Management System.

type R. Fig. 5 illustrates the mapping t_{2c} associated with structural mapping T_2 of Fig. 2. The condition $R : e$ is replaced with a condition on the foreign key value.

$$
\begin{array}{l}
create\ e := E((:A_1 = a_1) \\
\quad and\ (:A_2 = a_2) \\
\quad and\ (:A[1].B_1 = b_{11}) \\
\quad \cdots \\
\quad and\ (:A[1].B_n = b_{1n}) \\
\quad \cdots \\
\quad and\ (:A[N].B_1 = b_{N1}) \\
\quad \cdots \\
\quad and\ (:A[N].B_n = b_{Nn}))
\end{array}
\quad
\begin{array}{c}
\\ \\ \\ \\ t_{1c} \\ \Rightarrow \\ \\ \\ \\ \\
\end{array}
\quad
\begin{array}{l}
create\ e := E((:A_1 = a_1) \\
\quad\quad and\ (:A_2 = a_2)); \\
\\
for\ i\ in\ 1..N\ do \\
\quad create\ ea := EA((:B_1 = b_{i1}) \\
\quad\quad\quad \cdots \\
\quad\quad\quad and\ (:B_n = b_{in}) \\
\quad\quad\quad and\ (R : e)) \\
endfor
\end{array}
$$

Fig. 4. Mapping t_{1c} associated with structural mapping T_1 of Fig. 1

$$
\begin{array}{l}
create\ e2 := E2((:B_1 = b_1) \\
\quad\quad and\ (:B_2 = b_2) \\
\quad\quad and\ (R : e))
\end{array}
\quad
\begin{array}{c}
\\ t_{2c} \\ \Rightarrow \\ \\
\end{array}
\quad
\begin{array}{l}
create\ e2 := E2((:B_1 = b_1) \\
\quad\quad\quad and\ (:B_2 = b2) \\
\quad\quad\quad and\ (:RId_1 = e.Id_1) \\
\quad\quad\quad \cdots \\
\quad\quad\quad and\ (:RId_n = e.Id_n)
\end{array}
$$

Fig. 5. Mapping t_{2c} associated with structural mapping T_2 of Fig. 2

4 A Particular Evolution Context: Data Reengineering

4.1 Definition

Data reengineering consists in deriving a new database from a legacy database and adapting the software components accordingly. Substituting a modern data management system (relational DBMS for instance) for an outdated data manager (typically standard file manager), or improving the database schema to gain better performance are popular scenarios. Typically, this migration process comprises the following three main steps [10]:

1. *Schema conversion*: the legacy database schema is translated into an equivalent schema expressed in the target technology.
2. *Data conversion*: the database contents are migrated from the legacy database to the new one. This step consists of a schema-driven *extract-transform-load* process.
3. *Program conversion*: the legacy programs are modified so that they access the new database instead of the legacy data. In the scenario studied, the functionalities, the programming language and the user interface are kept unchanged. This conversion step is generally a complex process that relies on the schema conversion step.

Data reengineering can be seen as a particular case of database applications evolution, in the sense that it typically involves compound, *semantics-preserving* schema transformations and related program transformations [4]. However, these program transformations do not only propagate schema transformations. In addition, they must *translate* the legacy DML[3] primitives into equivalent code fragments using the target DML.

4.2 Semantics-Based Schema Conversion

There are different approaches to convert the source schema into the target schema. Our approach consists of two steps:

1. Recovering the conceptual schema (i.e., the *semantics*) of the source database through a database reverse engineering phase [8].
2. Designing the target database from the CS obtained so far, through classical database engineering techniques.

This schema conversion approach has the merit of producing a well-designed, fully-documented database rid of the flaws of the legacy data, that forms a sound basis for both existing and future applications.

4.3 Wrapper-Based Program Conversion

In migration and interoperability architectures, wrappers are popular components that convert legacy interfaces into modern ones. In this context, a wrapper is a data model conversion component that is called by the client application programs to carry out operation on the database. For instance, a set of standard files is given an object-oriented API suited to modern distributed architectures.

