Discovering the Objectual Meaning of Foreign Key Constraints in Enterprise Applications

Cristina Marinescu
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
cristina.marinescu@cs.upt.ro

Abstract

*The software industry is increasingly confronted with the issues of understanding and maintaining a special type of object-oriented systems, namely enterprise applications. A specific concern for these applications is to understand the persistent data (usually stored in a RDBMS), and its manipulation in the object-oriented source code. While foreign keys are an important means for indicating relations within the persistent data, oftentimes, by looking solely at the database schema, it is impossible to determine the exact nature of these relations. This paper proposes a novel approach for determining a refined understanding of the relations among the persistent data, by correlating the information about foreign keys extracted from the database schema with the way the data are used in the source code. By analyzing two enterprise systems we found that the proposed approach helps specifying a significant number of foreign key constraints in terms of their objectual meaning (*e.g.*, if they denote an inheritance or an aggregation relation). Thus, the approach contributes to enhancing, in an automated manner, the understanding of a system's database schema by tying it to the source code that uses it.*

Keywords: data reverse engineering, foreign keys, RDBMS model, object-oriented model, enterprise applications

1. Introduction

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 [12]. Due to these characteristics, 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 persistence of the involved data. As these systems are more and more encountered in the software industry, the issues of understanding and maintaining them become increasingly important.

It has been repeatedly emphasized [1, 12, 22] 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 significantly increased.

A substantial number of reverse engineering methods and quality assessment techniques have been developed for object-oriented systems [3, 13, 16]. But, when it comes to understand and evaluate the quality of the relationships existing within the relational part of the system (*i.e.*, the persistence layer) and of the relationships 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 that help an organization determine the structure, function, and meaning of its data [6]. DRE is a complex and expensive task and needs to be supported by program understanding and tools [7].

In this paper we propose a novel approach for capturing the semantics of the relations between the elements of the persistence layer within an enterprise application, by correlating the information extracted from the database schema with the insight provided by the usage of the database elements within the source code. The approach enriches the semantics of database relations established by means of foreign keys by revealing its "objectual" meaning (*e.g.*, if the foreign key denotes an inheritance or an aggregation relation). Thus, the approach has a twofold benefit:

1. helps to refine the understanding of a system's database, leading to an easier maintenance and evolution of the database schema
2. helps to better understand the key elements of the business model as these elements are the main constituents of the database schema.

The paper is structured as follows: in Section 2 we present a brief overview of enterprise applications. Next (Section 3) we state precisely the problem addressed by the paper. In Section 4 we describe in detail the elements of the proposed approach. In Section 5 we provide a brief description of the tool support that ensures the automation of the entire approach. In Section 6 we describe and discuss the evaluation 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

The main characteristics of enterprise applications are that they usually involve a significant amount of persistent data, and user interface screens, together with users that manipulate the data concurrently [12]. Examples of such systems are payrolls, insurance, or patient records systems.

Within such applications a key design practice is to decouple the commands that ensure the persistence (usually the persistence is supplied by a database, mostly relational database) and application logic because mixing them hampers understanding and testing the application. The goal is to be able to change the user interface (graphic UI or HTML based-browser) or the persistence provider without affecting the rest of the application i.e., the application logic. These constraints lead to a multi-layered architecture, consisting of three primary layers, namely the *data source*, the *domain* and the *presentation* layer [10].

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#). Consequently, the main source code entities, belonging to the aforementioned layers, are those which are common for the object-oriented paradigm (*e.g.*, methods, classes). Our second assumption is that persistence is provided by a relational database (*e.g.*, MySQL, DB2) 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 *execute Query(String sql), executeUpdate(String sql)*.

3. Refining Database Schema Relations

In an enterprise application the two most reliable sources of information for knowledge recovery are the database schema and the source code. For each of these there is a good and a bad part:

- In a RDBMS, while the **database schema** usually captures well the key model elements – as this is mainly what is kept persistent – the ability to express *relationships* (*e.g.*, using foreign keys) is rather weak [15], being almost impossible to express precisely the type of that relation (to differentiate between a part-whole and a hierarchical specialization relation).
- Concerning the **source code** the good part is that the semantics of relationships that can be expressed is much richer than in the case of database schemas (*e.g.*, the explicit inheritance relation, or the containment relation expressed using instance variables [9]). Yet, at the source code level we have another problem: the elements of the business model are "cluttered" with implementation details, and this makes it hard to distinguish between the source code elements (*e.g.*, classes and methods) that correspond to the business model and those that are strictly used to support the concrete implementation.

