A Framework for Modeling, Building and Maintaining Enterprise Information Systems Software

Alexandre Cláudio de Almeida[1], **Glauber Boff**[1], **Juliano Lopes de Oliveira**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Caixa Postal 131, Campus II, CEP 74.001-970 - Goiânia - GO - Brasil

{alexandre,glauber,juliano}@inf.ufg.br

Abstract. *An Enterprise Information System (EIS) software has three main aspects: data, which are processed to generate business information; application functions, which transform data into information; and business rules, which control and restrict the manipulation of data by functions. Traditional approaches to EIS software development consider data and application functions. Rules are second class citizens, embedded on the specification of either data (as database integrity constraints) or on the EIS functions (as a part of the application software). This work presents a new, integrated approach for the development and maintenance of EIS software. The main ideas are to focus on the conceptual modeling of the three aspects of the EIS software - application functions, business rules, and database schema - and to automatically generate code for each of these software aspects. This improves software quality, reducing redundancies by centralizing EIS definitions on a single conceptual model. Due to automatic generation of code, this approach increases the software engineering staff productivity, making it possible to respond to the continuous changes in the business domain.*

1. Introduction

Modern organizations make intensive use of Enterprise Information Systems (EIS) to manage and control increasingly complex business processes and data in order to support both operational and decision making processes [Debevoise 2007]. EIS support business processes by mediating the flow of information between the actors in the enterprise and providing accurate information to improve these actors performance.

The development of EIS software requires thorough understanding of business domain and business rules (BR). The ability to connect business processes and business rules, making quality information available on time, defines the real value of the EIS software.

The construction and evolution of EIS software capable of providing this ability have challenged the Software Engineering community for many years. In spite of the advances in the software technology, which eliminated a lot of incidental problems, EIS software is still built using the traditional approach: software requirements, business rules, and data models are separately analyzed, designed, and implemented. The object-oriented approach was not able to solve this problem since persistence and performance issues frequently lead to the division of software responsibilities between the application software and the underlying database system, with business rules been embedded on one of both of these components.

While current EIS software is able to attend immediate business needs, it is very difficult to evolve or to adapt this software to the continuous changes in the business domain, since the concepts of this domain are spread among different components.

In this paper we present a software framework that solves this difficulty by adopting a model based approach to EIS software development. In this approach, the application software, the business rules, and the underlying database schema are generated from a single conceptual model of the business domain concepts. Thus, evolving or adapting the EIS software to new business requirements is restricted to changing the conceptual model of the EIS software. The software, the rules and the database are automatically constructed from this conceptual model.

Due to space limitations, we will not discuss the details of the user Interface component or the Service component. We focus on Metadata, Persistence and Business Rules components to explain our approach for code generation and database schema evolution based on EIS metadata.

To introduce the ideas of this approach and the framework which implements these ideas, this paper is organized as follows. Section 2 provides an overview of the framework that generates the EIS code from the system conceptual model. Section 3 discusses the conceptual modeling of databases and business rules for EIS. Section 4 details the code generation for business rules and for the EIS database. Section 5 discusses aspects of EIS model evolution that are critical for the maintenance of EIS. Finally, Section 6 presents concluding remarks, comparing our approach to related works and pointing directions for future work.

2. Framework Overview

Our framework was inspired on the ideas of model-driven engineering [Schmidt 2006]. The framework was built to support automatic generation and maintenance of EIS software components using the EIS conceptual model as its input.

The macro-architecture of the framework, shown in Figure 1, has five main components. The Interface, Business Rules and Persistence components contains transformation procedures and tools that map each aspect of the EIS conceptual model (application functions, business rules, and database schema, respectively) to software implementation models, generating the corresponding code.

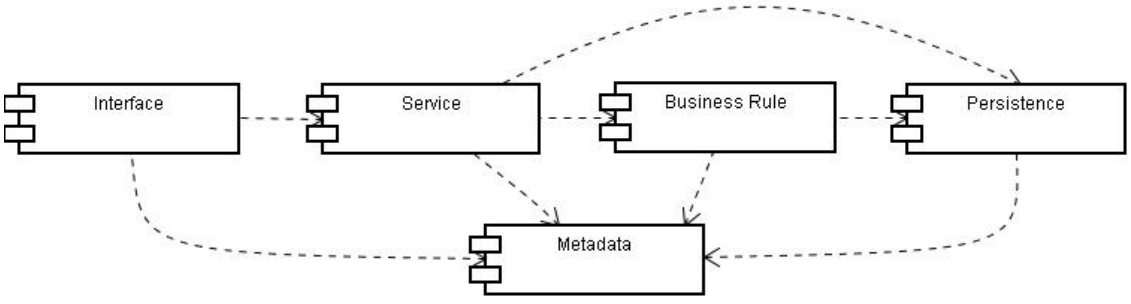

Figure 1. Framework components architecture.

The **Metadata** component is at the core of the framework. It supports all other components and is responsible for managing the EIS conceptual model. This conceptual model defines metadata that are used by the other framework components to build and manage the execution of the EIS software code (application functions, business rules, and database schema). This component doesn't need any service from other components but it provides some services in its interface that are used in Object-Relational Mapping, like obtaining logical key attributes from an entity.

The **Interface** component of the architecture uses the EIS conceptual model (or simply the EIS metadata) to automatically generate the user interface widgets for the business applications. The Metadata component maintains, for each EIS concept, a User-Interface mapping describing how to present, in a user interface widget, each business concept.