In the data reengineering context we suggest the opposite approach, i.e., the use of *inverse wrappers* [9]. An inverse wrapper encapsulates the new database and provide access to the migrated data through a legacy API. It converts all legacy DMS requests issued by the legacy programs into requests compliant with the new DMS. Conversely, it captures the results from the new DMS, converts them according to the legacy format, and delivers them to the calling programs.

Our program conversion approach is a three-step method:

1. From all the schema transformations that are successively applied during the schema conversion phase (described in Section 4.2), we derive the mapping between the legacy DB schema (LDS) and the target DB schema (TDS).
2. From the LDS-to-TDS mapping, inverse wrappers are automatically generated. In practice, we generate one wrapper for each migrated record type.
3. The wrappers obtained so far are interfaced with the legacy application programs. This step mainly consists in replacing the legacy DML statements with corresponding wrapper invocations.

This program conversion approach allows the legacy applications to work on the new database with minimal alteration, and therefore at low cost.

[3] Data Manipulation Language.

4.4 Illustration

Let us consider COBOL record type ORD (Fig. 6−left) that is converted into two equivalent relational tables ORDERS and DETAILS (Fig. 6−right). This conversion is the result of the combination of several schema transformations:

1. $T_1 \equiv$ transforming the compound, multivalued attribute ORD-DETAIL into an entity type and a relationship type R;
2. $T_2 \equiv$ transforming the relationship type R into foreign key ORD_CODE;
3. $T_n \circ \cdots \circ T_3 \equiv$ renaming some data constructs to improve expressivity and comply with SQL syntax.

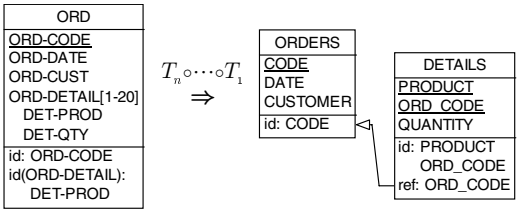

Fig. 6. Conversion of a COBOL record type into two relational tables

Each COBOL primitive that manipulate ORD records must be replaced with an equivalent procedural fragment on the relational data structures. Then let us examine the translation of the following COBOL statement:

```
WRITE ORD
```

This statement is first expressed as an abstract primitive:

```
create o := ORD((:ORD-CODE = ORD-CODE OF ORD)
    and (:ORD-DATE = ORD-DATE OF ORD)
    and (:ORD-CUST = ORD-CUST OF ORD)
    and (:ORD-DETAIL[1].DET-PROD = DET-PROD OF ORD-DETAIL ORD ORD(1))
    and (:ORD-DETAIL[1].DET-QTY = DET-QTY OF ORD-DETAIL OF ORD(1))
            ...
    and (:ORD-DETAIL[20].DET-PROD = DET-PROD OF ORD-DETAIL OF ORD(20))
    and (:ORD-DETAIL[20].DET-QTY = DET-QTY OF ORD-DETAIL OF ORD(20)))
```

By applying mapping t_{1c} associated with T_1 we obtain:

```
create o := ORD((:ORD-CODE = ORD-CODE OF ORD)
            and (:ORD-DATE = ORD-DATE OF ORD)
            and (:ORD-CUST = ORD-CUST OF ORD));
for i in 1..20 do
    create det := DETAILS((:DET-PROD = DET-PROD OF ORD-DETAIL OF ORD(i))
                      and (:DET-QTY = DET-QTY OF ORD-DETAIL OF ORD(i))
                      and (R : o))
endfor
```

By applying mapping t_{2c} associated with T_2 we refine this code fragment:

```
create o := ORD(  (:ORD-CODE = ORD-CODE OF ORD)
              and (:ORD-DATE = ORD-DATE OF ORD)
              and (:ORD-CUST = ORD-CUST OF ORD));
for i in 1..20 do
    create det := DETAILS((:DET-PROD = DET-PROD OF ORD-DETAIL OF ORD(i))
                      and (:DET-QTY = DET-QTY OF ORD-DETAIL OF ORD(i))
                      and (:DET-CODE = o.ORD-CODE))
endfor
```