Consequently, in order to recover the relationships among business model elements, and thus recover the understanding of the modeled domain, we need to combine information both from the database schema and the source code. In this context, we propose a two step approach to recover these relations:

1. **Step 1**: We use the information (*i.e.*, tables, columns, foreign keys) from the database schema to focus (*i.e.*, reveal) the classes that store the data from the database. We calls these classes *Data Keepers* (see Section 4);
2. **Step 2**: We extract from the source code the relationships among *Data Keepers* and use them to refine the semantics of the foreign key constraints defined among various tables.

3.1. Relations between Tables and Classes

Considering a table `T`, there can be several ways it can be linked with the source code:

- *Table is not accessed* in the source code. In this case probably the table is just "dead code" and therefore it is useless to understand its structure and function. In our approach we denote such tables as being in a N(**N**ot Used) relation with the source code. Usually this is a case where the *Drop Table* refactoring [2] could be applied.
- *Table is used* in the source code. In this case we say that between table `T` and the source code there is an U (**U**sed) relation and consequently is worth looking closer look at `T`.

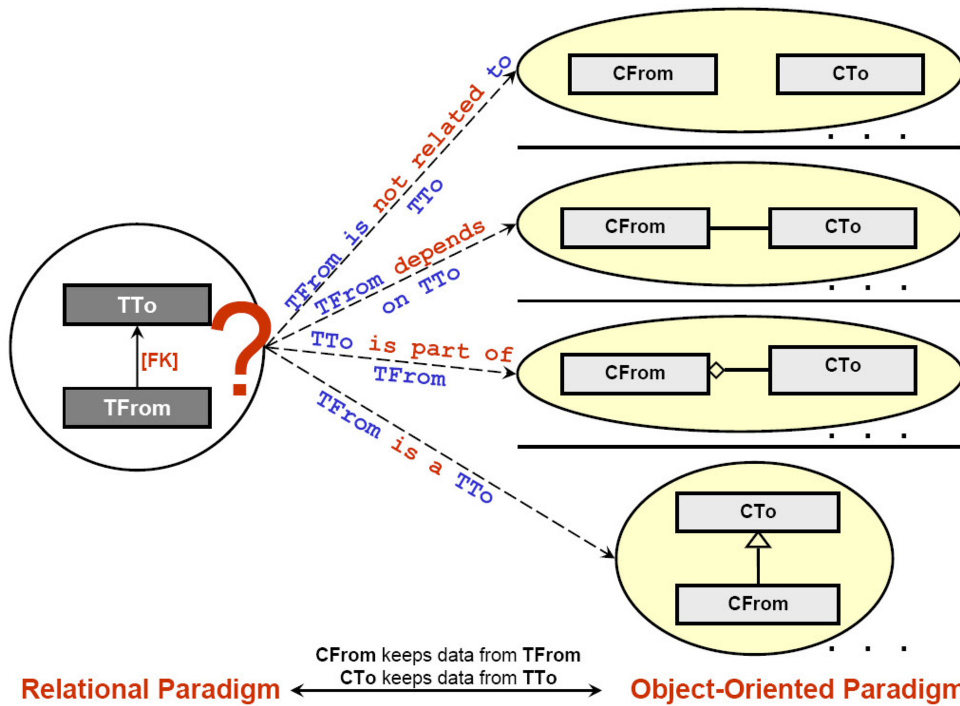

Figure 1. The objectual meaning of a foreign key, as revealed in the source code.

- A special case of usage is when there are one or more classes whose instances *store data from* `T`. As mentioned earlier, we call those classes *Data Keepers* [1] and denote the relation between table `T` and the object-oriented code as a S (**S**toring Data) relation. Knowing the related *Data Keepers* for a table `T` provides us with additional information about the meaning of the data within table `T`; furthermore, the methods of these *Data Keeper* classes might provide us with information regarding the function of the same data.

3.2. Refined Semantics of Foreign Keys

One important relation that connects the elements (*i.e.*, tables) of a relational database schema are the *foreign keys* constraints. But, as we mentioned before, the semantics of this relation is very vague as it can denote anything from a simple dependency to a part-whole relation between the concepts represented by the two tables.