The **Service** component provides tools and services to register and convert information from the Interface to other components of the framework. Thus, it acts like a facade, isolating the user interface aspects of the EIS. For instance, the Interface component depends on the Service component to map user interface data to and from persistent data, as well as to evaluate business rules. The separation of the Interface component makes it easier to change the user interface without affecting the core of the EIS. This component need services provided from Business Rule Component to validate User information and services provided from Metadata Component used in ORM. The Service component provides services to manipulate the entity instance. These services are creating, read, update and delete.

The **Business Rule** component is responsible for managing a centralized business rules repository, which is stored in a database. To access the database, the Business Rule uses the Persistence component services provided in its interface. The main responsibilities of the Business Rule component is to translate the OCL rules definitions to platform specific code, and to provide runtime facilities for evaluating and enforcing these rules.

The **Persistence** component maps the EIS conceptual model to the operational data model of the underlying DBMS and manages the evolution of the database schema according to the EIS conceptual metadata. This component also manages all the access to persistent data, isolating the other framework components from changes in the database technology. This component requires services provided in Metadata component that are used to set up and adjust stored procedures parameters. This component also provides services to create and evolve database schema. This component requires the same functionalities used in ORM provided in the Service Component.

2.1. Using the Framework to Build an EIS

The first step to build an EIS using our framework is registering Business Entities Metadata. Entity Metadata is divided in three types: representation, presentation and rules. After register these information, tables and stored procedures (used by Persistence component PC) are generate automatically to store, manipulate and validate the Entity data (used by Rules component RC). After that, the software to manipulate Entity information is ready to use.

When a program is executed, the Interface component obtains, from Metadata component (MC) the relative Meta information from the Business Entity. This Meta in-

formation is used to assembly the User Interface (UI) that manipulates Entity instances. Create, read, update and delete are functions provided by UI. The Entity instance obtained by UI is passed to Service Component (SC). For example, if the user chooses the Create operation in UI the Service component pass this instance to Rule component for validation. If ok, the SC obtains the metadata from MC and does the Object Relational Mapping (ORM); the SC pass the mapped data to PC that call the specifics stored procedures from the manipulated Entity.

3. Conceptual Modeling of EIS Database and Business Rules

The Business Rules (BR) of an EIS can be considered as statements that define or constraint any business aspect [Group 2000]. These rules formalize the business concepts, the relationships among these concepts, and the constraints that must be enforced to guarantee the integrity and consistency of business data and processes.

Since BR constraint business operations, they are also known as application domain rules [Morgan 2001]. The implementation of application programs allows EIS users to perform business processes and to manipulate business information according to the BR. Therefore, one can think of BR as abstract expressions that define and constraints the EIS, directing it to fulfill the underlying business domain information needs. From this perspective, business rules can be classified in four categories [Gottesdiener 1997]:

1. definitions (of business terms);
2. facts (that connect terms);
3. constraints (on terms and/or facts); and
4. derivations (which infer new terms and/or facts from those already known).

The first two categories contain structural rules, which are well supported by current modeling tools (UML, or the relational data model, for instance). Thus, our work focuses on the other two rules categories: constraints and derivations (which we call, respectively, validation and derivation rules). We refer to the set of rules in these two categories as action rules.

Traditionally, BR are represented and implemented as code embedded into the application programs or into the database schema. Thus, business rules are second class citizens, subordinated to databases or application programs. Rules are analyzed, designed and implemented as a dependent concept, considering an implementation perspective of the system.

This approach has several drawbacks, mainly on the portability and maintainability of the EIS, due to the tight coupling between the definitions of what the system must do (specified by the BR) and how the system works (coded in the application programs and database constraints) [Date 2000].

To minimize these difficulties, EIS BR should be abstractly and independently represented, and should contain no implementation detail (platform or technological definitions, for instance). Defining rules in such a way is possible, but it demands an adequate infrastructure to manage the independent rules.

In our approach all BR properties are stored in a single Enterprise Information System Business Rule (EIS-BR) repository, implemented as a rules database. Thus, a

database management system (DBMS) enforces security and provides accessibility for authorized users to access the BR repository. Only authorized staff has access to the centralized repository of rules definitions.

To conceptually represent the structural aspects of the EIS domain, such as business concepts, instances, relations, and static constraints, we implemented an object-oriented variation of the classic Entity-Relationship (ER) Conceptual Data Model. Using this model we can automatically generate the EIS database schema, using well-known ER to SQL mapping algorithms. However, the ER model provides appropriate support only to structural constraints; action rules, assertions and derivation rules are not directly supported by this model.

To define and evaluate these types of rules, which are not naturally represented with ER modeling primitives, we chose a language specifically designed to express rules within an object-oriented model. The OCL (Object Constraint Language) expressions allow the designer to define:

1. Invariant conditions (which must be satisfied in all system states);
2. Query expressions (covering all the model elements); and
3. Constraints on operations that change system´s state (e.g., pre-conditions).

Combining the expressive power of the ER conceptual data model with OCL dynamic constraints expressions allows our framework to specify both structural and behavioral constraints in a high level, abstract model of the EIS. In the example illustrated in Figure 2, an ER model represents a simplified enterprise domain. This simple conceptual model contains structural constraints, such as the following cardinality (or multiplicity) constraints: Rule 1) ”An Employee must work in a single Department”; and Rule 2) ”An Employee can manage at most one Department”.