Finally, the propagation of the renaming transformation $T_3 \cdots T_n$ provides us with an abstract code which is fully compliant with the structure of the SQL statements:

```
create o := ORDERS((:CODE = ORD-CODE OF ORD)
               and (:DATE = ORD-DATE OF ORD)
               and (:CUSTOMER = ORD-CUST OF ORD));
for i in 1..20 do
    create det := DETAILS((:PRODUCT = DET-PROD OF ORD-DETAIL OF ORD(i))
                      and (:QUANTITY = DET-QTY OF ORD-DETAIL OF ORD(i))
                      and (:ORD_CODE = o.CODE))
endfor
```

Generating the COBOL/SQL code is then straighforward:

```
WRITE-ORD.
  MOVE ORD-CODE OF ORD TO WR-CODE.
  MOVE ORD-DATE OF ORD TO WR-DATE.
  MOVE ORD-CUST OF ORD TO WR-CUSTOMER.
  EXEC SQL
     INSERT INTO ORDERS (CODE, DATE, CUSTOMER)
     VALUES (:WR-CODE, :WR-DATE, :WR-CUSTOMER)
  END-EXEC.
  MOVE 1 TO IND.
  PERFORM INSERT-DETAILS UNTIL IND >= 20 OR SQLCODE NOT= ZERO.

INSERT-DETAILS.
  MOVE DET-PROD OF ORD-DETAIL OF ORD(IND) TO WR-PRODUCT.
  MOVE DET-QTY OF ORD-DETAIL OF ORD(IND) TO WR-QUANTITY.
  MOVE ORD-CODE OF ORD TO WR-ORD-CODE
  EXEC SQL
     INSERT INTO DETAILS (PRODUCT, QUANTITY, ORD_CODE)
     VALUES (:WR-PRODUCT, :WR-QUANTITY, :WR-ORD-CODE)
  END-EXEC.
  ADD 1 TO IND.
```

Note that the resulting COBOL code can be equivalently generated in-line or encapsulated into an inverse wrapper. In our research, we favor the latter architecture. Therefore the initial COBOL `WRITE` statement is simply replaced with a corresponding wrapper invocation.

5 Tool Support

We have developped a prototype tool to support our data reengineering methodology, and particularly the program conversion phase. The architecture of this tool combines two complementary transformational technologies, namely the DB-MAIN [19] CASE tool and the ASF+SDF Meta-Environment [3].

5.1 Mapping Definition

We use the transformation toolkit of DB-MAIN to carry out the chain of schema transformations needed during the schema conversion phase. DB-MAIN automatically generates and maintains a history log of all the transformations that are applied to the legacy DB schema (LDS) to obtain the target DB schema (TDS). This history log is formalized in such a way that it can be analyzed and transformed. Particularly, it can be used to derive both the mappings between the LDS and the TDS.

5.2 Wrapper Generation

So far, wrapper generators for COBOL-to-SQL and IDS/II[4]-to-SQL have been developed. These generators are implemented through a plug-in of DB-MAIN. They take the LDS-to-TDS mapping as an input and generate the code that provides the application programs with a legacy interface to the new database. Each generated wrapper is a COBOL program with embedded SQL primitives.

The inverse wrapper generation involves two kinds of conversion rules. The first one involves API translation and is independent of the schema transformation. The legacy DMS primitives are simulated using target DMS primitives. For instance, a COBOL `READ` statement may require a complex procedure based on SQL cursors. This mapping layer is specific to each couple source/target models couple. The second conversion rules derives from schema transformations and are independent of API translation. The primitives initially expressed on the source schema are expanded into procedural fragments comprising primitives expressed on the target schema through rewriting rules such as those shown in Fig. 4 and 5.

5.3 Legacy Code Adaptation

The legacy application programs transformation relies on the ASF+SDF Meta-Environment. We use an SDF version of the IBM VS COBOL II grammar, which was obtained by Lämmel and Verhoef [12]. We specify a set of rewrite rules (ASF equations) on top of this grammar to obtain two similar program transformation tools. The first tool is used in the context of COBOL-to-SQL migration, while the second one supports IDS/II-to-SQL conversion.