In order to illustrate the potential semantics of a foreign key relation, let's assume having two tables `TFrom` and `TTo` connected by a foreign key which sets a constraint on the data from table `TFrom` with respect to the data from table `TTo`. Let us also assume having a *Data Keeper* class for each of the two tables – `CTo` for `TTo` and `CFrom` for `TFrom`. The data from these two tables connected by the foreign key may be used in the source code in different ways, revealing thus various semantics for the foreign key (see Figure 1):

- There is no interaction between `CFrom` and `CTo`. In this case we consider that this reflects also the relation among the tables, being possible that table `TFrom` contains unused data.
- Methods from `CFrom` call methods from `CTo`. In this case we consider that, as well as among the *Data Keeper* classes, between the two tables related by the foreign key there is a D (**D**ependency) relation. Consequently, the meaning of the foreign key is that data from `TFrom` *is used together* with data from `TTo`.
- `CFrom` and `CTo` are in a part-whole (aggregation) relation. In this case we consider that the A (**A**ggregation) relation characterizes also the tables related by the foreign key. Consequently, the semantics of the foreign key is that the data stored in `TTo` *is part of* the data stored in `TFrom`.
- `CFrom` is derived from `CTo`. In this case we interpret that the data stored within `TFrom` *is a* specialized data of `TTo` and, consequently, an I (**I**nheritance) relation occurs.

1 we do not use the *Data Transfer Object* [12, 21] term because a *Data Keeper* might have both access the database and in the same time define some functionality; but note that every DTO is a *Data Keeper*.

The approach described in detail in the next section enables the detection of the objectual meaning of foreign keys, *i.e.*, it discriminates between the aforementioned D, A and I relations. More precisely, the approach "labels" each foreign key with a description of its objectual meaning in form of a 6 characters string (see Figure 2). The first character denotes if the foreign key relation is explicitly defined in the database schema or if it is inferred from the joint usage of tables in the source code [18]. The next two characters specify the relation of each of the two tables with the source code (the N, U or S relations). The last three characters appear in the characterization, if and only if the relations among the tables and the object-oriented part have the S type (*i.e.*, for each of the two tables, there is at least one *Data Keeper* class). Note, that a foreign key relation might have none or all the three objectual meanings.

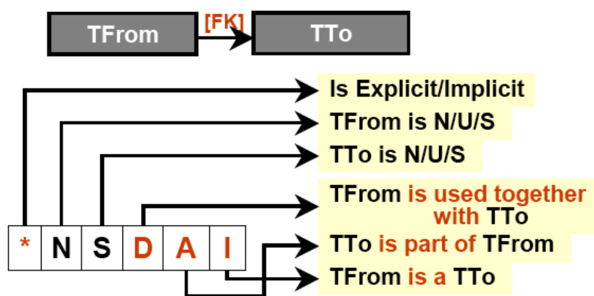

Figure 2. Objectual meaning of foreign keys.

Remark. Often we encounter situations where the database schema relies heavily on naming conventions and in this context, if there it is a foreign key between a table called *Person* and a table called *Student*, we expect an I objectual meaning and, thus, we can consider superfluous our approach. But it may happen that this assumption is false, the data from the tables being totally unrelated by an I RELATION, and our approach can confirm/infirm every assumption of this type we make.

4. Recovering the Meaning of Foreign Keys

Before describing the technique we developed for detecting objectual meaning of foreign keys in enterprise applications, we first need to summarize the main entities and their relationships involved in the approach.

Figure 3 shows that on the source code side we mainly deal with *methods* and *attributes* that belong to *classes*. A class might be descended from none or more classes. One of the main source code relations that we are interested in is that a method can *call* (or be *called by*) other methods.

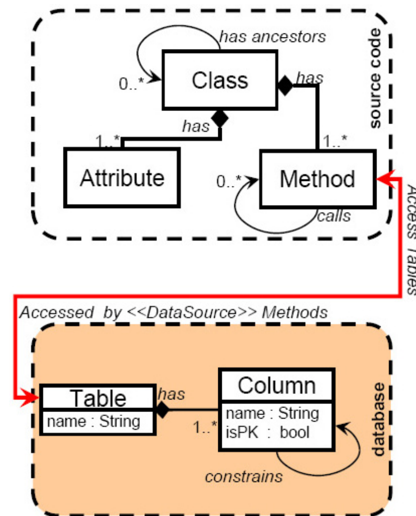

Figure 3. A meta-model of the main entities from source code and database.

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. Concerning the database relations, we are interested in the constraints (implicit or explicit) defined by means of *foreign keys* among the columns belonging to different tables. On top of everything, we are most interested in the relation 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, this relation being bidirectional (*e.g.*, a table is *accessed by* methods from the data source layer).

Based on this model, we present below how we discover *objectual meaning* of foreign keys in enterprise applications.

