Harvesting models from web 2.0 databases

Oscar Diaz, Gorka Puente, Javier Cánovas, Jesus Garcia Molina

Noname manuscript No.
(will be inserted by the editor)

Harvesting Models from Web 2.0 Databases

Oscar Díaz[1], Gorka Puente[1], Javier Luis Cánovas Izquierdo[2] and Jesús García Molina[2]

ONEKIN Research Group[1], University of the Basque Country and ModelUM Research Group[2], University of Murcia e-mail: `{oscar.diaz,gorka.puente}@ehu.es`[1] `{jlcanovas, jmolina}@um.es`[2]

The date of receipt and acceptance will be inserted by the editor

Abstract Data rather than functionality, is the source of competitive advantage for Web2.0 applications such as wikis, blogs and social networking websites. This valuable information might need to be capitalized by third-party applications or be subject to migration or data analysis. Model-Driven Engineering (MDE) can be used for these purposes. However, MDE first requires obtaining models from the wiki/blog/website database (a.k.a. model harvesting). This can be achieved through SQL scripts embedded in a program. However, this approach leads to laborious code that exposes the iterations and table joins that serve to build the model. By contrast, a Domain Specific Language (DSL) can hide these "how" concerns, leaving the designer to focus on the "what", i.e. the mapping of database schemas to model classes. This paper introduces *Schemol*, a DSL tailored for extracting models out of databases which considers Web2.0 specifics. Web2.0 applications are often built on top of general frameworks (a.k.a. engines) that set the database schema (e.g., *MediaWiki*, *Blojsom*). Hence, table names offer little help in automating the extraction process. Additionally, Web2.0 data tends to be annotated. User-provided data (e.g., wiki articles, blog entries) might contain semantic markups which provide helpful hints for model extraction. Unfortunately, this data ends up being stored as opaque strings. Therefore, there exists a considerable conceptual gap between the source database and the target metamodel. *Schemol* offers extractive functions and view-like mechanisms to confront these issues. Examples using *Blojsom* as the blog engine are available for download.

Key words model-driven engineering, web2.0, harvesting, data re-engineering

1 Introduction

Model Driven Engineering (MDE) is not only applicable for creating new software systems, but it can also be used for modernizing or reengineering existing systems [1,41]. The latter first requires model harvesting (a.k.a. model extraction), i.e. the process whereby models are obtained from other software artefacts (e.g., code, databases, spreadsheets, etc) [24,37]. Once in the model realm, model transformation techniques can be used to perform modernization [33]. Whereas distinct works address model-driven code reverse engineering [24,37], little attention has been paid to data reverse engineering using MDE techniques. This paper addresses model harvesting from databases with a special focus on Web2.0 applications (e.g. wikis, blogs).

Web2.0 applications are particularly prone to model harvesting. For wikis/blogs, data (rather than code) is the main asset to retain and capitalize upon. Indeed, distinct wikis/blogs can run on the very same engine (e.g. *MediaWiki/Blojsom*), hence offering the same functionality. The difference rests on the data. This valuable data need to be retained. This implies facilitating migration to new platforms (e.g. blog engines), more to the point if we consider "the perpetual beta" that characterizes Web2.0 engines where backward compatibility is not always ensured. Furthermore, this data is a main asset to capitalize upon. Being provided by end users, Web2.0 data is frequently the subject of analysis. Studies on social networking scrutinize on Web2.0 data repositories to discover interaction patterns and user clusters

[31]. These scenarios can benefit from MDE techniques as long as models are obtained from Web2.0 databases (i.e., those managed by Web2.0 applications).

For our purposes, model harvesting implies obtaining models, conforming to a metamodel *MM*, from a database whose schema is *S*. Both *MM* and *S* might exist previously to the harvesting effort. This can potentially lead to a wide semantic gap between *MM* and *S*. As an example, consider the *Wikipedia* as a database along the *MediaWiki* database schema, and the *InteractionPattern* as a metamodel that is being used to represent contribution patterns for wiki articles [31]. Both, the *MediaWiki* schema and the *InteractionPattern* metamodel abstract the reality based on their own knowledge and data needs. Therefore, it is not surprising that both representations (i.e., the schema and the metamodel) greatly differ in their conceptual models. This breach is even wider for Web 2.0 databases due to:

1. Web2.0 applications run on top of so called "engines": "blog engines" (e.g., *Blojsom [19]*) or "wiki engines" (e.g., *MediaWiki* [8]). The point to notice is that these engines set the database schema. That is, all wikis using the *MediaWiki* engine are stored using the very same schema. Likewise, blogs using the *Blojsom* engine use the same tables. Of course, this is not the case for traditional database applications where, let's say, Oracle, does not restrict the tables of your database. Therefore, table names offer little help in automating the harvesting process.
2. Data tends to be annotated. Annotation is about attaching additional information (metadata) about an existing piece of data. Unlike "traditional" databases, Web2.0 data is likely to be annotated with HTML tags, microformats, CSS classes or RDFa annotations. Annotations can provide valuable clues about hidden entities. Unfortunately, annotations are stored together with the data as opaque strings, transparent to the database management system (DBMS).

Previous observations sustain the main premise of this work, namely, that *both the independent existence of the database and the metamodel, and the peculiarities of Web2.0 databases, might lead to a considerable semantic gap between the source database schema and the target metamodel.* Such gap can be bridged through SQL scripts embedded in a program using standardized APIs (e.g., JDBC for Java). However, this approach is tedious, and exposes the iterations and table joins that go to create the model. Object-Relational Mappers (e.g., *Hibernate* [4]) or "Model-Relational Mappers" (e.g., *Teneo* [12]) offer limited help. These mappers are first thought to make artefacts persistent rather than to obtain artefacts from tables. This makes these mappers assume a close resemblance between the database schema and the metamodel (or persistence object classes). By contrast, our premise is that the database exists independently from the metamodel, leading to a potentially wide gap which cannot always be bridged through mappers. A Domain Specific Language (DSL) can hide these "how" concerns, leaving the designer to focus on the "what". DSLs are reckoned to enhance the quality, productivity, maintainability and portability while permitting domain experts understand, validate and develop the DSL programs themselves [18].

This paper introduces *Schemol,* a DSL tailored for extracting models out of databases. *Schemol*'s main challenges (and contributions) can be summarized as follows:

– *Expressing database-to-model mapping in an intuitive way. Schemol* capitalizes upon the similarities between the database and modelware realms: database schemas and tuples can be easily equated to metamodels and models, respectively. This likeness advocates for *Schemol* to look like a traditional model-to-model transformation language such as *ATL* [28], hence, facilitating adoption among MDE practitioners.
– *Facing wide semantic gaps.* Our starting premise is the existence of conceptual mismatches between the source database schema and the target metamodel. This implies that model elements can not always be linked back to database tables but database queries. When a model element has not a clear counterpart in terms of tables, *Schemol* resorts to capture this model element in terms of a database query (i.e. a view).
– *Exploiting the annotated markup embedded in table attributes* (for Web2.0 databases). *Schemol* provides functions for making accessible annotations which are currently embedded as values of table attribute. This permits model elements (or attributes of model elements) to be obtained from annotations.

The rest of the paper is structured as follows. Section 2 provides grounds for a DSL by illustrating model harvesting using *JDBC*, *Hibernate* (i.e., an object persistence framework) and *Teneo* (i.e., a model persistence framework). Sections 3, 4 and 5 gradually introduce *Schemol* along the

	JDBC	**Hibernate**	**Teneo + m2m**	**Schemol**
Transparency in DB access	✗	✓	✓	✓
Transparency in model creation	✗(EMF)	✗(EMF)	✓	✓
Transparency in control flow	✗(Java)	✗(Java)	✓	✓
Transparency in reference resolution	✗(Java)	✗(Java)	✓	✓
Query Language	SQL	JPQL	JPQL + OCL-like	ad-hoc (dot notation)/SQL
Technologies required	JDBC, EMF, Java, SQL	JPA, Java, JPQL	JPA, JPQL, m2m lang	Schemol, SQL

Table 1 Approaches to model harvesting: *"declarativeness"* dimension. Only *Teneo* supplemented with model-to-model transformation techniques achieves similar degrees of transparency as those of *Schemol*.

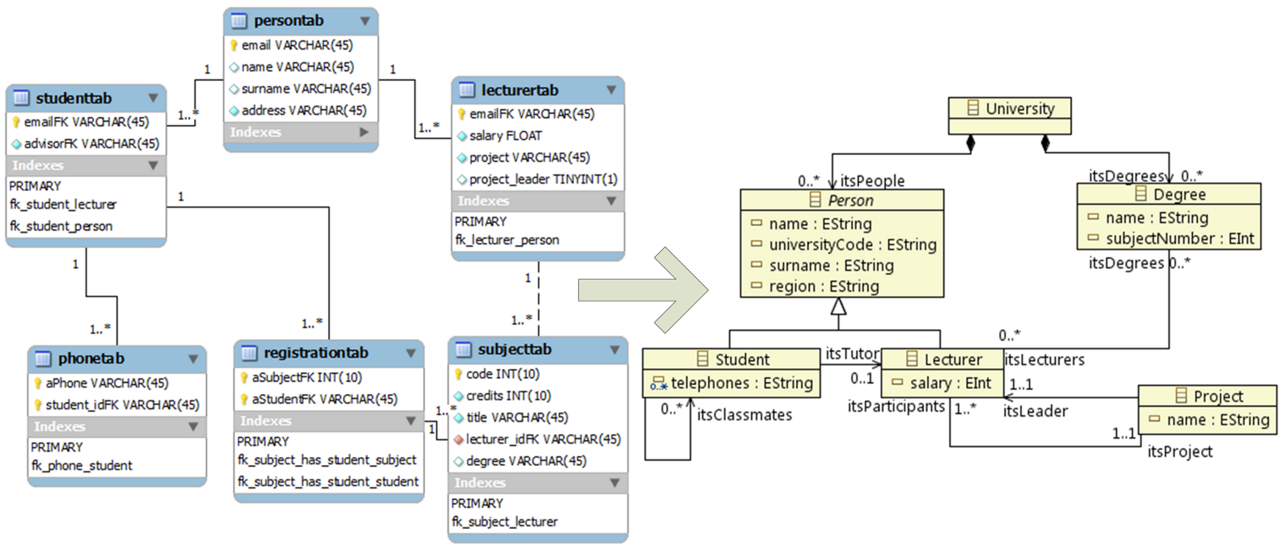


Fig. 1 *University* database schema (foreign keys add the suffix *FK*, e.g., *advisorFK*) & *University* metamodel (associations add the prefix *"its"*, e.g., *itsTutor*).

different means to construct model elements out of database tuples. Section 6 addresses the importance of annotated content in Web2.0 databases. This section illustrates how model elements can be derived from these annotations, using *Blojsom* as a case study. Section 7 outlines the implementation of the *Schemol* engine. Section 8 discusses the *Schemol* approach. A review on related works and some conclusions end the paper.

2 Harvesting Models Out of Databases

Model harvesting out of a database (hereafter, just "model harvesting") requires to express how model elements can be obtained from an *existing* database. Notice that this differs from model/object persistency where the database can be created *ad-hoc* for making the model persistent. In model harvesting, both the database schema and the metamodel are two independent artefacts, and the challenge rests on bridging how the domain has been captured in each realm. An entity that is given first-class status in the database (i.e., the entity is realized as a table) might not be given similar status in the target metamodel (i.e., the entity is not captured as an element).

Model harvesting involves two main tasks: 1) querying the database to obtain the necessary data, and 2) using this data for the creation and initialization of model instances. This section introduces three possible approaches to model harvesting: (1) raw programming through direct database access (illustrated through *Java+JDBC*), (2) object-relational mappers (ORMs) (illustrated through *Hibernate*), and (3), model-relational mappers combined with model-to-model transformation (illustrated through *Teneo+ATL*). The comparison provides grounds for a DSL for model harvesting (illustrated by *Schemol*). These approaches differ in both declarativeness and expressiveness.

	JDBC	**Hibernate**	**Teneo+m2m**	**Schemol**
Main Focus	Database access	Object persistence	Model persistence	Model harvesting
Single-tuple element	✗	✓	✓	✓
Multi-tuple element	✗	✓	✓	✓
Specialization-sourced element	✗	✓	✓	✓
Column-based element	✗	✗	limited (through m2m)	✓
Collection element	✗	✗	limited (through m2m)	✓
Annotation-based element	✗	✗	✗	✓

Table 2 Approaches to model harvesting: *"expressiveness"* dimension. For the *Teneo* option, *"limited"* indicates that *Teneo* should be supplemented with model-to-model transformation techniques to account for either column-based elements or collection elements.

Declarativeness. Model harvesting involves: database access (i.e., defining database services), model creation (i.e., creation and population of model elements), control flow (i.e., order in which model elements are created), reference resolution (i.e., the issue of turning primary keys and foreign keys into identifiers and associations, respectively) and data querying. Approaches can be compared based on the transparency offered for each of these tasks, i.e., whether they should be taken care of by the programmer or abstracted into a framework. Table 1 summarizes the findings which are later discussed.