However, the constraint expressivity of this simple model is limited, since only structural constraints can be represented. For example, suppose that the following business rule most be enforced: Rule 3) ”An Employee can manage only the Department in which he works”. This business rule is not represented in Figure 2, and due to ER expressive power limitations, it can not be stated without modifying the EIS model structure.

Our solution to this problem is to use OCL for representing rules that cannot be expressed using ER modeling primitives, such as validation and derivation rules. The business rule mentioned above could be formally represented in OCL as the query expression shown in Code 1.

In this approach, rules are specified as an independent aspect, i.e., rules are first class citizen in the EIS conceptual model. The separation of rules, data and functions is not complete, since rules have influence on data and functions; however, there is no subordination of rules specification with regard to data or functions specifications. Moreover, rules are formally expressed, but without dependency of implementation technologies or specific platforms.

4. Code Generation Mechanisms

The model-driven engineering approach on which our framework is based demands automatic code generation facilities from the abstract conceptual EIS model. In this work

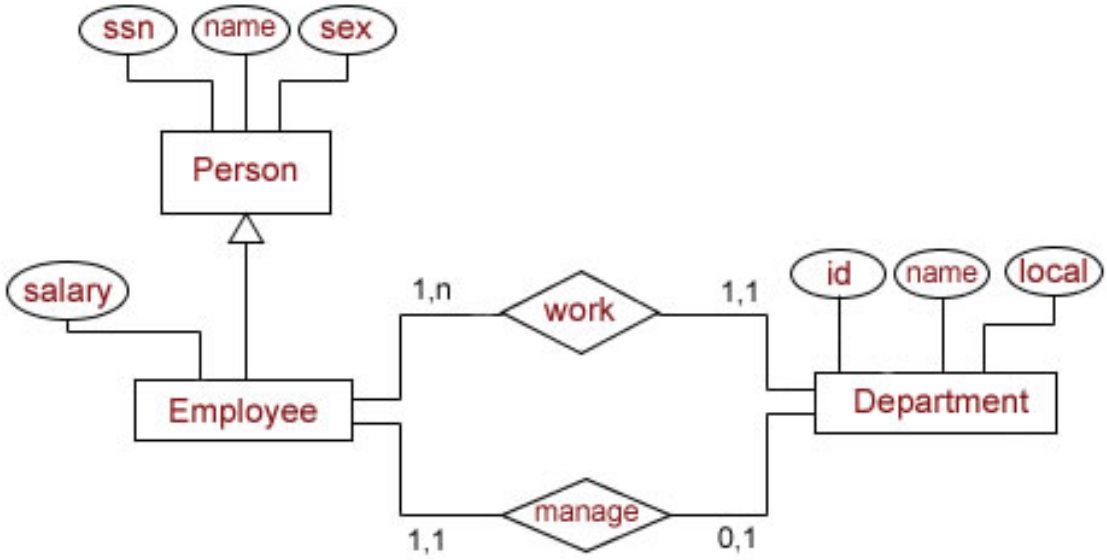

Figure 2. A Simplified Conceptual Model for an EIS.

we focus on two main aspects concerning this code generation: EIS database definition and manipulation (Section 4.1) and business rules implementation (Section 4.2). User interface and application programs rely on the database and rules code, but due to space limitations are not discussed in this paper.

4.1. Schema generation and manipulation

A database schema transformation tool in the Persistence component generates a SQL schema for an EIS database through automatic translation of the EIS conceptual metadata. The specific operational data model of the DBMS defines the details of the mapping process, since there are variations of SQL features supported on different database management systems.

As we have said, the conceptual model used in our framework is based on the Entity-Relationship model. Our transformation tool is capable of generating SQL structures corresponding to the conceptual primitives of the source model, such as strong and weak entities, specialization hierarchies, relationship aggregation, and composite attributes.

There are several tools that perform the same kind of transformation, generating a SQL database schema from an ER conceptual schema. The new idea in our framework is to use behavior information that is also captured in our conceptual model (via action rules) to automatically generate stored procedures that provide basic CRUD (Create, Read, Update, Delete) operations on entity instances. Every conceptual entity type has a set of built-in procedures to perform these CRUD operations on its instances. The application functions benefit from this feature.

To illustrate the database schema transformation tool operation, consider the schema shown in Figure 2. The purpose of this example is to show the result of database schema transformations to generate an operational data definition for the conceptual schema presented in Figure 2.

```
1  context Employee::validate_Employee(
2    employee_id : Integer ) : Boolean
3  body:
4   let
5     idDeptEmpWork : Integer = Employee.allInstances()->
6          select (id = employee_id).departament_work.id,
7
8     idDeptEmpManage: Integer = Employee.allInstances()->
9          select (id = employee_id).departament_manage.id,
10
11    rv_Employee: Boolean =
12    (
13      if(idDeptEmpManage <> null and
14         idDeptEmpWork <> idDeptEmpManage)
15        then true
16      else false
17      endif
18    )
19
20   in
21     result = rv_Employee
```