4.1. Detecting Relations between Tables and Code

As we have seen in the previous section, knowing the N, U and S relations is a mandatory step for recovering the meaning of foreign key relations. In terms of the aforementioned model (Figure 3), discovering the N and U relations between tables and source code requires counting the number of methods belonging to the data source layer which access each inspecting table. Consequently, if the number is zero, it means that we encounter a N relation, otherwise we have an U relation.

In order to find if there is a S relation between a table `T` and the source code, as well as the causes of the relation (*e.g.*, the *Data Keeper* classes corresponding to `T`), three

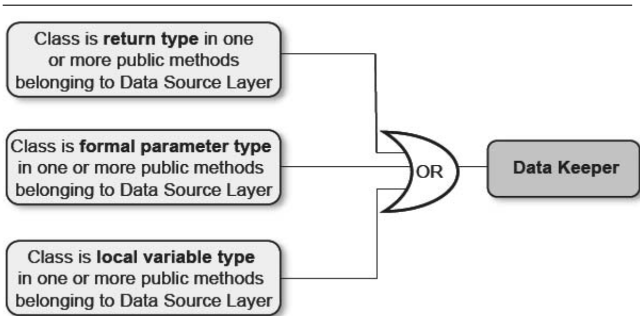

Figure 4. Detection of a *Data Keeper* class.

operations need to be performed: (i) detect all *Data Keeper* classes in the source code; (ii) for each *Data Keeper* find the related tables *i.e.*, the tables from which the *Data Keeper* stores/retrieves data; (iii) discover for each table the set of corresponding *Data Keepers*, based on the previous two operations.

Operation 1: Detect *Data Keeper* classes. *Data Keepers* are classes used to "transport" data from/to the database. The instances of these classes are usually created and used from methods belonging to the *data source layer* of the enterprise application *i.e.*, from those classes that directly interact (by querying or modifying) with the database. In order to find out the methods from the data source layer we use our approach from [17]. It takes into account the various usages of third-party libraries and/or frameworks that are specific to the data source layer. In this context, a method is considered to belong to the data 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., for systems written in Java, the method invokes the `executeQuery()` method from the `java.sql` package).

Because *Data Keepers* store the data transported from/to the database their detection is done by visiting all the methods belonging the data source layer and considering all the classes that are used in these methods as: (i) *return types* – for the case where data extracted from the database must be passed to the other layers of the system; (ii) *parameter types* – for the case where data is passed to the data source layer in order to be stored in the database; (iii) *types of local variables* – for the case where multiple data having the same type are extracted from the database and passed to the other layers embedded in a predefined collection. Figure 4 summarizes the complete detection rule of a *Data Keeper*.

Operation 2: Find tables related to each *Data Keeper*. We consider for each *Data Keeper* the following tables as being related:

- Tables accessed from each *data source layer method* that either **calls** or **is called by** methods from the *Data Keeper*. The reason for this rule is that it ensures finding the tables from which data kept by the *Data Keeper* are stored and respectively retrieved.
- the tables *directly accessed* by the methods from the *Data Keeper*. This last case might look confusing, as we would expect for each *Data Keeper* class to belong only to the domain layer and thus not be responsible for storing/retrieving data in/from tables. But note that we reverse engineer legacy enterprise applications where clear initial design intentions might have become "blurred" during implementation. Thus, we expect at least some of these *Data Keepers* that access also the database tables to reveal signs of design flaws; yet, this quality assessment is beyond the scope of this paper.

Operation 3: Find the *Data Keepers* for each table. Finding for each table the set of related *Data Keepers* is just a processing step based on the results of the previous operations. The operation is done as follows: for each table `T` we iterate over the identified *Data Keeper* classes and for each of them check if the `T` is among the *related tables* which were identified during **Operation 2**. If for `T` at least a *Data Keeper* was found we mark table `T` as having an S relation with the source code (see Section 3).

4.2. Detecting the Semantics of Foreign Key Relations

Discovering the D, A and I RELATIONS between tables related by foreign keys relies on two categories of input data: (i) the foreign keys and (ii) the set of related *Data Keepers* for each table, which was built during the previous step (see **Operation 3**). In Figure 5 we present the algorithm for discovering the D, A and I relations, algorithm which has to be applied for each pair of classes, possible nested, (`CFrom`, `CTo`) – where `CFrom` belongs to the group of *Data Keepers* related to `TFrom` and `CTo` belongs to the *Data Keepers* related to `TTo`.