Expressiveness. DSLs trade generality for expressiveness in a limited domain. General-purpose languages such as Java + JDBC, can be more general but less expressive, in the sense that they require convoluted solutions to capture domain-specific scenarios, and hence, they can put domain experts off. From this perspective, expressiveness is related with declarativeness but a DSL can be declarative but little expressive in the sense that few domain scenarios can be directly described. Hence, assessing expressiveness requires to first identify the harvesting scenarios, and next, check whether these scenarios can be directly expressed. Table 2 compares *JDBC, Hibernate, Teneo* and *Schemol* along their capacity to handle these scenarios (next introduced) in an explicit way (i.e. by providing dedicated constructs).

We illustrate these divergences by an example from the *University* domain (see Figure 1). The database designer focuses on people and subjects. Due to normalization, person data is split across different tables: *personTab*, *studentTab* (which keeps student specific data), *phoneTab* (that collects the set of phones) or *registrationTab* (that holds the M:N relationship between students and subjects). By contrast, the metamodel designer focuses on people and degrees. The representation of people resembles that of the database but without the limitation of normalization. However, *Degree* and *Project* emerge as first-class elements while they where hidden in the database. A harvesting DSL should be able to express not only direct mappings but also more intricate situations such as those of *Degree* and *Project*. Specifically, the following scenarios are considered:

– *Single-tuple elements.* In this case, there exists a direct correspondence between tuples and model elements. Each tuple will have a model element counterpart which is populated from the tuple columns. This implies that the entity at hand is given first-class representation in both the database and the metamodel.
– *Multi-tuple elements.* Database normalization forces data about an entity to be separated among different tables. This process might need to be reversed when constructing model instances. The construction of student instances illustrates this point. Data scattered around different tables is put together to come up with a *Student* model element.
– *Specialization-sourced elements.* In this case, entity data is distributed along a hierarchy. For instance, student data is also contained in *personTab,* which stores data that applies to both students and lecturers. The mapping should indicate how this situation should be mapped into class elements. Different scenarios are supported by object-relational mappers as presented below.
– *Column-based elements.* Model element instances do not always equate to tuples. *Project* is a case in point. It has not been considered as a main concern in our database sample (no *projectTab* exists) while it has been explicitly represented in the target metaclass. This does not mean that the database is unaware about projects, but that this concern has not been given first-class status. For instance, the notion of project can be ascertained from the *project_leader* attribute of *lecturerTab*. Column-based

elements are those that can be derived from attribute values.

– *Collection elements.* Collections are not always explicitly modelled since (some) data about the collection can be derived from its constituents. Indeed, SQL SELECT statement evidences this fact by providing the GROUP BY clause. This clause retrieves data about sets of tuples (rather than single tuples) that conform to meaningful units. This clause reflects the existence of aggregates that are not explicitly captured by the database schema but users can query about. The metamodel designer can decide to give then first-class status. *Degree* illustrates this situation. A degree is a collection of subjects.
– *Annotation-based elements.* This caters for a Web2.0 peculiarity: annotated content. Annotations can also be the source of full-fledged entities/attributes that are captured explicitly in the metamodel. The running example illustrates this case through the *address* attribute. This attribute contains a string which holds *HCard* annotations[1]. *HCard* provides distinct classes to annotate the different components of an address (e.g., street-address, locality, region, etc). When moved to the model realm, only the *region* becomes a *person* attribute.

Being a general-programming language, *Java+JDBC* supports all these scenarios but through direct coding. *Schemol* has also been designed to support the harvesting casuistic but now in more abstract terms through declarative mappings. Finally, *ORM* tools (both class and model mappers) are limited in their expressiveness to capture some of these settings. This is not surprising since *ORM* tools are first thought for replicating class structures into tables rather than from mapping tables into classes. The rest of this section provides grounds for this comparison by providing some code snippets for each of these approaches.

2.1 Using Call-level Interfaces

Model harvesting first involves querying the database to obtain the necessary data. Database access can be conduced through call-level interfaces (e.g., ODBC or JDBC) where SQL statements are used to recover data into datasets. Once loaded, this data can populate the models using the *Eclipse Modeling Framework* API (EMF) [39]. Figure 2 shows a snippet for the creation of *Student* elements using JDBC.

[1] *HCard* is a microformat [9].

Student data is split along distinct tables (e.g., *personTab, studentTab*, *phoneTab*, *registrationTab*). The snippet shows how data from these different tables is first collected into result sets (e.g., *rsPersonData, rsPhones*) (lines 24-27). The creation of a *Student* element includes obtaining *itsClassmates* which might refer to *Student* model elements not yet created. Hence, populating this attribute might cause to create students other than the one at hand (line 45). Therefore, a student might be already created by the time their tuple counterpart is processed. This explains the condition at the beginning of the snippet (line 16).

Once data is loaded, the metamodel can be populated. Having an *Ecore* metamodel as input, *EMF* generates mirror classes for each of the elements defined in the metamodel. These classes are next used to instantiate (line 18) and set element attributes (lines 29-33). Notice that the mapping and order in which model elements are created is completely up to the programmer. An additional issue is reference resolution. Foreign keys need to be turn into their identifier counterparts. This is illustrated for the *itsTutor* association that is obtained from the *advisorFK* foreign key (lines 36-40).

Any model element can be obtained through *ad-hoc* code. The drawback is that this approach makes explicit the joining of tables, the iterations along result sets, the mapping of hierarchies and so on. Besides being time consuming, surfacing these details makes the code difficult to maintain. As in other domains, the solution rests on providing an abstraction layer. Object-relational mappers provide such a layer.

2.2 Using Object-Relational Mappers

Object-Relational Mappers (ORMs) are used for *transparent* object persistence. The mapper establishes a bidirectional link with data in a relational database and objects in code. No need to explicitly insert/delete/update tuples. Persistence objects are transparently uploaded and downloaded by the mapper. The Java Persistence Architecture API (JPA) is a Java specification for this purpose. *Hibernate* is a JPA-compliant ORM [4]. When ORMs are used for model harvesting, two mappings are involved: the table-to-class mapping (e.g., from *studentTab* to *studentClass*) and the class-to-model mapping (e.g., from *studentClass* to the *Student* model element).

Table-to-class mapping (see Figure 3). This mapping is the duty of the ORM. Now, data is not longer kept

```
 // Main method {...}
 Class.forName("com.mysql.jdbc.Driver").newInstance();
 String url = "jdbc:mysql://localhost/universitysch";
 conn = DriverManager.getConnection(url, "root", "12345");
 Statement st = conn.createStatement();
 ResultSet rsStudents = st.executeQuery("select emailFK, advisorFK from studentTab");
 //createDegrees();createLecturers();createProjects();
 while (rsStudents.next())
  createStudent(rsStudents);
 //conn.close(); save(); catch Exceptions {...}
}
private Student createStudent(ResultSet rsStudent) {
 StudentModel studentModel = null;
 // If student has been created previously, return it
 // (studentCache stores created students)
 if((studentModel = studentCache.get(rsStudent)) == null) {
  // Creation of the Student model element
  studentModel = UniversitypkgFactory.eINSTANCE.createStudent(); //Create model instances
  // Store the student into the university
  university.getItsPeople().add(studentModel);
  try {
   //create statements {...}
   //Collect data
   ResultSet rsPersonData = st1.executeQuery("select name, surname, address from personTab where"+
                                             " email='"+ rsStudent.getString("emailFK")+"'");
   ResultSet rsPhones = st2.executeQuery("select aPhone from phoneTab where student_idFK='"+
                                         rsStudent.getString("emailFK")+"'");
   rsPersonData.first();
   // Initialization of the main attributes
   studentModel.setName(rsPersonData.getString("name"));
   studentModel.setSurname(rsPersonData.getString("surname"));
   studentModel.setUniversityCode("Student_"+rsStudent.getString("emailFK"));
   studentModel.setRegion(getRegion(rsPersonData.getString("address")));
   while (rsPhones.next())// Add the phones
    studentModel.getTelephones().add(rsPhones.getString("aPhone"));
    // To establish the tutor we must locate first the lecturer in the ecore model
    for (Iterator iter = university.getItsPeople().iterator(); iter.hasNext(); ){
     Person lect = (Person) iter.next();
     if (lect instanceof Lecturer && lect.getUniversityCode().equals(rsStudent.getString("advisorFK")))
      studentModel.setItsTutor((Lecturer) lect);}
    ResultSet rsClassmates = st3.executeQuery("SELECT aStudentFK as emailFK, advisorFK FROM registrationTab"+
     " reg1, studentTab stu1 where stu1.emailFK = aStudentFK and aSubjectFK in (select aSubjectFK from" +
     " registrationtab reg2 where reg2.aStudentFK = '"+ rsStudent.getString("emailFK")+"')");
    while (rsClassmates.next())
     studentModel.getItsClassmates().add(createStudent(rsClassmates));
    studentCache.put(rsStudent, studentModel);
    } catch (SQLException ex) System.err.println(ex.getMessage());
   } else return studentModel;
    return studentModel;
  }
```

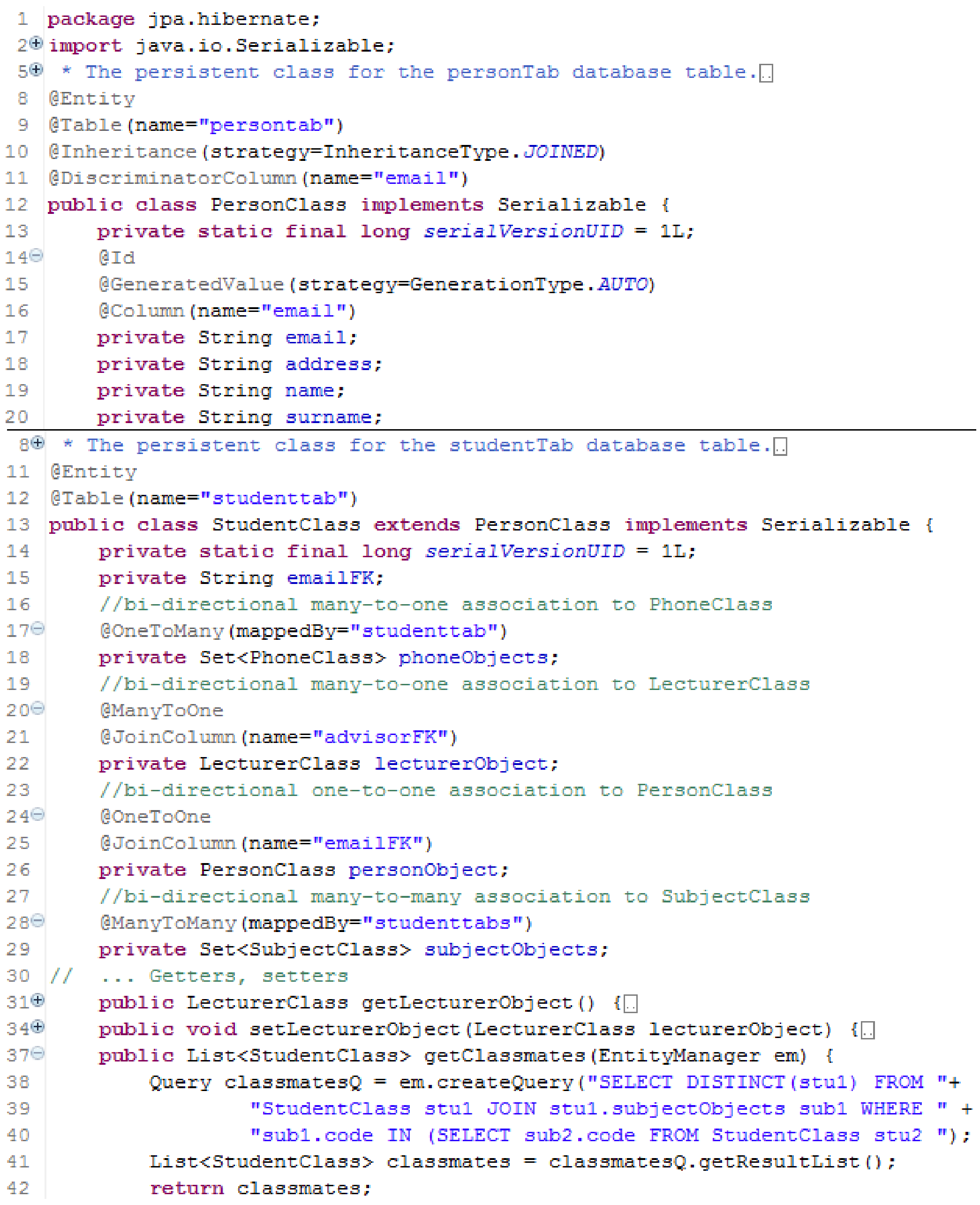
Fig. 2 *JDBC* approach: all model elements can be created at the expenses of tedious coding.

in datasets but persistent objects. Using *JPA*, the designer annotates how persistent classes are to be mapped to the database. Single-tuple elements are annotated with their table counterpart (i.e., @*Table* annotation). Specialization-sourced elements should indicate the strategy to be followed to capture the hierarchy implicitly supported through tables (i.e., @*Inheritance* annotation)[2]. For our sample problem, the *person* hierarchy follows the one-table-per-class strategy (*JOINED* strategy in *JPA* parlance) (line 10): *PersonClass* maps to *personTab*, *StudentClass* maps to *studentTab*, and so on. Finally, the @*DiscriminatorColumn* indicates the key column (e.g., *email*, line 11).

