

Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications

Cristina Marinescu
LOOSE Research Group
“Politehnica” University of Timișoara, Romania
cristina.marinescu@cs.upt.ro

Abstract

As enterprise applications become more and more complex, the understanding and quality assurance of these systems become an increasingly important issue. One specific concern of data reverse engineering, a necessary process for this type of applications which tackles the mentioned aspects, is to retrieve constraints which are not explicitly declared in the database schema but verified in the code. In this paper we propose a novel approach for detecting the relational discrepancies between database schemas and source-code in enterprise applications, as part of the data reverse engineering process. Detecting and removing these discrepancies allows us to ensure the accuracy of the stored data as well as to increase the level of understanding of the data involved in an enterprise application.

Keywords: quality methods, data reverse engineering, enterprise applications, meta-models

1. Introduction

As object-oriented systems become more and more complex, the understanding and quality assurance of these systems are increasingly important issues. Consequently, the need of using proper reverse engineering techniques to analyze these systems became a must. Moreover, understanding and maintenance activities become even more difficult, when considering a recently emerged category of software systems, namely *enterprise applications*. Almost all enterprise applications involve two programming paradigms: the object-oriented one, for implementing the entire business logic, and the relational paradigm, for ensuring the persistency of the involved data.

It has been repeatedly emphasized [9, 16] that the development of software systems that involve both paradigms raises significant understanding and maintenance problems, especially when it comes to the relations among the two “worlds”. Therefore, the need of defining advanced reverse

engineering techniques and qualitative methods to support the development and maintenance of these systems has substantially increased.

A substantial number of reverse engineering methods and quality assessment techniques have been developed in the past for object-oriented systems [2, 12, 19]. But, when it comes to understand and evaluate the quality of the relationships existing within the relational part of the system (*i.e.*, the persistency layer) and of the interrelationships between the relational and the object-oriented parts, we are still in need for more advanced techniques and tools.

These concerns emphasize the increasing role of *Data Reverse Engineering* (DRE), *i.e.*, “a collection of methods and tools to help an organization determine the structure, function, and meaning of its data.”[4]. One specific concern which needs to be addressed in the context of DRE is to retrieve constraints which are not explicitly declared in the database schema but are verified in the code [10]. This is a complex and expensive task and needs to be supported by program understanding and tools [5].

In this paper we propose a novel approach for detecting *relational discrepancies* between database schemas and source-code in enterprise applications. The approach defines and uses specific structural analyses based on a representation (model) of a software system that contains and correlates relevant entities and relationships from both the object-oriented (*e.g.*, classes, method calls) and the relational parts (*e.g.*, tables, columns, accessed tables) of an enterprise application.

The paper is structured as follows: in Section 2 we present a brief overview of enterprise applications. Next (Section 3) we state exactly the addressed problem regarding relational discrepancies by means of an example. In Section 4 we describe the approach proposed for identifying such discrepancies. In Section 5 we give a description of the tool support that ensures the automation of the entire approach. In Section 6 we describe and discuss the validation of our approach based on the results obtained from two case-studies. The experiment is aimed to reveal both

the applicability and the accuracy of the approach. The paper concludes with a discussion on related work (Section 7) and some final remarks concerning the future work (Section 8).

2. Enterprise Applications. Assumptions

Enterprise applications are about the display, manipulation, and storage of large amounts of complex data and the support or automation of business processes with that data [9]. Within such applications a key design practice is to decouple the commands that ensure the persistency (usually the persistency is supplied by a database, mostly relational database) and application logic because mixing them hampers understanding and testing the application. This constraint leads to a multi-layered architecture, consisting of three primary layers, namely the *data source*, the *domain* and the *presentation* layer [8].

Enterprise applications can follow a large variety of implementation patterns. Therefore, we must state that in the rest of the paper we focus on enterprise applications implemented in an object-oriented language (*e.g.*, Java, C#). Our second assumption is that persistency is provided by a relational database (*e.g.*, MySQL) and that the communication between the entities belonging to the data source layer and the relational database is performed by executing SQL commands as embedded strings from well-known methods as *executeQuery(String sql)*, *executeUpdate(String sql)*.