Code 1: *A Business Rule expressed in OCL.*

In a typical database environment, the application developer (or the database administrator) has to translate the conceptual schema into the operational schema. This translation is supported by database schema transformation tools in the main DBMS. Our tool also makes this translation automatically, generating the SQL schema shown in Figure 3.

```
CREATE TABLE Person (
  ssn VARCHAR NOT NULL,
  name VARCHAR NOT NULL,
  sex VARCHAR NOT NULL,
  PRIMARY KEY (ssn),
  CHECK (sexo = 'Masculine' OR sexo = 'Feminine')
)

CREATE TABLE work (
  idDepartment VARCHAR NOT NULL,
  ssnEmployee VARCHAR NOT NULL,
  FOREIGN KEY (idDepartment)
          REFERENCES Department (id),
  FOREIGN KEY (ssnEmployee)
          REFERENCES Employee (ssn)
)

CREATE TABLE Department (
  id VARCHAR NOT NULL,
  name VARCHAR NOT NULL,
  local VARCHAR NOT NULL,
  PRIMARY KEY (id)
)

CREATE TABLE Employee (
  ssn VARCHAR NOT NULL,
  salary DOUBLE PRECISION NOT NULL,
  FOREIGN KEY (ssn)
      REFERENCES Person (ssn)
)

CREATE TABLE manage (
  idDerpartment VARCHAR NOT NULL,
  ssnEmployee VARCHAR NOT NULL,
  FOREIGN KEY (idDepartment)
      REFERENCES Department (id),
  FOREIGN KEY (ssnEmployee)
      REFERENCES Employee (ssn),
  UNIQUE (idDepartment)
)
```

Figure 3. SQL Schema generated from the Conceptual Schema in Figure 2.

The main advantages of our approach when comparing to conventional DBMS tools are: a) the generated schema is integrated with other generated aspects of the EIS software (rules and application functions); and b) data manipulation stored procedures are automatically incorporated to the database schema, providing an important facility to upper level application and user interface code.

The transformation tool generates the SQL-DDL schema and also the stored procedures (SQL-DML) to manipulate instances of all entities and relationships of the conceptual schema (which are all converted to SQL tables). Figure 4 shows the stored procedures created to manipulate the database schema in Figure 3. Only the signature of each stored procedures is shown due to space limitations, but our tool automatically generates the full procedure code for each basic data manipulation operation: select, insert, update

and delete.

Person

```
CREATE OR REPLACE FUNCTION updatePerson (ssnOrig VARCHAR, ssnMod VARCHAR,  nameMod VARCAHR, sexMod VARCHAR)
CREATE OR REPLACE FUNCTION insertPerson (ssn VARCHAR,  name VARCAHR, sex VARCHAR)
CREATE OR REPLACE FUNCTION deletePerson (ssn VARCHAR)
CREATE OR REPLACE FUNCTION selectPerson (ssn VARCHAR)
CREATE OR REPLACE FUNCTION selectPerson ()
```

Employee

```
CREATE OR REPLACE FUNCTION updateEmployee (ssnOrig VARCHAR, ssnMod VARCHAR,  nameMod VARCAHR, sexMod VARCHAR,
                                           , salaryMod DOUBLE PRECISION)
CREATE OR REPLACE FUNCTION insertEmployee  (ssn VARCHAR,  name VARCAHR, sex VARCHAR, salary DOUBLE PRECISION)
CREATE OR REPLACE FUNCTION deleteEmployee  (ssn VARCHAR)
CREATE OR REPLACE FUNCTION selectEmployee  (ssn VARCHAR)
CREATE OR REPLACE FUNCTION selectEmployee  ()
```

Department

```
CREATE OR REPLACE FUNCTION updateDepartment (idOrig VARCHAR, idMod VARCHAR,  nameMod VARCAHR, localMod VARCHAR)
CREATE OR REPLACE FUNCTION insertDepartment (id VARCHAR,  name VARCAHR, local VARCHAR)
CREATE OR REPLACE FUNCTION deleteDepartment (id VARCHAR)
CREATE OR REPLACE FUNCTION selectDepartment (id VARCHAR)
CREATE OR REPLACE FUNCTION selectDepartment ()
```

Manage

```
CREATE OR REPLACE FUNCTION insertManage (idDepartment VARCHAR,  ssnEmployee VARCHAR)
CREATE OR REPLACE FUNCTION deleteManage(idDepartment VARCHAR,  ssnEmployee VARCHAR)
CREATE OR REPLACE FUNCTION selectManage (idDepartment VARCHAR)
```

Work

```
CREATE OR REPLACE FUNCTION insertWork (idDepartment VARCHAR,  ssnEmployee VARCHAR)
CREATE OR REPLACE FUNCTION deleteWork(idDepartment VARCHAR,  ssnEmployee VARCHAR)
CREATE OR REPLACE FUNCTION selectWork (idDepartment VARCHAR)
```

Figure 4. Stored Procedures automatically generated from the schema in Figure 2.

These automatically available stored procedures are used as basic building blocks for the construction of the EIS specific applications. All the persistence aspects of the application programs are encapsulated into these procedures. Thus, our framework enforces the data independence principle.

Figure 5 shows the ER Transformation package, which contains the main functions of the Persistence component. In this figure, the Metadata component is represented as a class of the package, but as we have already discussed, it is an independent component of the framework.

The DBMappingLibrary class obtains mapping information necessary to transformation process from ER to target DBMS. The Generation Processor is a class that contains all essential methods for generating SQL-DDL schema and stored procedures. The Evolution Processor is a class that treats the database evolution manipulation process.

4.2. Code generation for business rules

In order to transform OCL business rules into SQL, the mappings between elements of these languages must be described. In [Demuth and Hussmann 1999] some patterns for invariants transformations are defined, specifying their general structure, attribute access and navigation across associations. These ideas were used in our work to develop transformation patterns according to our EIS conceptual metadata.