Once the mapping is specified, it is up to *Hibernate* to properly upload/download objects into tables. The program can now request the mapper to load *Student* instances into memory. This results in, for each student tuple a student object counterpart is generated in the program runtime.

Class-to-model mapping (see Figure 4). Once data is loaded as object instances, we proceed to create model elements using *EMF*. Figure 4 shows the creation of a

[2] JPA strategies include: *SINGLE_ TABLE* (i.e., the whole inheritance hierarchy is flattened into a single table), *TABLE_PER_CLASS* (i.e., every "leaf class" in the hierarchy is mapped to a separate table which includes all the attributes along the hierarchy), and *JOINED* (i.e., every class in the hierarchy is mapped to a separate table which keeps only the attributes declared on the mirror class).



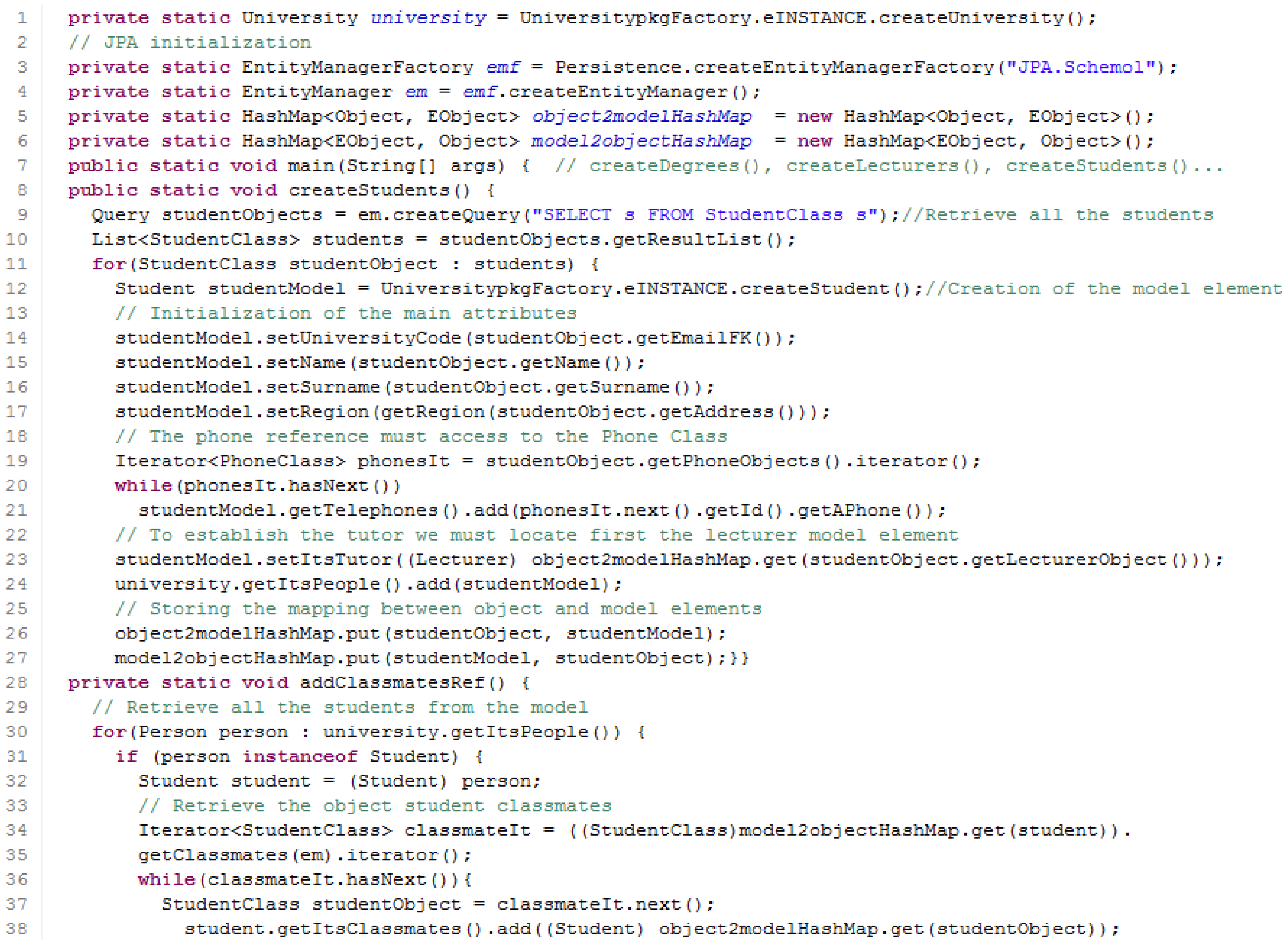
```
 1 package jpa.hibernate;
 2 import java.io.Serializable;
 5  * The persistent class for the personTab database table.
 8 @Entity
 9 @Table(name="persontab")
10 @Inheritance(strategy=InheritanceType.JOINED)
11 @DiscriminatorColumn(name="email")
12 public class PersonClass implements Serializable {
13     private static final long serialVersionUID = 1L;
14     @Id
15     @GeneratedValue(strategy=GenerationType.AUTO)
16     @Column(name="email")
17     private String email;
18     private String address;
19     private String name;
20     private String surname;
```

```
 8  * The persistent class for the studentTab database table.
11 @Entity
12 @Table(name="studenttab")
13 public class StudentClass extends PersonClass implements Serializable {
14     private static final long serialVersionUID = 1L;
15     private String emailFK;
16     //bi-directional many-to-one association to PhoneClass
17     @OneToMany(mappedBy="studenttab")
18     private Set<PhoneClass> phoneObjects;
19     //bi-directional many-to-one association to LecturerClass
20     @ManyToOne
21     @JoinColumn(name="advisorFK")
22     private LecturerClass lecturerObject;
23     //bi-directional one-to-one association to PersonClass
24     @OneToOne
25     @JoinColumn(name="emailFK")
26     private PersonClass personObject;
27     //bi-directional many-to-many association to SubjectClass
28     @ManyToMany(mappedBy="studenttabs")
29     private Set<SubjectClass> subjectObjects;
30 //  ... Getters, setters
31     public LecturerClass getLecturerObject() {
34     public void setLecturerObject(LecturerClass lecturerObject) {
37     public List<StudentClass> getClassmates(EntityManager em) {
38         Query classmatesQ = em.createQuery("SELECT DISTINCT(stu1) FROM "+
39                 "StudentClass stu1 JOIN stu1.subjectObjects sub1 WHERE " +
40                 "sub1.code IN (SELECT sub2.code FROM StudentClass stu2 ");
41         List<StudentClass> classmates = classmatesQ.getResultList();
42         return classmates;
```

Fig. 3 ORM approach. Table-to-class mapping: *PersonClass* objects are generated from *JPA* annotations. So are *StudentClass* objects. Besides database-sourced attributes, programmers can provide additional methods (e.g., *getClassmates,* line 37*).*

studentModel out of a *studentObject.* Mono-valued attributes are resolved with a simple assignment (lines 13-17). However, multi-valued attributes require iterating over the distinct values in the object instance (e.g., *telephones*, lines 18-21). Additionally, the programmer is also responsible for keeping the object-to-model and model-to-object mapping tables which are needed for reference resolution. The *itsTutor* association provides an example (line 23). This association already exists between objects. Now, we need to recreate this association for model elements. This is the purpose of the *object2modelHashMap* table and the *model2objectHashMap* table. These tables keep track of the model elements being created so far. Hence, the creation of a *studentModel* (line 11-24) is followed by an update on these mapping tables (lines 25-27). Therefore, ORMs do not provide transparent model creation or reference resolution.

For both single-tuple elements and specialization-sourced elements, ORMs considerably reduce coding. However, some attributes and elements in the target metaclasses (e.g., *Student's itsClassmates*, *Degree*) could not always



```
private static University university = UniversitypkgFactory.eINSTANCE.createUniversity();
// JPA initialization
private static EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPA.Schemol");
private static EntityManager em = emf.createEntityManager();
private static HashMap<Object, EObject> object2modelHashMap  = new HashMap<Object, EObject>();
private static HashMap<EObject, Object> model2objectHashMap  = new HashMap<EObject, Object>();
public static void main(String[] args) {  // createDegrees(), createLecturers(), createStudents()...
public static void createStudents() {
  Query studentObjects = em.createQuery("SELECT s FROM StudentClass s");//Retrieve all the students
  List<StudentClass> students = studentObjects.getResultList();
  for(StudentClass studentObject : students) {
    Student studentModel = UniversitypkgFactory.eINSTANCE.createStudent();//Creation of the model element
    // Initialization of the main attributes
    studentModel.setUniversityCode(studentObject.getEmailFK());
    studentModel.setName(studentObject.getName());
    studentModel.setSurname(studentObject.getSurname());
    studentModel.setRegion(getRegion(studentObject.getAddress()));
    // The phone reference must access to the Phone Class
    Iterator<PhoneClass> phonesIt = studentObject.getPhoneObjects().iterator();
    while(phonesIt.hasNext())
      studentModel.getTelephones().add(phonesIt.next().getId().getAPhone());
    // To establish the tutor we must locate first the lecturer model element
    studentModel.setItsTutor((Lecturer) object2modelHashMap.get(studentObject.getLecturerObject()));
    university.getItsPeople().add(studentModel);
    // Storing the mapping between object and model elements
    object2modelHashMap.put(studentObject, studentModel);
    model2objectHashMap.put(studentModel, studentObject);}}
private static void addClassmatesRef() {
  // Retrieve all the students from the model
  for(Person person : university.getItsPeople()) {
    if (person instanceof Student) {
      Student student = (Student) person;
      // Retrieve the object student classmates
      Iterator<StudentClass> classmateIt = ((StudentClass)model2objectHashMap.get(student)).
      getClassmates(em).iterator();
      while(classmateIt.hasNext()){
        StudentClass studentObject = classmateIt.next();
          student.getItsClassmates().add((Student) object2modelHashMap.get(studentObject));
```

Fig. 4 ORM approach. Class-to-model mapping

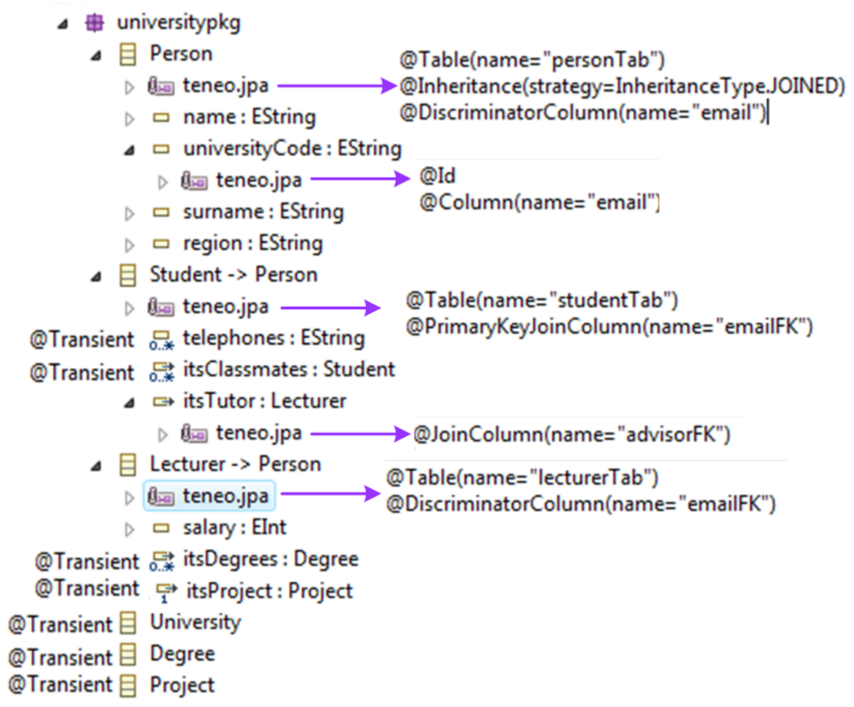
find a direct database counterpart (e.g., *Student* classmates attribute). Classmates are those students that share at least one subject in common with the student at hand, including himself. No direct counterpart exists in the database. Hence, it could have then been obviated in the persistent class hierarchy. However, on the search for an easy object-to-model mapping, we can extend the *StudentClass* (see Figure 3) with a method that looks like as the getter for the *Classmates* attribute (line 37-42). This method would require iterating over the student set. Fortunately, *JPA* provides a query language, named *JPQL,* which permits to hide these iterations as a *SQL*-like query. The bad news is that you need to know *JPQL*. Another shortcoming is that of column-based elements and collection elements since they can not be captured through *JPA* annotations. In this case, manual coding is required.