Both program transformation tools are suitable in the context of partial migration, i.e., when only some legacy record types are actually migrated to the

[4] IDS/II is the BULL implementation of CODASYL.

new database platform. In this case, only the DML instructions manipulating migrated data are replaced with corresponding wrapper invocations. The other DML instructions, which still access the legacy data, are left unchanged.

We emphasize that the transformed legacy programs still manipulate the data through the legacy schema. In practice, this requires reorganizing the data declaration parts of the programs. For instance, in the case of COBOL-to-SQL reengineering, the following modifications are performed:

- the migrated files declarations are removed from the `INPUT-OUTPUT` section of the `ENVIRONMENT` division;
- the migrated record types declarations are moved from the `FILE` section of the `ENVIRONMENT` division to the `WORKING-STORAGE` section of the `DATA` division. Thus, the COBOL records become COBOL variables which are used as an argument of the wrapper calls.

6 Case Studies

Two small but realistic different legacy systems have been reengineered. Fig. 7 gives an overview of both case studies. As a first experiment, we converted an academic COBOL application managing data about students, registrations, results of exams, etc. The legacy database, consisting of 8 large COBOL files, was migrated to a relational database. The 15 legacy programs were successfully interfaced with 8 generated wrappers. A second case study was performed in collaboration with the company REVER, Belgium, devoted to system reengineering. The goal of this project was to reengineer a COBOL system from a city administration. This legacy system uses the IDS/II database manager and is made of 60 programs, totaling 35 KLOC. The resulting application consists of 57 KLOC, including 24 generated wrappers.

For both case studies the program conversion phase (i.e., wrapper generation and legacy code transformation) was fully automated. Our tools have proved to be quite efficient: generating the 24 IDS-to-SQL wrappers took 4 seconds while transforming the legacy programs required a bit less than 8 minutes.

	Case Study 1		Case Study 2	
	Source	Target	Source	Target
Host Language	COBOL	COBOL	COBOL	COBOL
DML	COBOL	SQL	IDS/II	SQL
# Entity types	8 records	18 tables	24 records	24 tables
# Attributes	291 fields	276 columns	257 fields	151 columns
# Rel. types	0	15 foreign keys	13 sets	21 foreign keys
# Legacy Programs	15	15	60	60
# Wrappers	0	8	0	24
# Wrappers calls	0	365	0	936
Legacy Code Size	7 KLOC	8.2 KLOC	35 KLOC	39 KLOC
Wrapper Code Size	0	6 KLOC	0	18 KLOC

Fig. 7. Case studies overview

7 Related Work

The concept of co-transformation (or coupled transformation) has been defined by Lämmel [14] as follows: *"A co-transformation transforms mutually dependent software artifacts of different kinds simultaneously, while the transformation is centred around a grammar (or schema, API, or a similar structure) that is shared among the artifacts."*. In [13] the same author identifies the category of coupled software transformations, describes their essence and lists typical application domains for co-transformations problems. For another example, we refer to the work by Lohmann and Riedewald [15] who present an elegant approach to automatically adapting transformation rules after a change in a grammar.

The concept of transformational engineering applied to data structures has been studied for more than two decades [16], first for schema engineering, then, later on, for data conversion. A fairly comprehensive approach has been described in [2]. However, as far as the authors know, propagating data structure transformations to procedural code has not been studied yet.

The use of wrapping techniques for reusing legacy software components is discussed in [18]. Papakonstantinou *et al.* [17] present a query translation scheme that facilitates rapid development of wrappers.

An iterative process model for legacy systems reengineering is proposed in [1]. One important phase of this method aims at making the legacy programs compatible with the migrated data, by replacing the data access instructions with calls to a *data banker*.

The purpose of [20] is to apply automatic restructuring transformations to large industrial Cobol systems in order to improve their modifiability. This work shows the suitability of ASF+SDF and the IBM VS COBOL II grammar for large-scale legacy code renovation.