If `CFrom` has an attribute whose type is `CTo` it means that the data stored in table `TTo` *is part of* the data stored by `TFrom` and, consequently, between the two involved tables there is an A relation. The algorithm continues by checking if `CFrom` extends `CTo`, regardless the fulfilment of the previous condition (*i.e.*, it is possible for class `CFrom` to aggregate class `CTo`, as well as to be derived from `CTo`). If the second condition is reached we encounter an I relation. Eventually, we check if there are calls from methods of `CFrom` to methods of `CTo`, in which case a D relation is established the two corresponding tables (`TFrom`, `TTo`) related by a foreign key.

If none of the previous conditions are satisfied, it means that the object-oriented source code of the enterprise application contains data provided by tables `TFrom` and `TTo` re-

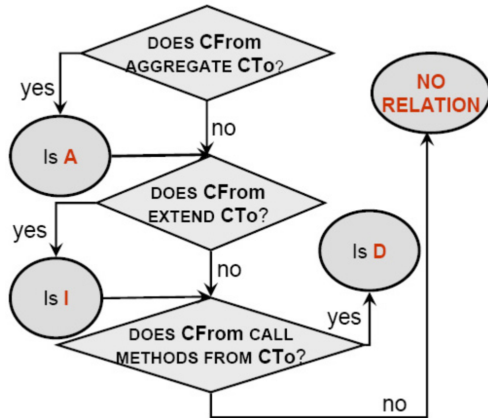

Figure 5. Discover the D, A and I RELATIONS.

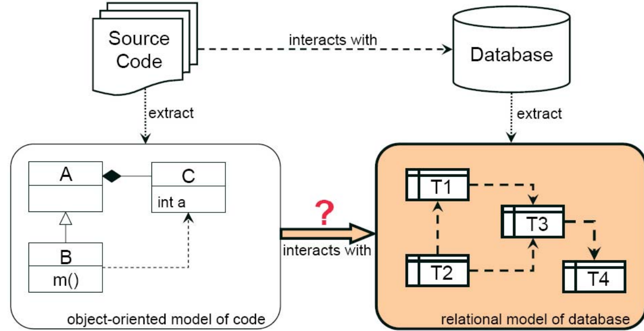

Figure 6. Design information must be extracted both from the source code and the database schema.

lated by a foreign key, but the constraint is not visible in the source code. It might be possible that only a part of the data contained by table `TFrom` is used.

As we can notice, we may encounter an A relation between `TFrom` and `TTo` without having a D relation as well. At first sight this is not a problem, but in a well-designed enterprise application we expect a class to aggregate another class in order to *call* the services provided by the aggregated class. Thus, if there are no calls, this probably means that among the two keepers that generate the relation the communication occurs via direct accesses of attributes, which is a violation of the encapsulation principle [11]. Yet, correlating design flaws with the types of relations between tables in the involved database is also beyond the scope of this paper.

5. Tool Support

A manual application of the approach presented so far is infeasible for large-scale enterprise applications; therefore, we have 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 entities from the object-oriented and the relational "worlds" and, most importantly, the relations between the two (see Figure 6). Thus the model extraction consists of three parts: (1) extract the model from the object-oriented source code, (2) extract the model from the database schema, and (3) extract the interactions between the object-oriented and the relational part of the enterprise system.

5.1. Extract the Source Code Model

In order to extract the model of the source code we need a meta-model which specifies the relevant design entities found in the modeled paradigm. In our tool support, we use the MEMORIA [24] meta-model.

The MEMORIA meta-model represents object-oriented entities as an interconnected set of classes, usually one for each type of design entity. The fields of a class modeling a design entity are either elementary properties of that design entity or links to other related classes. For example, a class that models the `Package` design entity is expected to have an elementary field which stores the name of the package and a field of type `Class[]` that establishes its connection to the classes the package contains.

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.

5.2. Extract the Database Schema Model

In order to extract the design entities from the database (*i.e.*, the model) we need a meta-model which specifies the found entities. In this case, the meta-model must contain entities for representing the tables and columns found in a relational database. Thus, the class (called `Table`) that models the table design entity 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 constrains – isNull, primary and foreign keys).

We have created a tool called DATES [2] for automating the entire approach. As part of DATES we developed a meta-model and a model extractor for relational databases, in particular *MySQL* and *Microsoft Access*.

2 Design Analysis Tool for Enterprise Systems.

5.3. 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* (see Figure 3). Thus, finding the interactions between the object-oriented and relational entities requires the identification of the methods that belong to the data source layer. As previously mentioned, we identify the methods from the data source layer using our approach from [17].