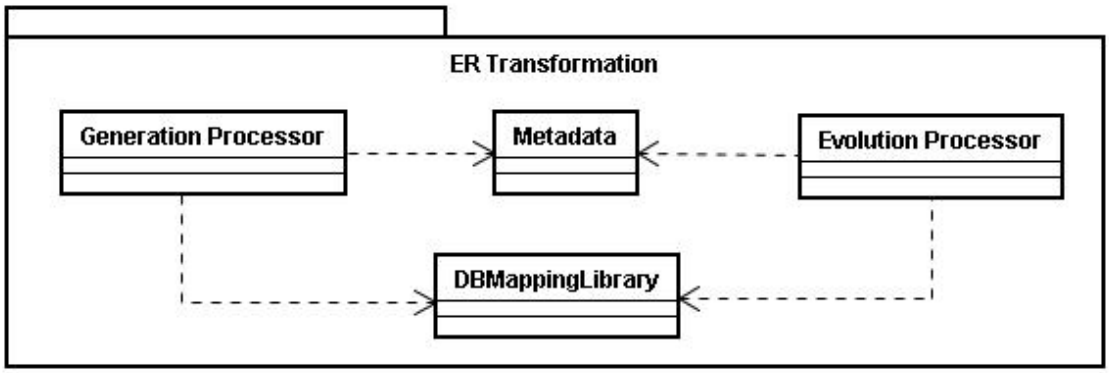

Figure 5. The Persistence component of the framework.

In our approach, all mappings are defined in a XML (Extensible Markup Language) [(W3C) 2009] file, based on the Dresden OCL Toolkit [Group 2008]. The XML file contains information to guide the rule transformation process. After defining the mappings, the next step is to parse the OCL expressions in order to validate them according to the OCL Grammar and to build an Abstract Syntax Tree (AST) with its elements.

The rule transformation process requires that all previous steps have been performed successfully. To transform the OCL expression into SQL code the following components are mandatory:

- A XML file containing mappings from OCL to the specific SQL dialect;
- AST and metadata structure, containing information about model elements.

Figure 6 shows the OCL Transformation package, which contains the main functions of the Business Rules component. In this figure, the Metadata component is the same used in ER transformations. The MappingLibrary class uses the external xerces.jar [Group 2008] component, which is responsible for reading a XML file containing mappings from OCL to the target platform.

The Translator is an interface that contains all the essential methods for executing transformations between models. It uses the OCL Parser external component, which is responsible for parsing OCL expressions.

The following sequence of activities must be performed to transform an OCL expression into SQL code:

1. Get mappings between OCL and SQL: it is necessary to retrieve all mappings that are defined in the XML file. Each mapping indicates how to generate code in a specific target language (in our case, the PostgreSQL procedural language - plPgSQL was adopted).
2. Get rule elements: the AST previously built contains all tokens in the OCL rule. Thus, we can use this activity to visit any node in the AST and to make it available for further activities. This step makes it easier to access important parts of the rule, such as its parameters.
3. Get business entities metadata: as we have explained, the metadata structure represents model elements (like entities, their attributes and relationships). Each business rule must have a rule context, which can be an entity or an attribute. It is

necessary to get the rule context related metadata to understand the entity structure (attribute composition and relationships). This knowledge is necessary to navigate through the specific rule context within the model.

4. Translate rule parameters: when an OCL rule have entry parameters, it is necessary to translate their types to corresponding SQL data types, according to the mappings defined in the XML file. All parameters have the same name as the respective attributes in the metadata model, but they can be changed in this step according to the needs of the transformation process.
5. Translate rule expressions: this is the most critical activity for the rule transformation process. In this step we walk through the AST visiting each node and, according to its type, it is possible to know which mapping pattern, defined in the XML file, should be applied to generate the SQL code.

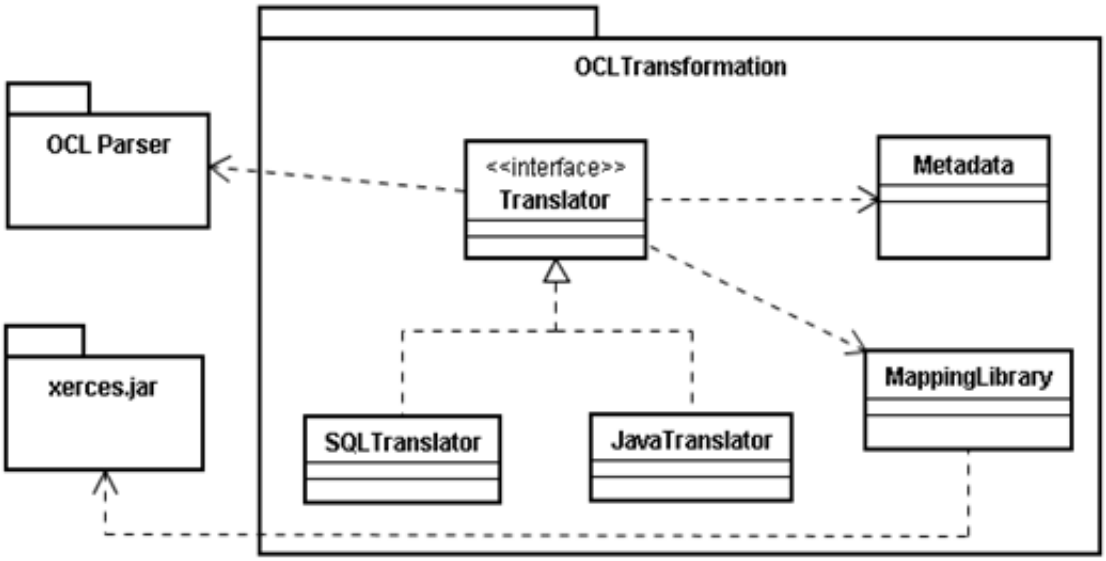

Figure 6. The Business Rule component of the framework.