3. Relational Discrepancies

A database schema models mainly tables, columns and the various relations and constraints that exist between the data stored in the database's tables. Most, if not all, of the relations are the reflection of the business logic (behavior) which is modeled in the source-code. Therefore, in a normal case there should be a correlation between the constraints defined on tables in the database schema and the co-usage of those tables in the source-code. But there is nothing to guarantee that the source-code and the database schema stay synchronized. When this is not the case we say that a *relational discrepancy* appears between the source-code and the database schema. We identified two possible cases of such relational discrepancies:

Missed Constraint. Two or more tables are constantly accessed together (jointly) in code but there is no constraint defined in the database schema to suggest that. For example, in Figure 1 we see that both classes B and C access always together tables T1 and T2, like these would be related by a foreign key. But looking in the database schema we don't find such a constraint between the two tables. The missed constraint hampers the accuracy as well as the understanding of the stored data.

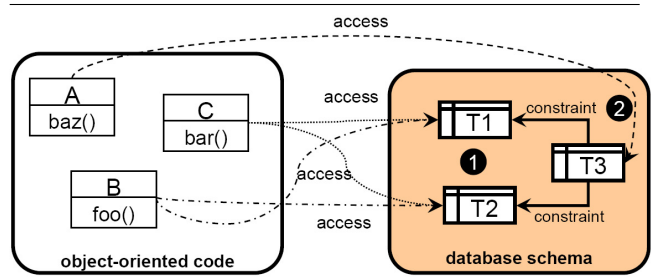


Figure 1. Two types of discrepancies.

Incomplete Data Usage. The second case of relational discrepancy appears when there is an explicit constraint defined in the database schema, but it is disregarded systematically in the source-code, *i.e.*, the classes don't access the tables jointly. Looking again in Figure 1 we notice that while there is a constraint defined among T1 and T3 no class is accessing both T1 and T3 together (directly or indirectly); instead, class A which accesses table T1 does it without using the correlated data found in T3. This means that the database might contain unused data which increases the necessary effort for understanding the stored data.

While both cases of discrepancy are very interesting, due to the space constraints of this paper we focus the remaining discussion on the first discrepancy case, *i.e.*, *Missed Constraint*.

3.1. Missed Constraint: An Example

In order to explain more clearly this case of relational discrepancy, let us consider a simple example. Considering an application that manages a library, let's assume that all the information regarding the library is stored in a relational database called `Library` which contains, among others, the following three tables, depicted in Figure 2.

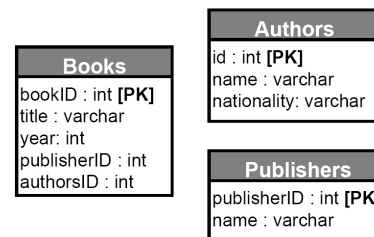


Figure 2. The `Library` database schema.

By looking carefully at Figure 2 we see that, although apparently there are no correlations among the three tables, two apparent links can be noticed in table `Books`, links that lead to the other tables (*i.e.*, columns `publisherID` and

authorsID, meaning that every book has an author and a publisher). Unfortunately, when the database was created, the two integrity constraints (foreign keys) that would ensure the integrity of the data stored within the table Books were not specified. Because these foreign keys are missing, we can't know for sure if fields publisherID and authorsID from table Books are correlated with fields in table Publishers resp. Authors. While this issue appears rather obvious in this example, this is mainly because: (i) the example is simple and the domain of the modeled data is well-known; (ii) the naming of correlated fields helps us. But in general, finding these missing correlations between tables, by looking exclusively at the database schema is hardly possible. Yet, knowing these integrity constraints is in practice an essential clue towards understanding and maintaining a large-scale system. That's why, in this paper we aim to detect these missing constraints, by looking at the way tables are accessed in the source-code.

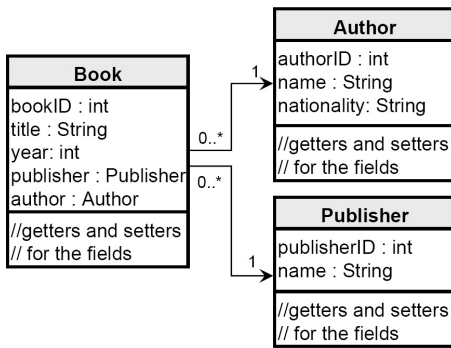


Figure 3. The Domain Classes.

Going on with the example let's take a look at the source-code, where these tables are used. In an enterprise application we find *domain objects* [9] which contain data from the database, objects which are stored/retrieved in/from the database by means of dedicated classes belonging to the data source layer, e.g., BookDS and PublisherDS. By applying Keller's *Foreign Key Aggregation* pattern [11] for our Library example we define the domain classes depicted in Figure 3.