The `Method` entity from the object-oriented MEMORIA [24] meta-model, was not initially conceived to reflect the relation with the database; consequently, we had to enrich it with information regarding the operations performed on a relational database (*e.g.*, delete, insert, select, update). Furthermore, the `Method` meta-model entity needs to keep a reference to the group of *accessed tables i.e.*, the tables used by the method when performing database operations. We also enriched also the `Table` entity from the DATES meta-model with a field having the `Method[]` type, in order to store the methods that access a table.

We decided to integrate DATES within the iPLASMA [20], which is an extensible software analysis environment for object-oriented systems. This facilitates the implementation of our approach described in this section and, furthermore, it can help us to take advantage of the quality assurance analyses already defined for object-oriented systems.

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 on two real-world enterprise systems. Some characteristics of the analyzed systems regarding their size, number of classes, methods and tables are summarized in Table 1. In Table 2 we present more characteristics of the analyzed systems.

System	Size(KB)	Classes	Methods	Tables
Payroll	500	115	580	12
CentraView	11’000	1’527	13’369	217

Table 1. The Case Studies in Numbers.

Payroll is an industrial enterprise application used to manage information about the employees of a company and their payment. *CentraView* is an open-source enterprise application which provides growing businesses with a centralized view of all customer and business information [3].

	Payroll	CentraView
Data Source Methods	112	3349
Columns	89	1281
Primary Key	12	170
Foreign Keys	7	247

Table 2. Database Schema Characteristics.

For each system we are going to discuss next the main findings of refining the meaning of foreign keys, by using the approach presented in this paper.

6.1. Findings in *Payroll*

The *Payroll* application contains 112 methods identified as belonging to the data source layer, the involved data is stored in 12 tables and there are 12 primary keys (one primary key for each table) and no explicit foreign keys. Yet, when considering also the implicit foreign keys that are found by analyzing the co-usage of tables in the source code [18] we fond 7 (implicit) foreign keys.

By applying automatically our approach, only D (**D**ependency) relations were detected for the 7 constraints. This revealed following aspects:

- each table involved in the classified constraint is used in the source code and, moreover, there is at least a *Data Keeper* class associated to each table.
- the data stored in each involved pair of tables is used together (*e.g.*, when we increase the value stored in the *MonthlySalary* column belonging to table *salary-history* we expect this operation to be performed for a particular *employee*).

Performing an in-depth investigation regarding the way tables are used from the code, we find out that table *employee* is used together with other 6 tables (*e.g.*, *salary-history*, *evaluation*, *training*, *training_employee*) and table *training_employee* is used together with table *training*, the last relation being indirectly related to table *employee*. Due to the fact that every stored data is related to table *employee*, we can say that this table tends to centralize most of data in the database belonging to the *Payroll* system. In general this centralization phenomenon can be either a database design flaw, or an inherently central table, which is very important for the understanding of the entire database.

3 http://www.sourceforge.net/projects/centraview

In order to find out whether the relations 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 related – by D RELATIONS – there are no *false positives* (*i.e.*, entities erroneously identified by the approach as being related).
- among the entities that were not classified as being related there are no *false negatives* (*i.e.*, entities which are related – by A or I RELATIONS – and were not identified by the approach as being related).

Within the *Payroll* system the manual inspection did not reveal any false positives nor false negatives with respect to the automatic classification provided by our approach.

6.2. Findings in *CentraView*

This application, as presented in the beginning of this section, is a large-scale enterprise application which has 1527 classes, 3349 methods belonging to the data source layer and the involved data is stored among 217 tables. By taking into account both the explicitly foreign keys and the implicit constraints detected by analyzing again the co-usage of tables within the code, we identified 247 relations among tables.

From the 217 tables we detected only 115 as being used in the source code, 76 of these being connected in the source code to one or more *Data Keeper* classes. In Figure 7 we present some associations provided by the approach between tables and theirs *Data Keeper* classes. From the provided information, for example, we find out that [4]:

```
(1) Table:       literaturerequestlink
    Data Keeper: ActivityVO

(2) Table:       eventregister
    Data Keeper: EventAttendeeVO

(3) Table:       ticket
    Data Keeper: TimeSlipDBVO
```

Figure 7. Some discovered S RELATIONS between Tables and *Data Keeper* Classes.

- (1) the data from the *literaturerequestlink* table is related to an activity, and we expect the system contains different types of activities (class *ActivityVO* has a method *getType()*).
- (2) *eventregister* contains registrations for a particular attendee – class *EventAttendeeVO* contains information regarding an event as well as an attendee.
- (3) the information contained into the *ticket* table is related to a particular project (class *TimeSlipDBVO* has a method which provides the associated *projectID*, as well as information retrieved from the *ticket* table).