All in all, ORMs are a good option when there is a close resemblance between tables and model elements. ORMs hide database access but still require explicit *EMF* coding to create model elements. The latter can be made transparent through "model-relational mapping tools".

2.3 *Using Model-Relational Mappers*

Model-Relational Mappers (MRMs) are to model programming what ORMs represent for object-oriented programming: means to achieve transparent persistency for large artefacts. *Teneo* addresses the persistence of *EMF* models using *JPA* [12]. This permits to directly annotate the metamodel rather than their *Java* class counterparts. Figure 5 illustrates the use of *Teneo* for the sample problem. *EMF* metamodels are now decorated with the very same *JPA* annotations used for Java classes in the previous section. Now, model creation and control flow are made transparent to the programmer.

Teneo is based on JPA, so it suffers from the same expressiveness limitations that ORMs. After all, *Teneo* rests on *Hibernate*. Column-based elements and collection elements can not be captured. This limitation might be partially overcome by using model-to-model transformations.



```
@Table(name="personTab")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="email")
@Id
@Column(name="email")
@Table(name="studentTab")
@PrimaryKeyJoinColumn(name="emailFK")
@JoinColumn(name="advisorFK")
@Table(name="lecturerTab")
@DiscriminatorColumn(name="emailFK")
```

Fig. 5 MRM approach. Annotated metamodel in *TENEO*: JPA now decorates model elements.

Broadly, this approach comprises three steps. First, an intermediate metamodel is defined that *mirrors* the database schema so that it can be easily annotated along JPA. Second, *Teneo* generates an intermediate model that conforms to this metamodel. Third, this intermediate model is mapped into a model which conforms to the desired target metamodel by using a model-to-model transformation language.

Besides being a laborious process, model transformation languages mainly focus on obtaining model elements out of other model elements. That is, the *Project* and *Degree* cases tend to be overlooked. Tackling these scenarios requires either the use of special rules (e.g., "called rules" in ATL), or resorting to helper functions (i.e., the designer needs to go down to code to express the mapping). Therefore, this last alternative of combining *Teneo* with model-to-model transformations could handle the *Project* and *Degree* cases but at the expenses of declarativeness. By contrast, a DSL for model harvesting can take semantic gaps as the norm rather than the exception, and hence, provide specific constructs to handle such gaps.

To conclude, current approaches to model harvesting either require extensive coding (subsection 2.1) or provide limited element constructors (subsections 2.2 and 2.3). Table 1 and Table 2 compare these three approaches, and set the expectations for a model harvesting DSL.

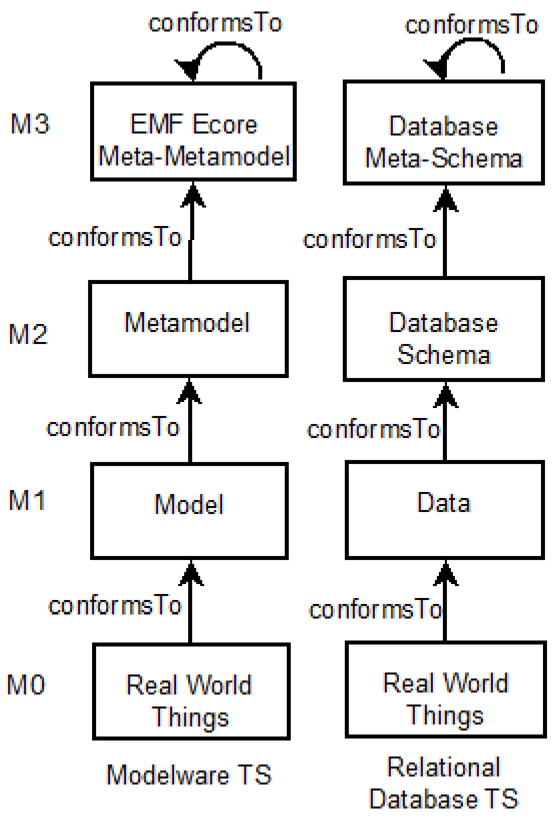


Fig. 6 Four-level model organization of database and modelware Technical Spaces.

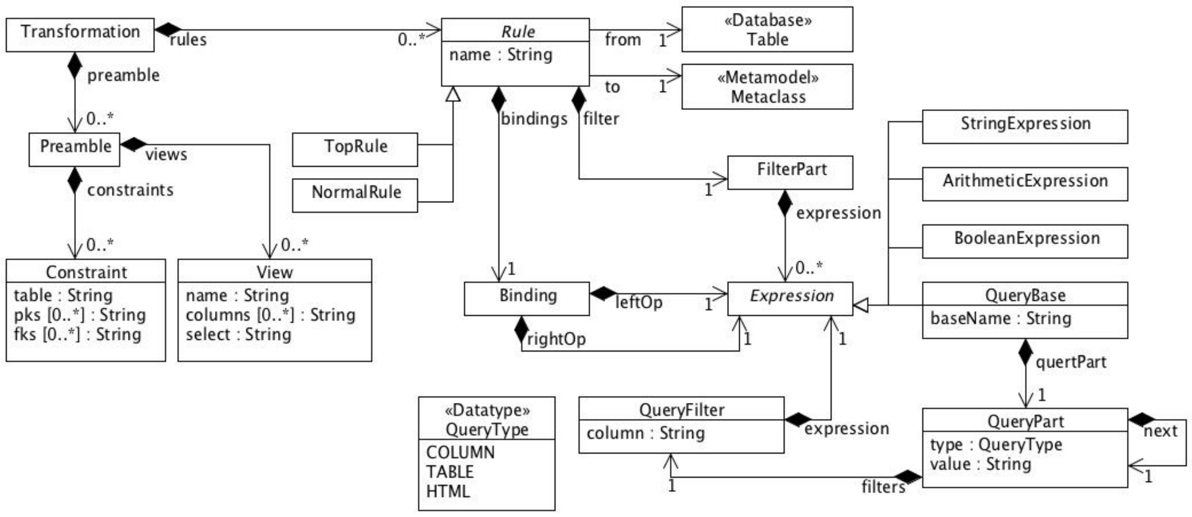


Fig. 7 *Schemol's* Abstract Syntax (Core)

```
rule <ruleName>
  from <source-databaseTable> <source-element-Id>
  to <target-metaclass> <target-element-Id>
  [ filter <expression(source-element-Id)> ]
  mapping
    { target-element-id = [ literal | query | expression ] }
end_rule
```

Fig. 8 Rule definition. Square brackets indicates optionality, and curly brackets, repetition.

3 The Schemol Language

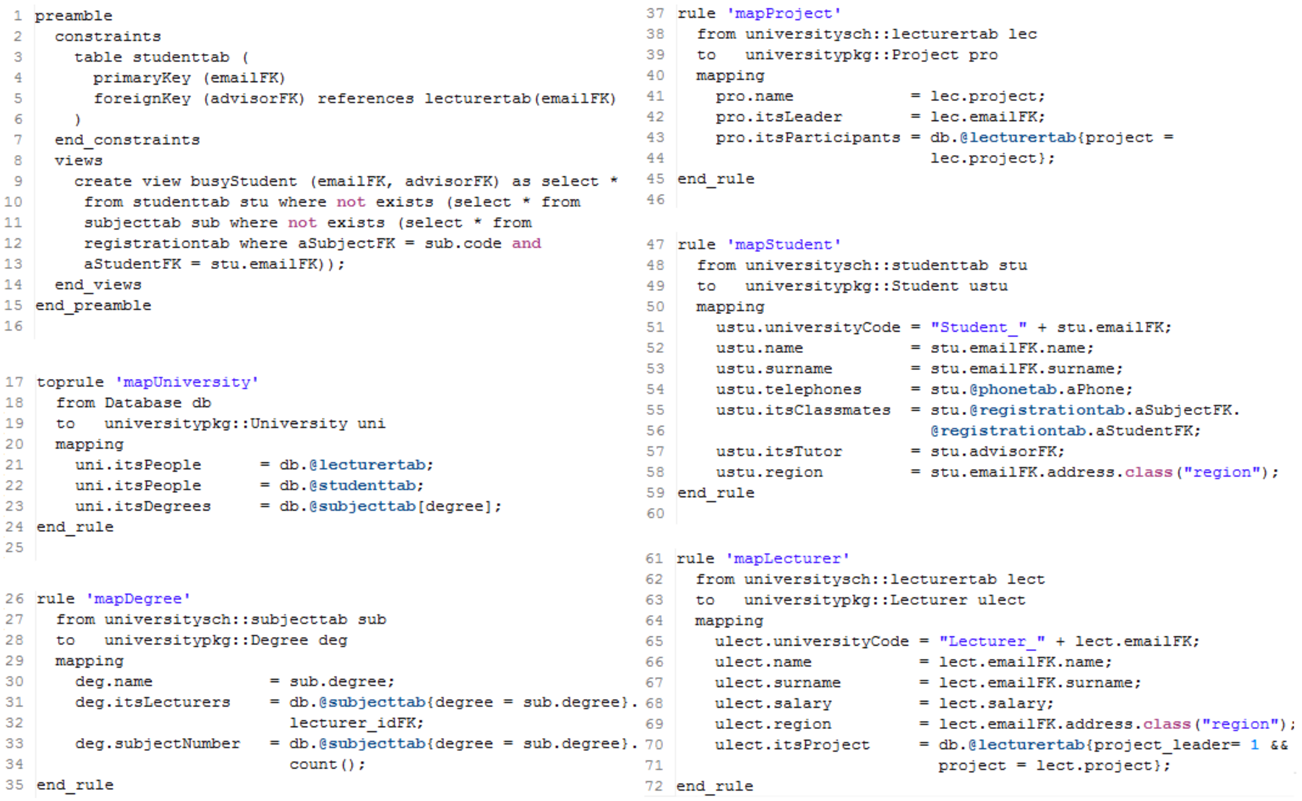
A layered architecture is commonly used to define the relationships among languages and models involved in a particular technical space (TS) [29]. The levels of such hierarchies are useful to describe the bridging between TSs. OMG's four level modeling framework [7] is one of the best known architectures. Figure 6 shows OMG's layered architecture realized for the modelware TS and the relational database TS. Following a bottom-up description, both TSs start with real-world entities (a.k.a. Universe of Discourse in database parlance). This "real world" is captured in terms of data that becomes tuples or models (layer M1). This data is collected along some established descriptions as captured by the database schemata or the metamodel (layer M2). Finally, languages exist to describe M2 artefacts (layer M3), e.g., SQL for describing database schemas, or either Ecore or EMOF to specify metamodels within the Eclipse Modeling Framework (EMF) [2].

Our bet is to capitalize on this similitude to make *Schemol* intuitive for MDE practitioners. Making *Schemol* intuitive implies following a pattern similar to that used by known transformation languages such as *ATL* [27] for model-to-model transformations or *Gra2MoL* [17] for extracting code from models.

ATL and *Gra2MoL* follow a rule-based approach. Broadly, each rule specifies how an element of the source model or code is mapped into one or several elements of the target model. Likewise, a *Schemol* rule specifies the mapping between a table of the source database schema *S,* and a class of the target metamodel *MM*. Figure 7 shows the abstract syntax for such a rule. Rules are grouped into a *transformation*. Besides rules, a transformation can also comprise a *preamble* (to be discussed in section 5. The concrete syntax of *Schemol* rules is shown in figure 8.