Defining data mappings is an important issue in our work. The MDE-based approach proposed by Didonet *et al.* [5] considers data mappings as *models*. From this starting point, the authors suggest the use of model weaving as the base to solve various data mapping problems.

8 Conclusions

We have presented a general co-transformational approach for database applications evolution. This approach consists in formally defining an evolution as the application of coupled transformations, that modify the database schema, the data instances and the application programs. A methodology and a prototype tool have been proposed for a particular scenario of evolution, namely data reengineering.

Coupling generative and transformational techniques provides us with a promising tool support for data reengineering. First experiments have shown the validity of the approach, at least for small to medium size programs. However, our methodology and tools still have to be consolidated and validated for large information systems migration. In particular, the performance impact of our wrapper-based architecture should be evaluated through industrial case studies.

Acknowledgements. Many thanks to Ralf Lämmel, João Saraiva and Joost Visser for the organization of the GTTSE 2005 Summer School and for their patient editing work. Thanks too to the anonymous reviewers for their precise and pertinent advices. This research has been carried out within the context of the RISTART project, supported by the Belgian *Région Wallonne* and the European Social Fund. We thank the SEN1 group (CWI, Amsterdam) and REVER s.a. for their fruitful collaboration in this project.

References

1. Alessandro Bianchi, Danilo Caivano, Vittorio Marengo, and Giuseppe Visaggio. Iterative reengineering of legacy systems. *IEEE Trans. Softw. Eng.*, 29(3):225–241, 2003.
2. M. Boyd and P. McBrien. Towards a semi-automated approach to intermodel transformation. In *CAiSE Workshops Proceedings*, volume 1, pages 175–188. Riga Technical University, 2004.
3. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
4. Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Co-transformations in information system reengineering. In *Proc. of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering*, volume 137(3) of *ENTCS*, pages 5–15. Springer Verlag, 2005.
5. M Didonet Del Fabro, J Bézivin, F Jouault, and P Valduriez. Applying generic model management to data mapping. In *Proc. of the Journées Bases de Donnes Avancées (BDA05)*, 2005.
6. J.-L. Hainaut. Specification preservation in schema transformations - application to semantics and statistics. *Data & Knowledge Engineering*, 16(1), 1996.
7. Jean-Luc Hainaut. Transformation-based database engineering. In P. van Bommel, editor, *Transformation of Knowledge, Information and Data: Theory and Applications*, chapter 1. IDEA Group, 2005.
8. Jean Henrard. *Program Understanding in Database Reverse Engineering.* PhD thesis, University of Namur, 2003.
9. Jean Henrard, Anthony Cleve, and Jean-Luc Hainaut. Inverse wrappers for legacy information systems migration. In *WRAP 2004 Workshop Proceedings*, volume 04–34 of *CS Reports*, pages 30–43. Technische Universiteit Eindhoven, 2004.
10. Jean Henrard, Jean-Marc Hick, Philippe Thiran, and Jean-Luc Hainaut. Strategies for data reengineering. In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Computer Society Press, 2002.
11. Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution: a transformational approach. *Data and Knowledge Engineering*, 2006. to appear.
12. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
13. Ralf Lämmel. Coupled Software Transformations (Ext. Abstract). In *Proc. of the First International Workshop on Software Evolution Transformations*, Nov. 2004.

14. Ralf Lämmel. Transformations everywhere. *Science of Computer Programming*, 2004. The guest editor's introduction to the SCP special issue on program transformation.
15. Wolfgang Lohmann and Günter Riedewald. Towards automatical migration of transformation rules after grammar extension. In *Proc. of 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 30–39. IEEE Computer Society Press, 2003.
16. S. B. Navathe. Schema analysis for database restructuring. *ACM Transactions on Database Systems*, 5(2):157–184, June 1980.
17. Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the International Conference on Declarative and Object-oriented Databases*, 1995.
18. Harry M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering*, 9:293–313, 2000.
19. The DB-MAIN official website. `http://www.db-main.be`.
20. N. Veerman. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5): 219–254, 2004.