We notice that a separate class was created for each table from the database, and that the Book class has references to the Author and Publisher classes.

Now (re)creating a Book object from the database implies retrieving data from the three aforementioned tables (Figure 2). In general, if more than two tables are involved there are two main approaches for doing this: (i) perform a single join between all the tables, or (ii) access separately each table and in the end set the references between the created objects [11]. Each of the two approaches has its own

pros and cons, but in practice both are widely used, and oftentimes we can encounter a combination of the two idioms. For the sake of revealing all the facets of the problem, in

```

class BookDS {
    public Book getBook(int id) { ...
        query = "SELECT * from Authors AS A,Books AS B" +
            " WHERE B.bookID=" + id +
            " AND A.id = B.authorsID";
        rs = statement.executeQuery(query);

        // create book object and set fields
        Book aBook = new Book();
        aBook.setBookID(rs.getInt("bookID"));
        aBook.setTitle(rs.getString("title"));
        aBook.setYear(rs.getInt("year"));

        // create author object and set fields
        Author theAuth = new Author();
        theAuth.setAuthorID(rs.getInt("id"));...
        // add author reference to book object
        aBook.setAuthor(theAuth);

        // get publisher object and set the
        // reference to it for the created book
        aBook.setPublisher(
            new PublisherDS().getPublisher(id));
        return aBook; }
}

class PublisherDS {
    public Publisher getPublisher(int id) {
        ...
        query = "SELECT * from Publishers " +
            "WHERE publisherID=" + id;
        rs = statement.executeQuery(query);

        // create publisher object and set fields
        Publisher aPublisher = new Publisher();
        aPublisher.setID(rs.getInt("publisherID"));
        aPublisher.setName(rs.getString("name"));
        return aPublisher; }
}
  
```

Figure 4. BookDS and PublisherDS classes.

our example we use such a *combined* solution for creating a Book object, as seen in Figure 4.

By looking at method getBook we first notice the SELECT statement and the join between the Books and Authors tables. This reveals us that the authorsID column in table Books is *de facto* a foreign key on table Authors. But this is unfortunately an *implicit constraint*, i.e., an information that we can't find explicitly in the database schema (Figure 2)!

Second, we notice that a Book domain object also needs a reference to a Publisher domain object, and it obtains it by calling the getPublisher method from PublisherDS class; more precisely, the Publisher object is created separately, by accessing table Publishers from class PublisherDS, and then its reference is attached to the Book object. This re-

veals an even more hidden dependency, *i.e.*, the one between the `Books` and the `Publisher` tables. Again, the database schema didn't make this correlation explicit.

In conclusion, starting from a class/method belonging to the *data source layer* it is possible to identify the group of tables that are used jointly, – called *group of related tables* both when the tables are accessed directly, but also when the correlation between tables is indirect (*e.g.*, via a method call). The ability to recover from the source-code all these correlations among the database tables, would provide us with useful information regarding the integrity constraints the existing schema contains, or should contain; and, which is most important, it helps us to understand the (oftentimes implicit) dependencies between the data manipulated in the system.

4. Detection of Relational Discrepancies

Before describing the technique we developed for detecting the *Missed Constraints* discrepancy, we first need to summarize the main entities and their interrelationships involved in the approach.

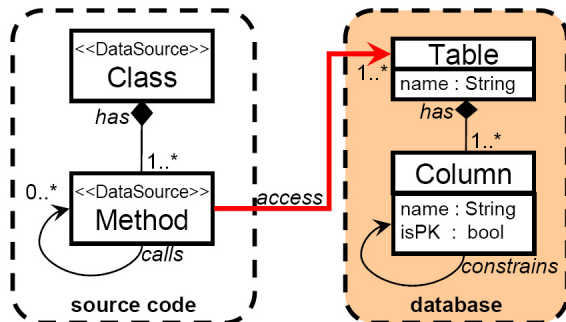


Figure 5. The main design entities that connects the source-code with the database.

Figure 5 shows that on the source-code side we mainly deal with *methods* that belong to *classes*. The main source-code relation that we are interested in is that a method can *call* (or resp. be *called by*) other methods. On the database side, we find *tables* that have a number of *columns*. Columns have a name and can also have the special property of being *primary key* for a table, *i.e.*, the values for that column are unique within the table. Concerning the database relations, we are interested in the constraints defined by means of *foreign keys* among the columns belonging to different tables. On top of everything, the relation that we are most interested in, is the one that connects the two “worlds”, *i.e.*, the source-code and the database. This connection appears in methods – belonging to the *data source*

layer – that access one or more tables by building and executing queries on the database (as seen in Figure 4).