Each rule specifies the mapping between a database table and a class of the target metamodel. This is achieved through four clauses, namely:

1. The *from* part, which specifies the source table together with a variable that will hold the actual tuple of this table at the time the rule is enacted (i.e., <*source-element-Id*>).
2. The *to* part, which specifies the target element metaclass as well as a variable to hold the instance being generated at enactment time (i.e., <*target-element-Id*>).



```
 1 preamble
 2   constraints
 3     table studenttab (
 4       primaryKey (emailFK)
 5       foreignKey (advisorFK) references lecturertab(emailFK)
 6     )
 7   end_constraints
 8   views
 9     create view busyStudent (emailFK, advisorFK) as select *
10      from studenttab stu where not exists (select * from
11      subjecttab sub where not exists (select * from
12      registrationtab where aSubjectFK = sub.code and
13      aStudentFK = stu.emailFK));
14   end_views
15 end_preamble
16

17 toprule 'mapUniversity'
18   from Database db
19   to   universitypkg::University uni
20   mapping
21     uni.itsPeople       = db.@lecturertab;
22     uni.itsPeople       = db.@studenttab;
23     uni.itsDegrees      = db.@subjecttab[degree];
24 end_rule
25

26 rule 'mapDegree'
27   from universitysch::subjecttab sub
28   to   universitypkg::Degree deg
29   mapping
30     deg.name             = sub.degree;
31     deg.itsLecturers     = db.@subjecttab{degree = sub.degree}.
32                            lecturer_idFK;
33     deg.subjectNumber    = db.@subjecttab{degree = sub.degree}.
34                            count();
35 end_rule

37 rule 'mapProject'
38   from universitysch::lecturertab lec
39   to   universitypkg::Project pro
40   mapping
41     pro.name              = lec.project;
42     pro.itsLeader         = lec.emailFK;
43     pro.itsParticipants = db.@lecturertab{project =
44                           lec.project};
45 end_rule
46

47 rule 'mapStudent'
48   from universitysch::studenttab stu
49   to   universitypkg::Student ustu
50   mapping
51     ustu.universityCode = "Student_" + stu.emailFK;
52     ustu.name           = stu.emailFK.name;
53     ustu.surname        = stu.emailFK.surname;
54     ustu.telephones     = stu.@phonetab.aPhone;
55     ustu.itsClassmates  = stu.@registrationtab.aSubjectFK.
56                           @registrationtab.aStudentFK;
57     ustu.itsTutor       = stu.advisorFK;
58     ustu.region         = stu.emailFK.address.class("region");
59 end_rule
60

61 rule 'mapLecturer'
62   from universitysch::lecturertab lect
63   to   universitypkg::Lecturer ulect
64   mapping
65     ulect.universityCode = "Lecturer_" + lect.emailFK;
66     ulect.name           = lect.emailFK.name;
67     ulect.surname        = lect.emailFK.surname;
68     ulect.salary         = lect.salary;
69     ulect.region         = lect.emailFK.address.class("region");
70     ulect.itsProject     = db.@lecturertab{project_leader= 1 &&
71                            project = lect.project};
72 end_rule
```

Fig. 9 *Schemol* transformation for the *University* case study.

3. The *filter* part, which includes a condition expression over the source element, such that the rule will only be triggered if the condition is satisfied. This part is optional.
4. The *mapping* part, which contains a set of *bindings* to set the attributes of the target element. Bindings are based on the rule parameters (*i.e.,* source and target element ids).

A *binding* construct establishes the relationship between a source table and a target metamodel element. Normally, a mapping part contains a binding for each target metaclass' attribute. A binding is written as an assignment using the '=' operator. Its semantics is close to the binding construct offered by transformation languages such as *ATL*: specifying what needs to be transformed into what, instead of how the transformation must be performed. The left-hand side must be an attribute of the target element metaclass. The right-hand side can be a literal value, a query or an expression.

As an example, consider populating the *University* metamodel from the *University* database (see Figure 1). To this end, Figure 9 shows the *Schemol* rules that will be gradually presented throughout the paper. This example together with the *Schemol* engine are available for download at *www.onekin.org/schemol* .

We first consider the *mapStudent* rule (lines 47-59). The rule states that for each *studentTab* tuple, a *Student* model element is to be generated. The bindings indicate how the distinct student attributes are obtained from table columns. If columns are those of the FROM table then, values can be readily obtained. For instance, the binding (line 51):

ustu.universityCode = *"Student_"* + *stu.emailFK;*

obtains *universityCode* by concatenating *"Student_"* to the *emailFK* column.

However, bindings should also face two additional issues: data scattering and reference resolution. The former refers to data being disperse along different tables (e.g., student data is split across *phoneTab* o *registrationTab*). This requires a query language which will be the subject of section 4. On the other hand, reference resolution implies turning foreign keys into object references. As an example, consider the following binding (line 57):

ustu.itsTutor = *stu.advisorFK;*

Unlike *universityCode*, *itsTutor* is an association. Hence, the binding is not a mere assignment but *Schemol* should turn *advisorFK* foreign key into its reference counterpart so that *itsTutor* will stand for a model association. Reference resolution is transparently handled by *Schemol* during rule evaluation (see later).

Next, the *mapStudent* rule specifies how *Student* elements are created but it does not indicate how the rest of the *Person's* specialization is handled. This is described in the *mapUniversity* rule (line 17). This rule states *Person* model elements (i.e., the content of *itsPeople* association) are to be obtained from *lecturerTab* and *studentTab* (hence, realizing the JPA's *TABLE_PER_CLASS* strategy). Specifically, the binding in line 22 is implicitly triggering the *mapStudent* rule as a result of rules being evaluated. Next subsection presents the details.

3.1 Rule Evaluation

Rule evaluation poses three questions: how is a rule enacted?, how is a rule triggered? and finally, what makes the triggering-enactment cycle start?

Rule enactment. This process has a tuple as an input parameter (the *FROM* clause), and returns a model element (the *TO* clause). The process starts by checking the *FILTER* predicate on the tuple. If satisfied, an instance of the target metaclass is created. Once created, the instance's attributes are obtained through binding. Three situations may arise according to the nature of the right-hand side of the binding:

1. If it is a literal value then, the value is directly assigned to the attribute specified in the left-hand side.
2. If it is an expression then, the expression is evaluated. The result is directly assigned to the attribute specified in the left-hand side. *Schemol* supports string, boolean and integer expressions together with their basic operators.
3. If it is a query then, the query is executed. Depending on the type of the result and the assigning element, two situations are distinguished. If both are of a primitive type, the result of the query is directly assigned. On the other hand, if the left-hand side attribute is a reference, and the query returns a tuple set then, the engine returns the model elements of the tuple counterparts. If not yet created, returned model elements are built on the fly by triggering the appropriate rules.

Rule triggering. Binding resolution can result in rule triggering. The type of a binding is that of their sides *(leftType, rightType)*. If the right part of a binding is a query returning a set of tuples then, *leftType* is the corresponding metaclass. In this case, *Schemol* resolves bindings by looking for rules whose *FROM* and *TO* clauses conforms to the *rightType* and *leftType* of the binding, respectively. The conformance directive states that a metaclass *M1* conforms to a metaclass *M2* if they are the same or *M1* is subtype of *M2*, whereas a table *T1* conforms to a table *T2* if they are the same.

As an example, consider the binding *"ustu.itsTutor = stu.advisorFK"* (line 57). Its right-hand side is a query that returns a tuple from *lecturerTab*. Hence, the type of this binding is *(Lecturer, lecturerTab)*. This binding is resolved by the *mapLecturer* rule (line 61): its *From* and *To* clauses conform to *lecturerTab* and *Lecturer*, respectively. If the type of the right-hand side element is not a single tuple but more than one then, it will be flattened, and the rule will be applied once for every tuple. Notice that all rules matching the binding type are fired, hence producing different triggering threads. The order in which rules are triggered must not affect the final result. Rules are undeterministically triggered.

Kick-off rules. Rule enactment implies binding resolution which, in turn, can cause other rules to be triggered, and so on. This cycle is initiated by the so-called "top rules". A transformation can contain several top rules which lead to distinct binding threads. If the top rule's *FROM* is a table then, the process starts from this table's tuples. Additionally, top rules might hold a special *FROM* element, *"Database"*, that denotes the collection of all tables. This permits to generate a model element that also acts as the root for the generated model. This is quite common in some metamodels (e.g., OMG metamodels).

Back to our example, the top-rule *"mapUniversity"* (line 17) states that the database first originates a *University* element which serves as the root of the output model. This element is initialized as follows: *itsPeople* is obtained from *lecturerTab* and *studentTab,* and *itsDegrees* is derived from *subjectTab.* The type of these bindings determines the next rules to be triggered. Once top rules are enacted, the rest of the rules are implicitly triggered through binding. This implies that not all rules are necessarily fired. For instance, if the source database has no student then, the *mapStudent* rule will never be triggered.

Hidden all these details about rule evaluation is what makes *Schemol* harvesting be much more declarative that their procedural counterparts. Specifically:

1. Iteration to handle tuple collections are hidden through mappings (i.e., mapping *Table* to *Elem* is realized as "for each tuple in *Table*, creates an *Elem* instance").
2. Reference resolution is hidden. *Schemol* keeps track of the instances already created so that references are appropriately handled. No need for the user to maintain an object2model mapping table.
3. Cascading creation of model instances is implicitly handled through bindings. When the left-hand side part of a binding stands for an element reference, the corresponding transformation rule can be implicitly enacted.

This is however half of the story. Mapping constructors improve declarativeness when there exists a natural correspondence between the source database and the target metamodel. However, this correspondence might not be available: model elements can not always be linked back to database tables. In this case, *Schemol* resorts to a query language.

4 Schemol's Query Language

This section addresses the role of the query language in a model harvesting setting. Model transformation languages use query languages for *traversing* the source model. In our case, "the source model" is a database schema. Databases already have a query language: SQL. SQL supports database traversal through joins: matching between attributes of potentially different tables. The issue is how to abstract from attribute matching to provide a more declarative query language better tuned for model harvesting. This is the topic of Section 4.1.

But SQL's intent exceeds database traversal. SQL queries can also be used to adapt the database schema to the requester goals, providing a customized "view". In our setting, *Schemol* is the query requester. *Schemol*'s queries have a very specific goal: populating model elements. Therefore, Schemol *queries* can also be regarded as "views" for the description of model elements in terms of database tables. It is then important to notice that *Schemol*'s query language surpasses the role of traditional query languages in transformation languages such as *ATL*. In a model harvesting scenario, queries are not just for model traversal but also for element construction (Section 4.2 and 4.3).

4.1 Joins: Supporting Multi-tuple Elements

SQL joins collect data from two or more tables based on attribute matching. We use joins to mimic "table navigation". This implies the addition of some syntactic sugar to make attribute matching look like "table navigation". To this end, foreign keys are interpreted by *Schemol* as "references", hence, liable to be navigated through. Navigation can be forward or backward.

The expression "*fromId.colName1.colName2*" denotes a forward navigation. The query retrieves the value of *colName2,* provided *colName1* is a foreign key. Otherwise, an error is raised. For instance, the expression *"stu.emailFK.name"* (line 52) obtains the *name* of the tuple where *emailFK* is the primary key.

The expression *"fromId.@tableName"* expresses a backward navigation. This query retrieves those tuples at *tableName* table that refers to the *fromId* tuple (i.e., they have a foreign key from *tableName* table to the table that contains the *fromId* tuple). It can also be understood in forward-like terms where *tableName* is interpreted as an attribute of *fromId* that holds those tuples at *tableName* that have a foreign key over *fromId*. For instance, the expression *"stu.@phonetab.aPhone"* (line 54) obtains the set of phones at *phoneTab* for the student held by the *stu* variable.

Navigation composition is possible as illustrated by the *itsClassmates* attribute (Figure 9, lines 55-56). The expression

> *stu.@registrationtab.aSubjectFK.*
> *@registrationtab.aStudentFK;*

obtains classmates as those students that share at least one subject in common with the student at hand, including himself. Its SQL query counterpart follows:

> select distinct(*aStudentFK*) from *registrationtab* where
> *aSubjectFK* in (select *aSubjectFK* from *registrationtab*
> where *aStudentFK= primaryKeyOfTheTupleAtHand*)

Notice that this expression implies a circular reference (i.e. a student is a classmate of himself) which is avoided by the *Schemol* engine. This example also illustrates the importance of foreign keys for schema traversal. *Schemol* only admits table traversal based on foreign keys. This is in contrast with relational joins that can be based on attributes with the same type, no matter they have been declared as foreign keys (e.g. you can join tables *Person* and *Department* based on the person age and the department budget as long as age and budget are type alike).

4.2 Selections: Column-based Elements

Model element instances might not always be directly traced back to database tuples. This does not mean that these instances can not be populated from the database but that an additional predicate is required to identify the hidden entity. This case is illustrated for the *Project* element. Projects are not captured as first-class entities in the database (no *projectTab* exists). Rather, this entity is embedded as part of *lecturerTab* (through the *"project"* and *"project_leader"* attributes). We then need to indicate what hint makes us create a project element: *"a Project element exists for each lecturer being a project leader"*. That is, *Project* harvesting requires an additional predicate on *lecturerTab*.

In this case, projects can be created simultaneously with lecturers. This is the approach taken in rule *mapLecturer* (Figure 9, line 61). The *project* column is used to derive the *itsProject* association in the metamodel (lines 70-71):

ulect.itsProject = db.@lecturertab{project_leader= 1 && project = lect.project};

Unlike previous bindings, here the right-hand side expression is not a query about the actual lecturer (kept in *"lect"*) but about *lecturerTab* (this is denoted through curly brackets). The query *"lect.project"* would have returned a string rather than tuples. Queries should return tuples for the binding to obtain element references. Previous expression returns *lecturerTab* tuples, hence the binding is resolved by the *mapProject* rule (line 37). Basically, the binding indicates that a project should be created for each project leader (i.e., *"project_leader= 1"*). Notice that the binding not only creates a project but also associates this project with its participants. This explains the second condition: *"project = lect.project"* which ensures that the current lecturer (kept in *"lect"*) participates in the project at hand.