The generated rule should be executed when predefined operations are done in application programs. Following the active database principle, our mechanism is sensitive to operations that modify the state of the EIS database: insert, update and delete. When these operations occur within an application program, the rule evaluation process is triggered.

As an example of rule code generation for stored procedures, consider the business rule modeled in Section 3: ”An Employee can manage only the Department in which he works”. This business rule was modeled in OCL, as shown in Code 1. After being processed by the transformation tool, the code in Code 2 was generated.

Our framework implementation maps the OCL business rules to stored procedures in a specific SQL dialect. Nevertheless, this is only a mapping decision. It is possible to adjust the mapping procedure to translate from OCL to a programming language, for instance.

```
1  CREATE OR REPLACE FUNCTION validate_Employee(
2    employee_ssn INTEGER
3  )RETURNS BOOL AS $$
4  DECLARE
5   idDeptEmpWork INTEGER;
6   idDeptEmpManage INTEGER;
7   rv_Employee BOOL;
8
9  BEGIN
10   SELECT INTO idDeptEmpWork Department.id
11   FROM Employee JOIN department_work ON Employee.pk =
12        department_work.pkEmployee JOIN Department ON
13        department_work.pkDepartment = Department.pk
14   WHERE Employee.ssn = employee_ssn;
15
16   SELECT INTO idDeptEmpManage Department.id
17   FROM Employee JOIN department_manage ON Employee.pk =
18        department_manage.pkEmployee JOIN Department ON
19        department_manage.pkDepartment = Department.pk
20   WHERE Employee.ssn = employee_ssn;
21
22   IF (idDeptEmpManage IS NOT NULL AND
23       idDeptEmpWork <> idDeptEmpManage)
24         THEN RETURN FALSE;
25   ELSE
26         RETURN TRUE;
27   END IF;
28 END;
29 $$ LANGUAGE plpgsql;
```

Code 2: *Transformation of the OCL Business Rule of Figure 3 into SQL*

5. Evolution of the EIS Conceptual Model

When the business context changes, it is necessary to evolve the EIS conceptual model to incorporate the new business concepts. In our approach for database schema evolution, the Persistence component database transformation tool receives a modified version of the EIS metadata. For each modified entity type, the component analysis all modifications, comparing the given entity with the correspondent entity stored in the current metadata database.

Modifications on the conceptual model lead to a set of schema evolution operations that must be propagated to the operational model (including both SQL-DDL and stored procedures code).

After verifying the consistency of the new version of the entity type, the database evolution tool automatically modifies the SQL-DDL, the stored procedures, and all the entity type instances stored in the database. Therefore, after the modification of a conceptual entity type, the whole schema is ready to be used to manipulate the entity type instances.

The design of the schema evolution mechanism defines all allowed modifications on a conceptual schema. Some modification operations can be executed directly on most Database Management Systems (DBMS); other operations are specific to our schema evolution mechanism.

For example, changing the domain of an integer attribute to alphanumeric is allowed by most DBMS, but not the contrary, because there may exist alphanumeric attribute values in the database instances that cannot be converted to integer. However, if

all values in the current database state could be converted to the integer domain, the conversion operation could be allowed. This can only be decided at runtime, since it depends on the current state of the database instances.

Changing an attribute domain is relatively simple, but our component supports complex schema evolution operations, such as changing the state of entity type from strong to weak (i.e., creating an identification dependency with another entity type). The component validates the modifications in the conceptual model and propagates the permitted operations to the operational schema, assuring the EIS database consistency. Therefore, the component avoids directly modification of the operational database schema by users or developers. Modifications are performed in the conceptual level and automatically propagated to the operational database schema.

The examples below use the EIS conceptual model shown in Figure 2. First, we will remove the manage relationship between entities Employee and Department. In order to remove this relationship, the following schema evolution operations are necessary:

1. Drop all stored procedures that manipulate the manage relationship instances.
2. Drop table manage.
3. Remove relationship manage from the conceptual model.

The second example shows how the mechanism works when an attribute is added to an entity. We will add the type attribute into the entity Employee. This attribute has two possible values: manager and employee. Mandatory operations to include this attribute are:

1. Create attribute type on the Employee entity in the conceptual model.
2. Drop stored procedures that manipulate the entity Employee.
3. Create type attribute on the table Employee.
4. Create a constraint that checks if an Employee is manager or employee.
5. Assign the value 'employee' to attribute type in all instances of Employee.
6. Create all stored procedures to manipulate the entity Employee.

Figure 7 shows the conceptual schema after all these modifications. Operation 1 makes the conceptual model update. Operations 2, 3, 4 and 6 update the physical schema and operation 5 propagates modifications to instances in the database.

Figure 8 shows the corresponding operational (SQL) schema re-generated from the updated conceptual schema.

In this example, after modifying the conceptual schema, we can note that a business rule specified in the schema before the modification ("An Employee can manage only the Department in which he works") is no longer necessary. The modified schema now is capable of structurally enforcing this business rule. Therefore, database schema evolution and business rules evolution have to be mutually consistent.

The obsolete rule must be eliminated, and it is now necessary to create other business rules to assure database consistency. For example: "A department cannot have more than one employee whose type is manager". This new business rule can be easily modeled in OCL and implemented as stored procedures using our approach.