Among the 247 foreign keys for which we automatically tried to detect their refined objectual meaning 41 A (**A**ggregation) relations were found. This reveals, for example, that data stored in the *vendor* table *is part of* the data stored in the *payment* table, the data stored in the *project* table *is part of* the data stored in the *cvorder* table as well as the data stored in the *invoice* table, the data stored in the table *addressrelate is part of* the data stored in the *invoice* table and a *cvfile is part of* the data stored in the *literaturetable*.

The approach also reported 4 I (**I**nheritance) relations among tables. Thus, we expect data stored within tables *projectlink*, *customfieldscalar* and *customfieldmultiple* to be different types of *syncnote*. As we can notice, there are only a few relations of this type and we think this reflects the fact that among the existing *Data Keeper* classes there are only a few inheritance relations.

In *CentraView* the technique detected 7 relations of D (**D**ependency) type. In Figure 8 we present the pairs of tables that are used together in the source code.

```
1. cvfilefolder cvfolder

2. cvfilefolder cvfile

3. emailmessagefolder cvfolder

4. cvfilelink cvfile

5. proposallink proposal

6. emailrecipient emailmessage

7. emailrule account
```

Figure 8. The D relations between tables.

We notice that class *ProposalListForm* is a reported cause for more than one relations from Figure 8, collaborating with various data which belongs to different unrelated tables. Consequently, we expect this class to implement more than one functionality. The investigation we perform reveals that, indeed, the class provides a lot of functionalities, being affected by the *God Class* [11] design flaw. Thus, our approach is useful, beside classifying the relations among tables, in detecting design flaws related to an improper use of the data stored in the database but, as pre-

4 The numbers below are references to the corresponding labels from Figure 7.

viously mentioned, at this moment we consider this aspect beyond the scope of our investigation.

Our approach did also detect 2 relations having the (composed) DAI type. An example of such relation is the one detected between table *user* related by a foreign key to table *individual*. This is a very interesting example of an "overloaded" relation of a foreign key. On one hand, a *user is an individual* (and thus the I (**I**nheritance) relation); on the other hand, a *user* is managing a set of *individuals* and thus it also *contains* the information of those *individuals* (and therefore the A (**A**ggregation relation).

Apart from the aforementioned relations, by applying our technique we also identified 66 foreign key relations for which no D, A or I "traits" were detected, although both tables involved in the relation had at least a *Data Keeper*. We analyzed this case in detail and identified two distinct situations: **Incomplete data usage** *i.e.*, the `CFrom` *Data Keeper* related to table `TFrom` does not make use of the data stored in the referenced table `TTo`. **Common User** *i.e.*, the two *Data Keeper* classes are used together in order to process all data from the two existing tables.

The remaining constraints could not be refined any further because the two involved tables do not have both at least an associated *Data Keeper* in the source code.

Discussion of *CentraView*. Performing a closer manual inspection of the provided relations, we find again several tables that centralize a lot of information in the database, being coupled with many other tables. For example, the most coupled table we found is *individual*, which is involved in 17 relations, 3 of them being A relations and 2 being DAI relations. Detecting these central tables and the specificity of their relations to the other tables is essential for understanding the database schema. As our manual inspection confirmed that the relations that were specified using our approach were correctly identified, we can indeed state that knowing the precise nature of the relations defined as foreign keys helps indeed for the understanding and knowledge extraction from the database of enterprise systems. Concerning the number of false negatives, although we didn't encounter during our manual inspection any such cases, further experiments are needed to assess rigorously the approach from this point of view.

7. Related Work

Finding the meaning of data stored in the relational database tables by using different entities related to the involved database has been done previously. For example, the authors of [25] propose an approach which, beside the database schema, uses existing screen displays, part of the user interface, in order to accomplish the goal of DRE. The approach is applicable for systems written in dBase.

In [5] are presented some idiosyncracies of relational database design. Some of the presented idiosyncracies can not be detected using only the database schema and, consequently, the existing meta-model in our approach may be helpful for a correct identification of some design problems related to the database schema.

In [14] the authors propose the DB-Main approach which mainly answers the question: how a change in the database schema is propagated within the inspected application? As the authors claim, in order to use the DB-Main approach, the database has to be fully documented and some documentation must be built using reverse engineering techniques. We think our approach is helpful in order to provide information regarding the database schema before applying the DB-Main approach.