4.3 Aggregations: Collection Elements

Some model elements can stand for collections. These collections might be represented through their members when in the database. For instance, consider that *lecturerTab* would not hold the *"project_leader"* attribute. We would have then re-written the query to identify projects as follows: *"a Project element exists for each set of lecturers sharing the same project"*. The binding would have looked as:

ulect.project = db@lecturertab[project]{project= lect.project}

In this case, square brackets are used to denote the SQL's GROUP BY clause. The SQL query counterpart will look like as *(SELECT * FROM lecturerTab WHERE project = "aCertainProject" GROUP BY project)* The query returns just a single tuple that stands for the project of the current lecturer (this explains the additional condition). It is worth noticing that if several lecturers participate in the same project, the project element is created just the first time, and from then on, this model element ID is returned as a result of the binding.

The *Degree* example provides an interesting twist to this situation. In the previous case, the "pivotal entity" (i.e., *lecturer*) is also captured by the target metamodel (i.e., the *Lecturer* element). By contrast, now the pivotal entity, i.e., *subject*, is not reflected in the metamodel. Rather, degrees directly hang from *University*. Therefore, the *mapUniversity* rule (line 17) directly creates *Degree* elements from *subjectTab* through the binding (line 23):

uni.itsDegrees = db.@subjecttab[degree]

The right-hand side expression is a query where subjects are grouped by degree. The query returns a sample tuple for each group. This basically implies that a *Degree* element is created for each group. The type of the binding is *(Degree, subjectTab)*. This binding is resolved by the *mapDegree* rule (line 26). Besides the name of the degree, the rest of the attributes are either literals or obtained through queries. The rule illustrates the case for *itsLecturers* and *subjectNumber*. The former is worked out from the lecturers teaching a subject included in this degree. As for *subjectNumber*, the function *COUNT* is used to obtain the total number of subjects in the degree at hand[3].

5 Schemol's Preamble

Most model-to-model transformation languages permit declarative rules to be mixed with imperative actions for practicality. Helper functions are offered in *ATL* as a back door to code when declarative rules are not expressive enough to capture the transformation. Likewise, the *transformation*

[3] Other aggregate functions include: *SUM, MIN, MAX* or *AVG*

preamble is the last resort to adapt the database schema to the mapping needs. Such gap might stem from both distinct conceptual models and distinct processing needs. The latter deserves further explanation.

Process optimization can lead to different ways to capture data. Normalization is a common case, but it is not the only one. As a real example, consider *Blojsom*. Since *Blojsom* has been largely tested, efficiency considerations advice to overlook foreign-key constraints. After all, *Blojsom* programs are the only ones that are supposedly accessing this database. If constraints are consistently respected by the code, and the database undergoes frequent updates then, removing constraints avoids the overhead of the extra conditionals. This is a common pattern in most application-proprietary databases. From our perspective, this implies that you lack a fundamental input to derive model associations. Indeed, *Schemol* relies on foreign keys to traverse the table space. If a foreign key is not declared in the catalog, *Schemol* will raise an error when attempting to navigate along this attribute.

A *preamble* permits the definition of *ad-hoc* foreign keys and views. In the JDBC approach (see Section 2.1), SQL scripts traverse the tables through joins. It is expected these joins to be based on equalities between primary keys and foreign keys, but this is not compulsory. Joins can be based on any pair of attributes as long as their types are compatible. By contrast, *Schemol* requires navigation to be based on explicitly defined foreign keys so that traversal is "safe": you ensure your navigation ends in an entity. However, foreign keys can be removed for optimization purposes, hence, preventing *Schemol* from navigating along these attributes. The *preamble* is a way to complete the schema without touching the schema itself (so without interfering with the database) so that hidden foreign keys can be surfaced.

As an example, consider *advisorFK* is not defined as a foreign key. This implies that expressions such as *"stu.advisorFK"* will be interpreted as returning a literal rather than a primary key, hence, blocking the triggering of the *mapLecturer* rule. In this case, the *preamble* extends the database schema by specifying *advisorFK* as a foreign key (see Figure 9, lines 3-6). Now, transformation rules can appropriately handle *advisorFK* as a foreign key.

As for views, they can be defined to describe the database schema in terms closer to those needed for the transformation. This would facilitate rule specification, comprehension and maintenance. Notice however, that the database is kept unpolluted of these spurious views (i.e., not backed by data processing needs) whose only purpose is to facilitate the transformations. The preamble is part of the *Schemol* language but based on SQL syntax.

For illustration purposes, consider our target metaclass would hold the *busyStudent* element. This element stands for those students registered in all courses. In relational algebra terms, this is a division. *Schemol*'s query language lacks the expressive power of relational algebra. In this case, the designer can resort to define a view (Figure 9, lines 9-13). Once defined, views can be used as any other "database table" during the transformation process. That is, rules can now be defined having *busyStudent* in the *FROM* clause.

6 Exploiting Annotated Content for Element Creation

Web2.0 databases tend to have annotated content. Although transparent to the database, annotations might need to be surfaced in the target metamodel. This section first makes the case by introducing common approaches to annotated content in Web2.0. Next, we introduce new functions in the *Schemol* query language to surface these annotation-based elements.

6.1 Self-descriptive Content

It is common for content to be mixed up with hints about how this content is to be presented, structured or understood. This section outlines three common mechanisms: HTML classes, microformats and RDF annotations.

Although used mainly for rendering, HTML tags can provide hints useful during the transformation. The HTML *class* attribute is a common example. As advised by the W3C, "often people use class names like *bluetext*, or *redborder*. A much better way to name your classes is with the role a certain HTML element of that class has." [13]. This very much refers to the existence of a concept that might be surfaced in the target metaclass.

More to the point, the usefulness of the *class* attribute goes beyond rendering to increase the semantic fidelity of HTML through microformats [9]. Broadly, microformats can be regarded as standardized *class* names to describe the data being markuped (a.k.a. semantic markup). For instance, if the data refers to your name then, you can explicitly state so by using the *fn* class, part of the *hCard* microformat [3], (e.g., `<h3 class="fn">Barak</h3>`). But microformats are only one way to semantic markup. Microformats are often referred to as the grassroot approach to Semantic Web where RDFa is being proposed [10]. RDFa is a W3C specification for attributes to express structured data in any markup language. RDFa often uses a subject, predicate, object combination called a triple to specify an attribute name/value pair about a particular resource. By default, the resource refers to the content being markuped. For instance, the markup `<h1><span property="dc:title"> Harvesting models </span></h1>`, indicates that *"Harvesting models"* stands for the Dublin Core *title* of this markup. In RDFa, HTML's "property" is used to denote the role played by the content. Authors are then encouraged to semantic markup their content. And this includes also the layman. As a result, blog engines are offering plugins that facilitate the inclusion of microformats (e.g., *StructuredBlogging* available for *Movable Type* and *Wordpress* [11]).

Previous observations highlight how user content can embed markup hints which are liable to be surfaced in the target model. To prevent designers from having to do the digging themselves, the *Schemol* query language introduces the following extraction functions:

1. *Class(HTMLclassName),* which applied to a string, recovers the content of the element whose class is *HTMLclassName.*
2. *Property(HTMLpropertyName)*, which applied to a string, outcomes the content of the element whose property is *HTMLpropertyName.*

Extraction functions can recover either atomic values or tuples. For instance, the expression *"anAttribute. class('address')"* recovers the content of those HTML elements of class *"address"* in the markup held as the value of *"anAttribute"*. If this content is atomic, the function behaves as a kind of derived attribute by yielding a set of addresses. However, this content can also embed full-fledged entities (e.g., the class *"address"* comes with other companion classes such as *"street", "zip" "cite"*, etc). In this case, *"class('address')"* would recover a set of tuples where companion classes become attributes of the so-generated tuples. Now, extraction functions are regarded as the realization of nested tables [40]. Nested tables are introduced in the object-relational model to denote a table that is embedded within another table. This is similar to our case where a tuple can contain attributes that hold other "embedded tuples". Despite being embedded, nested tables need to be named. *Schemol* automatically names nested table after the *className* of the extraction function plus the *EMB* prefix (e.g., *EMBaddress*). The very same behaviour applies to the *property()* function where "nested tables" are extracted from companion properties. The combined use of classes and properties in the very same markup has not been considered yet.

The bottom line is that *Schemol* handles "real tuples" (those directly extracted from the database) and "embedded tuples" (i.e., those obtained from the markup) in the very same way. Next subsections present these functions at work.

6.2 Case Study: Blog Entries as Product Elements

Initially thought as personal diaries, blogs' scope has broadened to become a medium for professionals to communicate. Distinct studies endorse the use of blogs to market products, build stronger relationships with customers, and obtain customer feedback [21,30,35]. Here, blogs become on-line catalogs, where blog entries (a.k.a. posts) describe a product, and customers can comment on these entries.

Source database schema. *Blojsom* is used as the blog engine [19]. A partial view of its database schema is depicted in Figure 10, and includes the following tables: *Entry*, which keeps the blog entries; *Trackback*, which basically indicates links between entries; *Category*; which stores how entries are classified along a set of categories. It is worth noticing that the entry content is stored as the atomic value of the *Entry's description* attribute whose type is *mediumtext.* In this example, this content is annotated along the *hproduct* microformat [5]. As stated by the proponents "if a browser or application understands that a specific web page contains the *hProduct* microformat, it can inform and help facilitate the best product choice by the consumers to fulfil their needs. Bringing order and structure to product data makes it easy for people and organizations to list the products and services they offer, and easy for web services to consume that product data for analysis and syndication" [5]. Figure 11 shows one such *description* and its HTML counterpart. This information can be available for browsers (that understand HTML) but it is transparent to the DBMS which just stores it as a mere string.

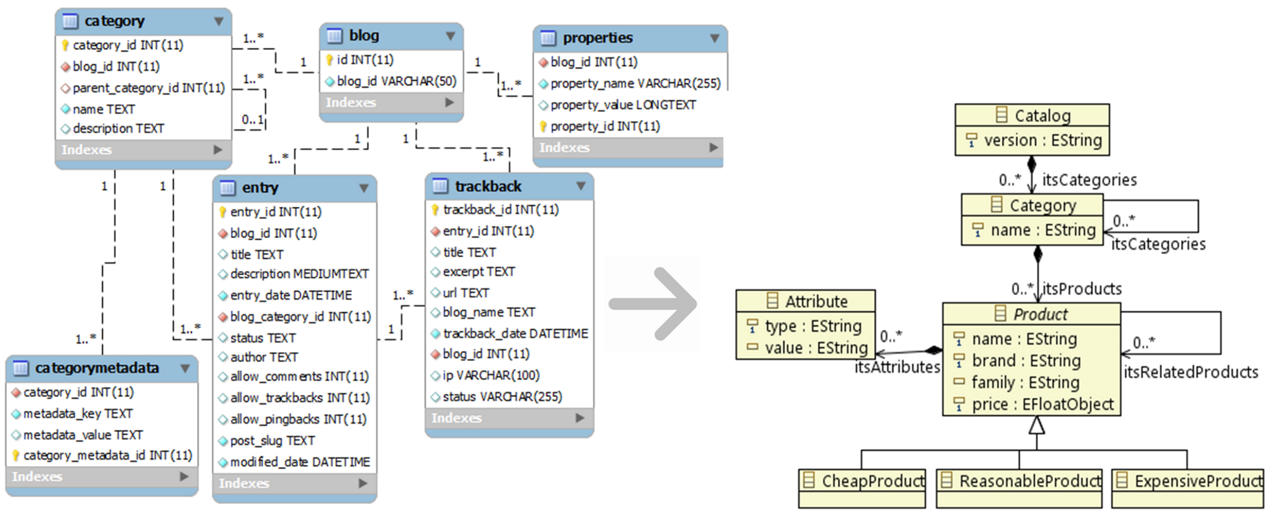
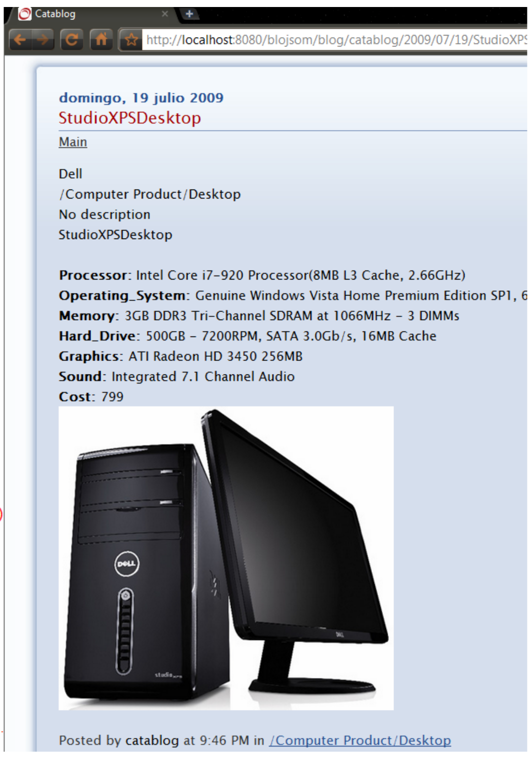