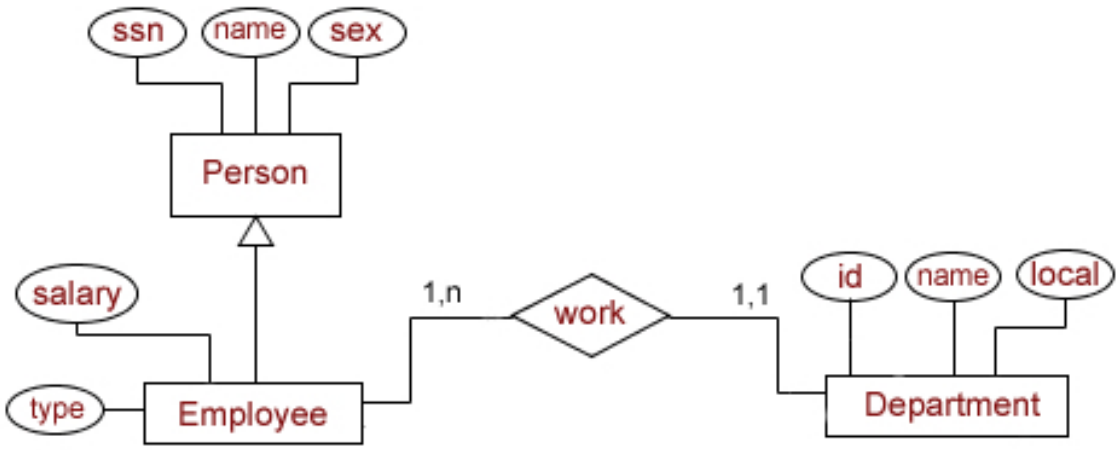

Figure 7. Conceptual schema after schema evolution.

```
CREATE TABLE Person (
  ssn VARCHAR NOT NULL,
  name VARCHAR NOT NULL,
  sex VARCHAR NOT NULL,
  PRIMARY KEY (ssn),
  CHECK (sexo = 'Masculine' OR sexo = 'Feminine')
)

CREATE TABLE work (
  idDepartment VARCHAR NOT NULL,
  ssnEmployee VARCHAR NOT NULL,
  FOREIGN KEY (idDepartment)
          REFERENCES Department (id),
  FOREIGN KEY (ssnEmployee)
          REFERENCES Employee (ssn)
)

CREATE TABLE Department (
  id VARCHAR NOT NULL,
  name VARCHAR NOT NULL,
  local VARCHAR NOT NULL,
  PRIMARY KEY (id)
)

CREATE TABLE Employee (
  ssn VARCHAR NOT NULL,
  salary DOUBLE PRECISION NOT NULL,
  type VARCHAR NOT NULL,
  FOREIGN KEY (ssn)
        REFERENCES Person (ssn)
  CHECK (type = 'manager' OR type = 'employee')
)
```

Figure 8. Operational (SQL) schema after schema evolution.

6. Conclusions and Related Work

The framework presented in this paper was implemented on a research project developed from 2005 to 2008 with financial support from the Brazilian National Council for Scientific and Technological Development (CNPq). The final objective of this project is to build a complete framework to create and evolve Enterprise Information Systems (EIS) for agricultural business domains.

In our framework, application programs, database schemata and business rules are conceptually described in a single conceptual (metadata) model and automatically implemented as separated but interdependent aspects.

The software mechanism of the framework is implemented in Java and has approximately 67 thousand lines of code. The conceptual schema of the EIS developed as a proof of concept contains over 200 business entities from the agricultural business domain.

The EIS rules repository contains almost 150 business rules specified and implemented in the EIS software. Among these rules there are about 85 structural (validation) rules and 55 action and derivation rules.

The generated operational database schema contains over 560 tables and three thousand stored procedures, including those used for data manipulation and business

rules.

Several tools use similar approaches to generate parts of the EIS software from model transformations [Objects 2008, Andromda 2008, Eclipse 2008, OpenArchitectureWare 2008, Borland 2008]. Many of them generate the database schema from the conceptual model, but they do not generate stored procedures or other manipulation facilities for the conceptual entities, neither they provide an integrated framework for EIS software development. Other approaches, like JPA [Biswas and Ort 2006], provide database mechanisms to generate automatically tables and manipulation facilities using annotations and EJB-QL (Enterprise Java Bean Query Language), but this approach provides no facilities to make changes in a conceptual model, for example, support to complex schema evolution operations, such as changing the type of an entity from strong to weak.

In AndroMDA, for instance, it is possible to transform business rules expressed in OCL into other languages, such as HQL (Hibernate Query Language) [Team 2009] and EJB-QL. Our translation mechanism differs from AndroMDA in the choice of the transformation paradigm: while our transformation is based on a high level platform independent language, AndroMDA transformations are based on strings and regular expressions.

In the database evolution context, works like [Comyn-Wattiau et al. 2003] suggest the support to bidirectional changes on system models. Our architecture proposes that changes should be made only on the conceptual model, with automatic propagation of changes to the operational model. The idea is that changes to the operational schemata should be prohibited, for the same reason that changes in the source code are allowed, but changing the machine code is not recommended.

Some works, like [Franconi et al. 2000], allow having several complete versions of the logical schema in the system, but our approach keeps only the last version of the schema. For large EIS, keeping several versions of the database is impracticable due the large amount of storage capacity needed and the low benefits associated with this practice. In other contexts, such as CAD software, the need for full versioning may be compensatory.