Based on this model, we can now describe the detection process, which implies the following next steps.

4.1. Build the Groups of Related Tables

The main goal of this first step is to find the tables which are related by the fact that they are accessed within the same usage context, at the source-code level. As we have seen in the `Library` example, both the `BookDS` and the `PublisherDS` classes are directly or indirectly related with all the three tables.

A key element of our approach is that we consider not only the direct usages of tables, but also the *indirect* ones. Consequently, for each method *M* belonging to the *data source layer* (*i.e.*, methods that access tables) the group of related tables (GRT) is built as follows:

- All tables accessed *directly* from *M*, by means of queries built and executed within the method are added to GRT(*M*).
- For each data source layer method **called from** *M*¹, we add to GRT(*M*) all the tables *directly* accessed from the called method.
- For each data source layer method which **calls method** *M*, we add to GRT(*M*) all the tables *directly* accessed from the method that calls *M*.

For our initial example, both GRT(`BookDS`) and GRT(`PublisherDS`) will contain all three tables, *i.e.*, `Books`, `Authors` and `Publishers`.

4.2. Detect “de Facto” Constraints

The fact that a pair of tables are used from the same usage context does not necessarily mean that there is a relational constraint between the two. Therefore, the next step is to discover possible interrelationships among the tables found in the GRT's built for each data-source method, during the previous phase.

Oftentimes there is no relational constraint between a table from a GRT group and any other table from that group; furthermore, obviously, constraints might not be bi-directional. For example, table `Publishers` is not connected to table `Books`, it is only the other way around, the relation is going only from `Books` to `Publishers`.

In order to discover the relational constraints from the source-code, we use a set of heuristic rules related mainly to the *naming* of tables and columns. Aiming to discover interrelationships between all the related tables, we take each *pair of tables* from every GRT built in the previous step.

¹ Of course, recursive calls are excepted.

Considering a pair of tables (T1, T2) we claim to have found a *constraint from T1 to T2* (i.e., T1 depends on T2), if there exists at least one pair of columns (CT1 - belonging to table T1, CT2 - from table T2) that satisfies one of the following naming conditions:

- the name of CT1 contains or is equal, – based on a case insensitive comparison – with the name of table T2; CT2 is (part of) the primary key of table T2; and CT1 and CT2 have the same type. In our example, `authorsID` column in `Books` contains the name of the `Authors` table, `Books.authorsID` and `Authors.id` are of the same type, and `Authors.id` is the primary key of table `Authors`.
- CT2 is (part of) the primary key of table T2, CT1 and CT2 are of the same type, and the name of CT1 contains or is equal with the name of CT2, based on a case insensitive comparison. In our example, the `publisherID` column in `Books` has the same type and the same name with the primary key of table `Publishers`.

Remark. Clearly, our approach relies heavily on naming rules. From this point of view, the heuristics above appear to be rather fragile. Yet, both our practical experience and our case studies (see Section 6) suggest that in most projects the database schema relies heavily on naming conventions. These naming rules might be slightly different than the ones we used, but this is one point where our technique can be easily adapted to comply to further such heuristics.

Still, we have to admit that unfortunately we encounter sometimes hidden dependencies among columns, that can't be captured using these naming rules (e.g., two columns called `StudentID` and the other `PersonID`). For such cases we could enhance the correlation heuristics, by relying on the relations found in an ontology, – like `WordNet`² – as done in a recent reverse engineering approach [19].

4.3. Identify the Missed Constraints

The input of this phase is provided by the group of the pairs of tables for which in the previous step we found an interrelationship. Some of these found relations might have already been defined within the schema of the database. Consequently, in order to find out only the discrepancies between database schema and source-code, we are going to inspect each possible relation from the group and if the relation has already been defined, we are going to remove it from the group.

For this last step we rely on the model of the involved database, more precisely on the constraint relations (see Figure 5) in order to find out if the inspected constraint has

been defined in the schema. If the relation is missing then we assume that we have successfully identified a *Missed Constraints* relational discrepancy.