Fig. 10 *Blojsom* database schema (left) & *Catalog* metamodel (right).

```
<div id="StudioXPSDesktop" class="hproduct">
  <span class="brand">
    Dell
  </span>
  <br/>
  <span class="category">
    /Computer Product/Desktop
  </span>
  <br/>
  <span class="description">
    No description
  </span>
  <br/>
  <span class="fn">
    StudioXPSDesktop
  </span>
  <br/>
  <span class="identifier">
    <span class="type">
      <strong>
        <br/>
        Processor:
      </strong>
    </span>
    <span class="value">
      Intel Core i7-920 Processor(8MB L3 Cache, 2.66GHz)
    </span>
  </span>
  <br/>
  <span class="identifier">
    <span class="type">
      <strong>
        Operating_System:
      </strong>
    </span>
    <span class="value">
      Genuine Windows Vista Home Premium Edition SP1, 64
    </span>
```


Catablog

http://localhost:8080/blojsom/blog/catablog/2009/07/19/StudioXPS

domingo, 19 julio 2009

StudioXPSDesktop

Main

Dell
/Computer Product/Desktop
No description
StudioXPSDesktop

Processor: Intel Core i7-920 Processor(8MB L3 Cache, 2.66GHz)
Operating_System: Genuine Windows Vista Home Premium Edition SP1, 6
Memory: 3GB DDR3 Tri-Channel SDRAM at 1066MHz – 3 DIMMs
Hard_Drive: 500GB – 7200RPM, SATA 3.0Gb/s, 16MB Cache
Graphics: ATI Radeon HD 3450 256MB
Sound: Integrated 7.1 Channel Audio
Cost: 799

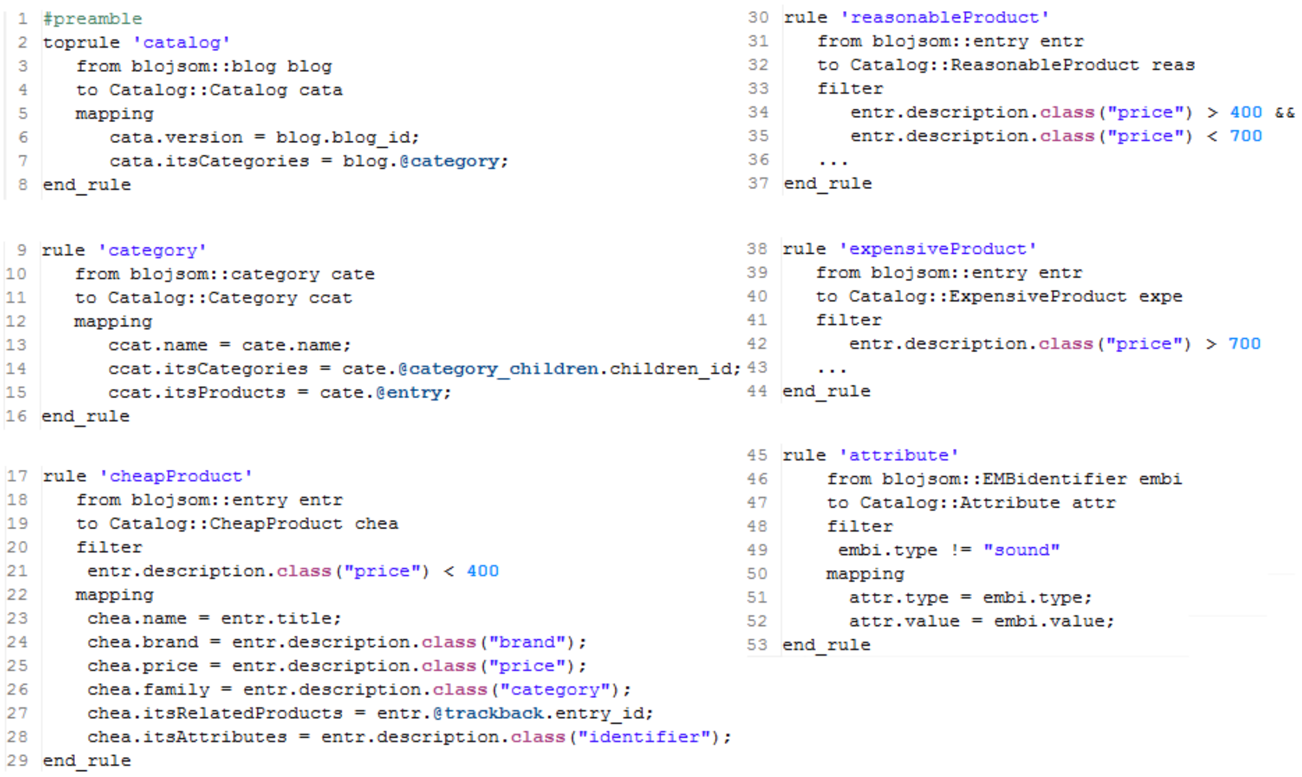
Posted by catablog at 9:46 PM in /Computer Product/Desktop

Fig. 11 A blog entry and its HTML counterpart. The HTML markup is annotated with the *hproduct* microformat.

Target metamodel. A catalog model is introduced that collects information about products, qualified by distinct categories (see Figure 10). Models for catalog description have been proposed by distinct organizations such as the *Open Catalog Format* (OCF)[32], an open standard to represent, store, and transport catalog descriptions.

Schemol transformation. A database-to-model transformation is defined to obtain catalog models out of *Blojsom* databases. In this case however, an important part of the semantics is not on the physical database but blurred inside the entry description along the *hProduct* microformat. The challenge is how to surface this structure so that it can be used during model transformation.

Schemol's *class()* function helps designers in pinpointing the HTML classes/properties of interest while leaving to the *Schemol* engine the tedious process of extracting this data from the raw text. Figure 12 illustrates this point. Properties for *cheapProduct* elements are obtained from



```
1  #preamble
2  toprule 'catalog'
3     from blojsom::blog blog
4     to Catalog::Catalog cata
5     mapping
6        cata.version = blog.blog_id;
7        cata.itsCategories = blog.@category;
8  end_rule

9  rule 'category'
10    from blojsom::category cate
11    to Catalog::Category ccat
12    mapping
13       ccat.name = cate.name;
14       ccat.itsCategories = cate.@category_children.children_id;
15       ccat.itsProducts = cate.@entry;
16 end_rule

17 rule 'cheapProduct'
18    from blojsom::entry entr
19    to Catalog::CheapProduct chea
20    filter
21     entr.description.class("price") < 400
22    mapping
23     chea.name = entr.title;
24     chea.brand = entr.description.class("brand");
25     chea.price = entr.description.class("price");
26     chea.family = entr.description.class("category");
27     chea.itsRelatedProducts = entr.@trackback.entry_id;
28     chea.itsAttributes = entr.description.class("identifier");
29 end_rule
```

```
30 rule 'reasonableProduct'
31    from blojsom::entry entr
32    to Catalog::ReasonableProduct reas
33    filter
34       entr.description.class("price") > 400 &&
35       entr.description.class("price") < 700
36    ...
37 end_rule

38 rule 'expensiveProduct'
39    from blojsom::entry entr
40    to Catalog::ExpensiveProduct expe
41    filter
42       entr.description.class("price") > 700
43    ...
44 end_rule

45 rule 'attribute'
46     from blojsom::EMBidentifier embi
47     to Catalog::Attribute attr
48     filter
49      embi.type != "sound"
50     mapping
51       attr.type = embi.type;
52       attr.value = embi.value;
53 end_rule
```

Fig. 12 *Schemol* transformation for the *Blog* case study.

entries in the blog (lines 17-29). However, the entry is stored as a string by the *description* attribute of the *entry* table. Applying the function *class* to *description* permits to recover the value of the *class*-qualified content. The expression *entr.description.class('brand')* applies function *class*, parametrized with *"brand"* to the content of the current *description* (line 24). Basically, the function recovers the content of those HTML elements whose class is *"brand"* (*class* = *"brand"*).

Additionally, *hProduct* proponents left open the possibility of adding additional product attributes. The *'identifier'* class is provided for this purpose. This class describes *ad-hoc* attributes along its *'type'* and *'value'*. Since this class has a complex content model, *Schemol* interprets *'identifier'* as a nested table with these two attributes. The binding in line 28 causes the triggering of the *'attribute'* rule. Notice the *'from'* clause that refers to the nested table as named by *Schemol* after the class name: *EMBidentifier.*

7 Implementation

Several tools exist for the definition of textual DSLs, which create a bidirectional bridge between grammarware and modelware technical spaces (e.g., xText [14] and EMFText [23]). These tools allow defining a concrete syntax for the DSL metamodel, and generating a DSL toolkit: DSL editors, extractors (a.k.a. injectors) of models from DSL specifications and generators of DSL specifications from models. As we are only interested in extracting models from *Schemol* textual definitions (i.e. the injector component), we have used *Gra2MoL* [17], mainly due to our knowledge of the tool.

The architecture of the Schemol engine includes three main components (see Figure 13):

1. *The Schemol interpreter*, which behaves as a rule engine for enacting and keeping the transformation flow. The interpreter also conducts the bindings by querying the database (right-hand side operand) and creating the model elements (left-hand side operand).
2. *The SQL generator*, which maps queries expressed using the dot notation into SQL joins.
3. *The model generator*, which handles petition for element construction.

Figure 13 illustrates the execution process of a *Schemol* transformation. Steps 1 to 3 perform the extraction of the *Schemol* model from the textual definition. Once the *Gra2MoL* engine injects the *Schemol* model, the *Schemol*

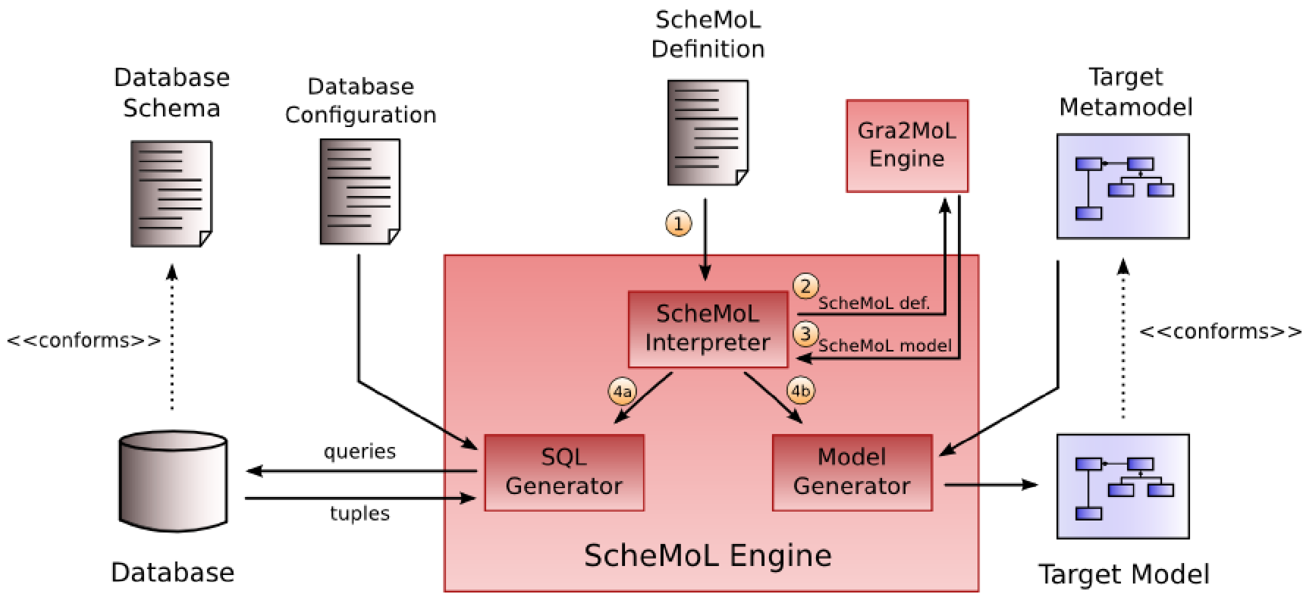


Fig. 13 *Schemol* Implementation.

Interpreter executes the transformation rules, which in turn, entail database querying and model generation (steps 4a and 4b). Therefore, the *Schemol* Engine has three inputs: the transformation definition itself, the target metamodel and the database connection configuration.

8 Discussion

This section reviews *Schemol* along the ISO 9126 Software Quality Model [6]. The revision identifies strengths but also limitations of the current approach. The latter opens new ways to improvement.

Suitability. *Schemol* constructs must correspond to relevant model-harvesting concepts. To this end, six scenarios have been introduced for model harvesting in the presence of wide conceptual gaps. These scenarios serve to pinpoint two main constructors for harvesting: mappings and views. Furthermore, extraction functions are introduced to extract model elements out of embedded markup. At this respect, DSL **extensibility** is also a desirable feature. So far, *Schemol* permits to add new extractive functions to accommodate further advances in semantic markup.