In the business rules context, several works investigated the automatic conversion of rules expressed in high level languages to software source code. The implementation in [Milanovic et al. 2008], for instance, shares rules between two rule languages from different domains: OCL together with UML and SWRL (Semantic Web Rule Language). In [Brambilla and Cabot 2006] there is a description of an OCL to SQL transformation and its tuning for web applications. [Cabot and Teniente 2006] proposes a method for changing integrity constraint representations by changing its context, but without changing the constraint meaning.

Our solution is based on the Dresden OCL Toolkit, a modular software platform for OCL, providing facilities for the specification and evaluation of OCL constraints. The toolkit performs parsing and type-checking of OCL constraints and generates Java and SQL code [Group 2008].

We have reused many ideas from this toolkit, but we had to modify and adapt several features to fulfill the requirements of our mechanisms. One important modification is related to the target language. Our mechanism generates stored procedures to convert

target business rules from OCL to SQL code while the original toolkit generates SQL code in form of database views [Heidenreich et al. 2008].

The main advantages of our approach are the portability and the maintainability of the EIS, besides the automatic code generation, which reduces the programming efforts for building the EIS software.

The portability is improved because the business rules are represented with an abstract declarative language (OCL), which is an OMG standard, just like UML. The rules are defined in OCL (a platform independent model) and automatically converted to a specific platform using a software translator.

Our approach improves the availability of the business rules, since there is a single repository where all the EIS rules are stored. This repository is managed by a DBMS that offers browsing and querying facilities, besides security and access control capabilities. In traditional EIS, the rules are hard-wired in either the application program code or in the database schema as integrity constraints.

By applying the classic separation of concerns principle, the separation of business rules, application code, and database schemata improves the software maintainability. Rules are documented in a single model, and are not mixed with the application code. This centralized organization improves the code organization and makes it easier to evaluate the impact of changes on business rules.

As future works, it would be interesting to use metamodeling concepts of model-driven development to adapt the framework presented in this paper, allowing to use different conceptual models, like UML or ORM, for example. In business rule component, code generation performance can be improved. Besides, a workflow engine can be developed to control access and business rules changes, and also manage their evaluation during IS execution.

References

Andromda (2008). Andromda Framework. `http://www.andromda.org/`, Acessed July 2009.

Biswas, R. and Ort, E. (2006). The Java Persistence API - A Simpler Programming Model for Entity Persistence. `http://java.sun.com/developer/technicalArticles/J2EE/jpa/`, Acessed July 2009.

Borland (2008). Together. `http://www.borland.com/br/products/together`, Acessed July 2009.

Brambilla, M. and Cabot, J. (2006). Constraint tuning and management for web applications. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 345–352, New York, NY, USA. ACM.

Cabot, J. and Teniente, E. (2006). Transforming ocl constraints: a context change approach. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1196–1201, New York, NY, USA. ACM.

Comyn-Wattiau, I., Akoka, J., and Lammari, N. (2003). A framework for database evolution management. In *ETAPS'03: Proceedings of the European Joint Conferences on Theory and Practice of Software*, pages 105 – 113. ETAPS'03.

Date, C. J. (2000). *What, not How - The Business Rules Approach to Application Development*. Addison-Wesley.

Debevoise, T. (2007). *Business Process Management with a Business Rules Approach: Implementing the Service Oriented Architecture*. BookSurge Publishing.

Demuth, B. and Hussmann, H. (1999). Using uml/ocl constraints for relational database design. In *UML*, pages 598–613.

Eclipse (2008). M2M. `http://www.eclipse.org/m2m`, Acessed July 2009.

Franconi, E., Gr, F., and M, F. (2000). A semantic approach for schema evolution and versioning in object-oriented databases. In *In Proc. of the 1st International Conf. on Computational Logic (CL 2000), DOOD stream*, pages 1048–1062. Springer-Verlag.

Gottesdiener, E. (1997). Business rules show power and promise. In *Application Programming Trends, vol. 4, n. 3*.

Group, S. T. (2008). Dresden ocl toolkit. Technische Universitat Dresden. Available at http://dresden-ocl.sf.net.

Group, T. B. R. (2000). Defining business rules - what are they really? Avaiable at `http://www.businessrulesgroup.org/first_paper/br01c1.htm`.

Heidenreich, F., Wende, C., and Demuth, B. (2008). A framework for generating query language code from ocl invariants. *ECEASST*, Vol. 9.

Milanovic, M., Gasevic, D., Giurca, A., Wagner, G., and Devedzic, V. (2008). Sharing ocl constraints by using web rules. *ECEASST*, 9.

Morgan, T. (2001). *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison-Wesley Longman Publishing Co., Inc.

Objects, I. (2008). ArcStyler Interactive Objects. `http://www.arcstyler.com`, Acessed July 2009.

OpenArchitectureWare (2008). The leading platform for professional model-driven software development. `http://www.openarchitectureware.org`, Acessed July 2009.

Schmidt, D. C. (2006). Model-driven engineering. In *Computer, vol. 39, no. 2*, pages 25–31. doi:10.1109/MC.2006.58.

Team, H. P. (2009). Hibernate query language. `http://www.hibernate.org/hibdocs/reference/en/html/queryhql.html`, Acessed July 2009.

(W3C), T. W. W. W. C. (2009). Extensible markup language (xml). Avaiable at http://www.w3.org/XML/. Accessed May 2009.