5. Tool Support

Applying manually the approach presented so far is unfeasible for large-scale enterprise systems; therefore, we developed an adequate tool support for our technique. From this point of view, the key issue is the definition and extraction of a proper model that captures in an unitary manner the object-oriented and the relational “worlds” and, most important, the relations between the two (see Figure 5). Thus the model extraction consists of three parts, as it follows:

Extract the Source-Code Model. In order to extract the object-oriented model of the source-code we need a meta-model which specifies the relevant design entities (e.g., classes, methods, attributes) found in the modeled paradigm. In our tool support, we use the `MEMORIA` [18] meta-model.

The model of the source-code contains, for each design entity, an instance of the class representing that particular entity in the meta-model which is created by a model loader.

Extract the Database Schema Model. The used meta-model for representing entities found in a relational database contains a class (called `Table`) that models the table design entity. It has a name and a field of type `Column[]` that establishes its connection to the columns the table contains. The class `Column` has several elementary properties (e.g., name, type, dimension, `defaultValue` and properties regarding constraints – `is-Null`, primary and foreign keys).

We have created a tool called `DATES`³ for automating quality assurance analyses that we define, like the one presented in this paper or the one described in [13]. As part of `DATES` we developed a meta-model and a model extractor for relational databases, in particular `MySQL` and `Microsoft Access`.

Extract Code-Database Interactions. We mentioned earlier that the methods that ensure the communication with the relational database belong to a layer called *data source*. Thus, finding the interactions between the object-oriented and relational entities requires the identification of the methods that belong to the data source layer.

In order to find out which methods belong to the data source layer we take a simple approach to this issue, by taking into account the various usages of third-party libraries and/or frameworks that are specific for the data source layer. In this context, a method is considered to belong to the data

² <http://wordnet.princeton.edu>

³ Design Analysis Tool for Enterprise Systems

source layer if it invokes one or more methods from a specific library that provides an API for accessing and processing data stored in a data source, usually a relational database (e.g., `executeQuery()` from the `java.sql` package).

We enriched the `Method` entity from the object-oriented MEMORIA [18] meta-model in order to make it store information regarding the operations the method performs upon the database – e.g., delete, insert, select, update. We store within this design entity the group of *accessed tables* from the performed database operations within. This enrichment of the MEMORIA meta-model is also part of our tool, namely DATES.

We decided to integrate DATES within the IPLASMA [15] environment, as this has facilitated the implementation of the three steps of our approach described in Section 4. Furthermore, the integration will help us to take advantage of the quality assurance analyses already defined for object-oriented systems.

Currently, the found missed constraints are saved in a text file, from where they can be manually applied on the schema of the database within the analyzed enterprise application.

6. Evaluation of the Approach

In order to evaluate the approach we have conducted different experiments on several enterprise applications. In this section we present the results obtained by applying the tools described in the previous section upon two of these enterprise systems. Some characteristics of the analyzed systems regarding their size, number of classes(NC), methods(NM) and tables(NT) are summarized in Table 1.

System	Size(bytes)	NC	NM	NT
Payroll	534,861	115	580	12
CentraView	11,162,565	1527	13369	217

Table 1. Some Characteristics of the Case Studies.

Payroll is an industrial enterprise application whose scope is to manage information about the employees from a company. *CentraView* is an open-source enterprise application which provides growing businesses with a centralized view of all customer and business information⁴. In Table 2 we present more characteristics of the analyzed systems as well as the number of the identified missed constraints between database schema and source-code. Next,

⁴ <http://www.sourceforge.net/projects/centraview>

for each system we are going to discuss the obtained results (the found missed constraints).

	Payroll	CentraView
Data Source Classes	16	836
Columns	89	1281
Primary Key	12	170
Foreign Keys	0	28
Missed Constraints	7	219

Table 2. More Characteristics and the Identified Discrepancies.

6.1. The Payroll Application

As we can see in Table 2, the *Payroll* application contains none foreign key. By applying our introduced technique we obtain 7 discrepancies between database schema and source-code. In order to find out whether the discrepancies were correctly identified we performed a manual investigation. Identifying correctly the relations between the entities from the existing relational database in an enterprise application means that among the entities that were classified as being/not being related by foreign keys there are no false positives and no false negatives.

During the manual investigation, we browsed through each of the 12 existing tables in order to find out which interrelationships exist among them. We have discovered 8 interrelationships, 7 of them being discovered by the approach. For example, the approach reports us that

- every stored history regarding salaries is attached to an employee (table *salaryhistory* should have a foreign key *employee* that refers column *Id* from the table *employee*).
- every stored evaluation belongs to an existing employee (there is a table called *evaluation* which should have a foreign key *employee* that refers column *Id* from the table *employee*).