Learnability. This usability subcharacteristic refers to the learning effort required to use *Schemol*. This has been a main concern during *Schemol* design. We strive to mimic as much as possible model-to-model transformation languages so that database-to-model harvesting looks like model transformation in *ATL*. The presumption is that this similarity reduces the learning curve, and eases adoption by the MDE community. Although the syntactic similitude is obvious, this presumption still needs to be proven through usability experiments. This characteristic is closely related with **supportability**, i.e. the ability to provide DSL support via tools, for typical model and program management, e.g., creating, deleting, editing, debugging, transforming, etc. So far, *Schemol* transformations are just text files. Error messages are those propagated from the underlying *MySQL* and *Java/EMF* APIs (e.g., no existence of a column name or an element attribute). Further work is required to also report errors in more abstract terms.

Efficiency. We do not consider time to be a main limitation since model harvesting is sporadically conducted without requiring to stop the source DBMS. Nevertheless, resource usage, specifically, storage of output models can be an issue in some modeling frameworks. Databases can be very large, and hence, model harvesting from databases can potentially result in large models. So far, frameworks such as EMF do not scale with larger number of models. So far, *Schemol* has been used to medium-size databases as those of wikis and blogs that commonly hold below ten thousand articles (or blog entries). Notice however, that model harvesting does not necessarily impose the complete database to be moved to the model realm. In some scenarios, it is enough to capture a subset of the database.

Interoperability. This subcharacteristic concerns the ability of a software component to interact with other components or systems. *Schemol* does not work in isolation.

Rather, it interacts with two technology realms: DBMSs and modeling frameworks. So far, *Schemol* supports *MySQL* and *EMF* as the DBMS and modeling framework, respectively. Drivers need to be developed for *Schemol* to allow for a broad range of options. Additionally, while relational schema description has been standardized by *ANSI*, this has not been the case for metamodel description. So far, *Schemol* supports one of the most popular formats, *Ecore,* but other formats can be easily added, should this be required.

9 Related Work

This work sits in the intersection between data re-engineering and model harvesting. Next paragraphs outline some related works in these areas.

Data re-engineering. Reverse engineering is "the process of taking software artefacts, discarding implementation decisions, and extracting the underlying conceptual content" [16]. Reverse engineering might be applied to both code and data. Data reverse engineering provides a collection of methods and tools to determine the structure, function, and meaning of the system's data [20]. This community has long struggle for extracting conceptual designs out of legacy databases [34]. Basically, the activity is split into two major tasks: Data Structure Extraction and Data Structure Conceptualization. The former aims to recover an up-to-date logical data model, while conceptual abstraction aims at mapping this logical data model to an equivalent conceptual design. Unlike model harvesting, here the target data model is unknown. Similarly to model harvesting, Data Structure Extraction is challenged by the fact that important information about the data model can be missing in the DBMS physical schema catalog. Although different patterns exist, this extraction process has to deal with uncertain assumptions and heuristics about legacy data models which makes extraction difficult to algorithmisize, and far from being a one-shot process. Rather, different authors advocate for iterative processes that incrementally guide the engineer to a consistent data model [25]. Additionally, rather than general-purpose algorithms, scripting languages have been proposed [26] that permit the designer to customize the reverse engineering process. The latter seems to suggest the use of "transformations" as means to explicitly state the mappings rather than being hidden in an algorithm. Finally, [36] proposes an MDA approach to data re-engineering. This basically means that intermediate artefacts are UML models. However, as target meta-models are unknown, "transformations" are embodied as algorithms that code heuristics to ascertain target models out of source models.

Moving from migration to integration of data sources, Extract-Transform-Load (ETL) processes constitute the back stage of Data Warehouse architectures. Here, the tracing of the existing data sources and the understanding of their hidden semantics is a main challenge. In a recent work [38], the use of an ontology is proposed to formally and explicitly capture the semantics of the involved data stores. The ontology does not exist *a priori*, and requires designers to explicitly indicate mappings between the data sources to be integrated. Once the ontology is in place, it consequently guides the generation of the ETL process. The similitude with our work rests on the use of ontologies as a kind of metamodels to guide the generation process. From this perspective, ontologies play a similar role to our target meta-model as "guidance" of the transformation process.

A transformational approach to migrate legacy data-intensive applications is reported in [22]. During the data conversion process, the physical schema is first extracted by parsing the DDL code. Next, the process continues by mapping the source to the target physical schemas. This is achieved through a chain of transformations which involves both program code analysis and analysis of database contents. This proposal does not use MDE techniques for implementing these transformations, and extraction is conducted programmatically. By contrast, *Schemol* advocates for an MDE approach by providing a DSL for model harvesting.

Model harvesting. The interest in model harvesting should be sought in the increasing upheaval on software modernization, specifically when MDE techniques are used. OMG's Architecture Driven Modernization (ADM) initiative [1] aims to facilitate the interoperability between modernization tools by defining a set of standard metamodels for representing information involved in a software modernization process. In fact, the metamodel provided by ADM to represent software artefacts, named KDM, includes persistent data as one of the assets to be captured. In this scenario, model harvesting is a key enabler as the first step to abstract for technological platforms, and building effective bridges between technical spaces (TS).

For the XML TS, the *Eclipse Modeling Framework* provides harvesting utilities [39]. This framework generates metamodels out of *XML Schemas* where extracted models conform to the generated metamodel. The main advantage of this approach is also its main limitation: generality and automaticity. Tuning the output model requires an additional transformation which might potentially incur in an important efficiency penalty.

For programming code, *Gra2MoL* [17] offers a bridge between grammarware and modelware. Extracting models out of code is commonly handled by building *ad-hoc* processors, which first parse the source text to obtain a syntax tree and then, create the target model by querying this syntax tree. This tedious and procedural approach is faced in *Gra2MoL*, a DSL for extracting models out of code using a rule-based approach based. From this perspective, *Schemol* has a clear parallelism to *Gra2MoL*. However, *Schemol* offers a bridge between dataware and modelware while addressing Web2.0 database specifics.

MoDisco (Model Discovery) is an extensible framework for model-driven reverse engineering, supported as an Eclipse Generative Modeling Technology (GMT) component [15]. Its objective is to facilitate the development of tools (“discoverers” in *MoDisco* terminology) to obtain models from legacy systems during modernization efforts. XML and Java discoverers are available. *Schemol* can then become an additional *MoDisco* discoverer for relational data.

10 Conclusions

As MDE becomes mainstream, model harvesting gains prominence as a key enabler to bridge other technical spaces to modelware. This paper introduced a DSL for model harvesting out of databases, with special focus on Web2.0 application databases. The result, *Schemol*, is a rule-based transformation language that regards database schema as deprecated models due to normalization considerations. This perspective permits to describe model harvesting as a model transformation, hence easing adoption among MDE practitioners. As far as we are aware of, this work is the first that provides this perspective.

As future work, there are several challenging threads to pursue. First, we plan to improve the interoperability of *Schemol* by facilitating drivers for other DBMS and meta-model languages. Also, we would like to conduct experiments on the usability of *Schemol*. Finally, another interesting direction is to tap on existing HTML extractive methods rather than building our own in *Schemol*.

Acknowledgments This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE); Fundación Séneca (Spain), grant 08797/PI/08 and the University of the Basque Country under contract U07/09. Javier Luis Cánovas Izquierdo enjoys a doctoral grant from the Fundación Séneca and Gorka Puente has a doctoral grant from the Spanish Ministry of Science & Education.

References

1. Architecture-Driven Modernization (ADM). Online; adm.omg.org [accessed 21-Dec-10]. 1, 9
2. Eclipse Modeling Framework. Online; www.eclipse.org/modeling/emf [accessed 21-Dec-10]. 3
3. hCard Microformat. Online; microformats.org/wiki/hcard [accessed 21-Dec-10]. 6.1
4. Hibernate. Online; www.hibernate.org [accessed 21-Dec-10]. 1, 2.2
5. hProduct Microformat. Online; microformats.org/wiki/hproduct [accessed 21-Dec-10]. 6.2
6. ISO 9126 Software Quality Model. Online; www.sqa.net/iso9126.html [accessed 21-Dec-10]. 8
7. MDA Specifications. Online www.omg.org/mda/specs.htm. 3
8. MediaWiki. Online; www.mediawiki.org [accessed 21-Dec-10]. 1
9. Microformats. Online; microformats.org [accessed 21-Dec-10]. 1, 6.1
10. Rdfa. Online; rdfa.info/wiki/Introduction [accessed 21-Dec-10]. 6.1
11. Structured Blogging. Online; structuredblogging.org/ [accessed 21-Dec-10]. 6.1
12. Teneo. Online; wiki.eclipse.org/Teneo [accessed 21-Dec-10]. 1, 2.3
13. Use Class With Semantics in Mind, W3C. Online; www.w3.org/QA/Tips/goodclassnames [accessed 21-Dec-10]. 6.1
14. XText. Online; www.eclipse.org/Xtext/[accessed 21-Dec-10]. 7
15. Gabriel Barbier, Hugo Bruneliere, Frédéric Jouault, Yves Lennon, and Frédéric Madiot. Modisco, A Model-Driven Platform to Support Real Legacy Modernization Uses Cases. In *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Elsevier Science, 2010. 9
16. Michael R. Blaha. On Reverse Engineering of Vendor Databases. In *Working Conference on Reverse Engineering (WCRE)*, pages 183–190, 1998. 9
17. Javier Luis Cánovas, Jesús Sánchez Cuadrado, and Jesús García Molina. Gra2MoL: A Domain Specific Transformation Language for Bridging Grammarware to Modelware in Software Modernization. In *MODSE 2008*, 2008. 3, 7, 9

18. Steve Cook. Domain-Specific Modeling and Model Driven Architecture. *MDA Journal*, 2004. Online at www.bptrends.com/publicationfiles/01-04kel-Cook.pdf (last accessed Oct 2010). 1
19. David Czarnecki. Blojsom. Online; `wiki.blojsom.com` [accessed 21-Dec-10]. 1, 6.2
20. Kathi Hogshead Davis and Peter H. Aiken. Data Reverse Engineering: A Historical Survey. In *Working Conference on Reverse Engineering (WCRE)*, pages 70–78, 2000. 9
21. Oscar Díaz and Felipe M. Villoria. Generating Blogs out of Product Catalogues: An MDE Approach. *Journal of Systems and Software*, 83(10):1970–1982, 2010. 6.2
22. Jean-Luc Hainaut, Anthony Cleve, Jean Henrard, and Jean-Marc Hick. Migration of Legacy Information Systems. In Mens and Demeyer [33], pages 105–138. 9
23. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *ECMDA-FA*, pages 114–129, 2009. 7
24. Javier Luis Cánovas Izquierdo and Jesús García Molina. An Architecture-Driven Modernization Tool for Calculating Metrics. *IEEE Software*, 27:37–43, 2010. 1
25. Jens H. Jahnke. Cognitive Support in Software Reengineering Based on Generic Fuzzy Reasoning Nets. *Fuzzy Sets and Systems*, 145(1):3–27, 2004. 9
26. Jens H. Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic Fuzzy Reasoning Nets as a Basis for Reverse Engineering Relational Database Applications. In *ESEC / SIGSOFT FSE*, pages 193–210, 1997. 9
27. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like Transformation Language. In *OOPSLA Companion*, 2006. 3
28. Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, pages 128–138, 2005. 1
29. Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces: An Initial Appraisal. In *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002. 3
30. Nicholas S. Lockwood and Alan R. Dennis. Exploring the Corporate Blogosphere: A Taxonomi for Research and Practice. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences - HICSS*, 2008. 6.2
31. Benjamin Markines. Socially induced semantic networks and applications. *SIGWEB Newsl.*, pages 3:1–3:3, September 2009. 1
32. MartSoft. Open Catalog Format. Online; `xml.coverpages.org/ocp.html` [accessed 21-Dec-10]. 6.2
33. Tom Mens and Serge Demeyer. *Software Evolution*. Springer, 2008. 1, 22
34. Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse Engineering: a Roadmap. In *International Conference on Software Engineering (ICSE)*, pages 47–60, 2000. 9
35. Carr N. Lessons in Corporate Blogging, 2006. Business Week Online `www.businessweek.com` [accessed 21-Dec-10]. 6.2
36. Macario Polo, Ignacio García Rodríguez de Guzmán, and Mario Piattini. An MDA-Based Approach for Database Re-Engineering. *Journal of Software Maintenance*, 19(6):383–417, 2007. 9
37. Thijs Reus, Hans Geers, and Arie van Deursen. Harvesting Software Systems for MDA-Based Reengineering. In *ECMDA-FA*, pages 213–225, 2006. 1
38. Alkis Simitsis, Dimitrios Skoutas, and Malú Castellanos. Representation of Conceptual ETL Designs in Natural Language Using Semantic Web Technology. In *Data & Knowledge Engineering*, 2009. 9
39. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2008. 2.1, 9
40. Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1996. 6.1
41. William M. Ulrich and Philip H. Newcomb. *Information Systems Transformation : ADM Case Studies*. Morgan Kaufmann, 2010. 1