Regarding the missed constraint which was not discovered by our approach, we found out that the false negative is due to the fact that one of the tables involved in the unreported constraint was *not used in the source-code*. Probably, it would have been good to apply in this case the *Drop Table* refactoring pattern presented in [1].

6.2. The CentraView Application

This application, as presented in the beginning of this section, is a large-scale enterprise application. By apply-

ing our approach, the tool found 219 missed constraints between database schema and source-code. Due to the fact that *CentraView* is significantly larger than *Payroll*, a manual investigation of the 217 tables is not feasible. Thus, we performed a “smart” manual investigation based on a subset of the interconnected tables, chosen using the following strategy – we manually removed from the database schema the 28 existing foreign keys aiming to see if by applying our approach, we will find these initially defined interrelationships among the 20 tables.

By applying our technique, we found only half (14) of the initial set of 28 relational constraints initially defined on the database schema. Thus, 14 relations were missing from the reported list and, therefore, we performed an in-depth analysis in order to find out the causes. We discovered that 11 out of the 14 initial relations that our approach did not detect, are missing from the reported list due to the fact that 8 of the tables involved in these foreign key were simply not used in the source-code! Probably the *Drop Table* pattern [1] is again recommendable in this case.

Regarding the remaining 3 undetected relations, we found out that they were not found because our matching heuristics did not apply for the involved columns. Yet it is worth mentioning that the 3 foreign keys were all defined between the same two tables, *i.e.*, the *search* resp. the *individual* table. The three undetected relations are foreign keys on columns *OwnerBy*, *CreatedBy*, *ModifiedBy*, all three referring to column *IndividualID*. By analyzing these column names, we are confident that our approach can be enhanced by enriching the matching heuristics with additional relations extracted from an ontology, like we suggested at the end of Section 4.2.

Apart from that “smart” manual investigation we took a closer look on the 219 reported constraints that should be applied upon the schema of the database and we present in Figure 6 some of them. Due to a high degree of understanding provided by the names of tables and columns, we can quickly find out the meaning of most discovered missed constraints. Thus, we expect more tasks to belong to a project (1), to encounter more invoices at the same address (2), an individual to have more tickets (3), an emailmessage to have none or more attachments (4) and an individual to attend more activities (5).

Regarding the three relations 5 - 7, we noticed that all of them should be applied upon the table *attendee* (*i.e.*, it is possible to discover more discrepancies that should be applied on the same table - this explains why it is possible to have 219 missed constraints which should be applied, due to the fact that at least 8 tables are unused, on less that 217 tables).

It is quite easy to see that relation 5 has been correctly identified but at a first sight relations 6 and 7 are ambiguous because they should be applied on the same column *Ac-*

1.	T:task T:project	C:ProjectID C:ProjectID
2.	T:invoice T:address	C:billaddress C:AdressID
3.	T:ticket T:individual	C:individualid C:IndividualID
4.	T:attachment T:emailmessage	C:MessageID C:MessageID
5.	T:attendee T:individual	C:IndividualID C:IndividualID
6.	T:attendee T:activity	C:ActivityID C:ActivityID
7.	T:attendee T:applicationform	C:ActivityID C:ActivityID

Figure 6. Some discovered related tables.

tivityID which should refer two different tables (*activity* and *applicationform*). In this case we took a closer look at the two last tables and we discovered that table *activity* contains information regarding an activity (*e.g.*, type, priority, status, owner, DueDate, CompletedDate) and table *applicationform* contains also related information regarding an activity (*e.g.*, salaryactual, assessmentneg) and even the primary key for table *applicationform* is called *ActivityID* – this case is nothing but a common practice for a foreign key to refer more than one table. Moreover, searching in the report provided by the introduced approach, we discovered that when a foreign key should refer column *ActivityID* from table *applicationform*, it always should refer also column *ActivityID* from table *activity* – we encounter this situation 5 times.

7. Related Work

A first category of approaches related to this topic is the one whose input is provided exclusively by the involved relational database. An example of such approach is *Referential Integrity Utility for IBM DB2 Cube Views* [7], a tool which detects missing primary keys, missing foreign keys, nullable foreign keys and generates data definition language to add the necessary DB2 informational constraints. The main difference between this approach and our approach resides in the fact that our approach will provide us only with those discrepancies which affect the analyzed enterprise application (*i.e.*, an approach based exclusively on a database schema may provide a lot of relational discrepancies upon tables which are not used by the source-code and, consequently, which do not affect the enterprise application).

Another category of approaches which helps in order to find out implicit data constraints among the involved

database is based on program slicing, like the one proposed in [5]. This approach is applied upon enterprise applications which communicate with relational databases via embedded SQL (e.g., SQL code written between EXEC SQL and END-EXEC), approach which is not suitable for enterprise applications which communicate with relational databases by SQL statements embedded in a string that are executed by methods like executeQuery().

Our approach, like the contribution from [6], uses a proprietary meta-model for representing relational entities, despite the fact that the Object Management Group (OMG) owns the Common Warehouse Metamodel (CWM) [17], a meta-model intended to bridge the gap between dissimilar meta-models by providing a common basis for meta-models. For our needs, we find the CWM meta-model very complex (e.g., it contains entities like triggers/stored procedures which we do not use) but in the future it might replace our meta-model.

Currently there are also enterprise applications where the persistence layer is manipulated by frameworks like Hibernate [3]. In this context we want to emphasize that our approach can be applied upon enterprise applications which use such frameworks, but in this case the information regarding the connections between the object-oriented part and the relational part will be extracted (and, consequently, we need to develop and use a different model loader) also from existing configuring XML files.

8. Conclusions. Future Work

The main features of the introduced approach are: (1) it allows us to identify automatically the related tables from which data are retrieved in the source-code, (2) based on the related tables, it identifies possible interrelationships between relational tables (e.g., Missed Constraints).

The aforementioned features requires the use of a specific meta-model for enterprise applications (the model we developed in [14]) which captures in an unitary manner the object-oriented and the relational paradigms and, most important, the relations between the two.

We conducted an experiment in which we automatically identified interrelationships between tables in enterprise applications, interrelationships which were not explicit defined in the schema of the involved relational table.

We intend in the future to: (1) extend the tool support in order to be able to use it upon enterprise applications written using other technologies (.NET, different persistence providers, different communication techniques), (2) continue the evaluation of the introduced approach against other enterprise applications.

Acknowledgments. This work is supported by the Romanian Ministry of Education and Research under Projects CNCIS TD(GR76/23.05.2007) and CEEEX(3147/11.10.2005).

References

- [1] S.W. Ambler and P.J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006.
- [2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Lessons learned in applying formal concept analysis to reverse engineering. In *Proc. International Conference on Formal Concept Analysis*, 2005.
- [3] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications, 2007.
- [4] E. Chikofsky. *The Necessity of Data Reverse Engineering - Preface to Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1996.
- [5] A. Cleve, J. Henrard, and J.L. Hainaut. Data reverse engineering using system dependency graphs. In *Proc. Working Conference on Reverse Engineering*, 2006.
- [6] I. de Guzman, M. Polo, and M. Piattini. An integrated environment for reengineering. In *Proc. IEEE International Conference on Software Maintenance*, 2005.
- [7] C. Baragoin et al. *DB2 Cube Views: A Primer*. IBM International Technical Support Organization, 2003.
- [8] K. Brown et al. *Enterprise Java Programming with IBM Websphere*. Addison-Wesley, 2001.
- [9] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [10] J.L. Hainaut, J. Henrard, J.M. Hick, D. Roland, and V. Englebert. The nature of data reverse engineering. In *Proc. Data Reverse Engineering Workshop*, 2000.
- [11] W. Keller. Mapping objects to tables: A pattern language. In *Proc. European Conference on Pattern Languages of Programs*, 1997.
- [12] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.
- [13] C. Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proc. IEEE International Conference on Program Comprehension*, 2006.
- [14] C. Marinescu and I. Jurca. A meta-model for enterprise applications. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2006.
- [15] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Rațiu, and R. Wetzel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance (Industrial and Tool Volume)*, 2005.
- [16] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison-Wesley, 2003.
- [17] J. Poole, D. Chang, D. Tolbert, and D. Mellor. *Common Warehouse Metamodel: An Introduction to the Standard for Data Warehouse Integration*. John Wiley & Sons, 2001.
- [18] D. Rațiu. *Memoria: A Unified Meta-Model for Java and C++*. Master Thesis, Politehnica University of Timișoara, 2004.
- [19] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Proc. Working Conference on Reverse Engineering*, 2006.