In [8] the authors describe a tool specifically designed for database reengineering which translates the physical schema of the database into a vendor-independent meta-model. Our approach, like the mentioned contribution, uses a proprietary meta-model, despite the fact that the Object Management Group (OMG) owns the Common Warehouse Metamodel (CWM) [23], 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.

In this paper we assume that persistence is provided by a relational database (e.g., MySQL) and the communication between the entities belonging to the data source layer and the relational database is performed by executing SQL commands as embedded strings. Currently there are also enterprise applications where the persistence layer is manipulated by frameworks like Hibernate [4]. 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 also from existing configuring XML files (and, consequently, we need to develop and use a different model loader).

8. Conclusions. Future Work

This paper is a step forward in enhancing the understanding of persistent data, as part of the maintenance of enterprise applications. The proposed approach provides the objectual meaning of foreign keys, as revealed from the interactions between the source code and database schema *i.e.*, the **D**EPENDENCY, **A**GRREGATION and **I**NHERITANCE relations. The aforementioned relations exist if and only if the involved tables have corresponding Data Keeper classes in the object-oriented source code.

The introduced features require the use of a specific meta-model for enterprise applications [19] which captures in an unitary manner the entities from the object-oriented and the relational paradigms and, most important, the relations between the two. The approach presented in this paper has shown that the strength of such a meta-model that bridges the gap between the object-oriented and the relational "worlds" facilitates a simple expression of analyses that uses information from both sides.

We have conducted an experiment in which we automatically detected the objectual semantics of foreign keys in enterprise applications. The experiment reveals that by using the approach presented in this paper a significant number of foreign key constraints can be semantically refined; thus, the approach contributes to a better understanding of the database schema.

We will focus our future work on the next fronts: (i) we intend to extend the tool support in order to be able to use it on enterprise applications written using other technologies (.NET, different persistence providers, different communication techniques); (ii) we intend to continue the evaluation of the introduced approach against other enterprise applications; (iii) we aim to define some assessment analyses which provide us only with those object-oriented design entities affected by design flaws which are connected to an improper use of the data from the database.

Acknowledgments. This work is supported by the Romanian Ministry of Education and Research under Projects CNCSIS TD(GR76/23.05.2007) and CEEX(3147/11.10.2005). I would like very much to thank Marius Minea and Radu Marinescu for reviewing this paper. Last but not least I would like to thank the LOOSE Research Group (LRG) for being such a great team.

References

[1] S. Ambler. *Agile Database Techniques.* John Wiley & Sons, 2003.

[2] S.W. Ambler and P.J. Sadalage. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley, 2006.

[3] 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.

[4] C. Bauer and G. King. *Java Persistance with Hibernate.* Manning Publications, 2007.

[5] M. Blaha and W. Premerlani. Observed idiosyncracies of relational database designs. In *Proc. Working Conference on Reverse Engineering*, 1995.

[6] E. Chikofsky. *The Necessity of Data Reverse Engineering - Preface to Data Reverse Engineering: Slaying the Legacy Dragon.* McGraw-Hill, 1996.

[7] A. Cleve, J. Henrard, and J.L. Hainaut. Data reverse engineering using system dependency graphs. In *Proc. Working Conference on Reverse Engineering*, 2006.

[8] I. de Guzman, M. Polo, and M. Piattini. An integrated environment for reengineering. In *Proc. IEEE International Conference on Software Maintenance*, 2005.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2003.

[10] K. Brown et al. *Enterprise Java Programming with IBM Websphere.* Addison-Wesley, 2001.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[12] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[13] M. Genero, M. Piattini, M. Esperanza Manso, and F. Garcia. Early metrics for object oriented information systems. In *Proc. International Conference on Object-Oriented Information Systems*, 2000.

[14] J.M. Hick and J.L. Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *Proc. Conceptual Modeling - ER*, 2003.

[15] W. Keller. Mapping objects to tables: A pattern language. In *Proc. European Conference on Pattern Languages of Programs*, 1997.

[16] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice.* Springer Verlag, 2006.

[17] C. Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proc. IEEE International Conference on Program Comprehension (ICPC), Athens, Greece*. IEEE Computer Society Press, 2006.

[18] C. Marinescu. Identification of relational discrepancies between database schemas and source-code in enterprise applications. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), submitted*. IEEE Computer Society Press, 2007.

[19] 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.

[20] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume), Budapest, Hungary*. IEEE Computer Society Press., 2005.

[21] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* John Wiley & Sons, 2002.

[22] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications.* Addison-Wesley, 2003.

[23] 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.

[24] D. Raţiu. *Memoria: A Unified Meta-Model for Java and C++.* Master Thesis, Politehnica University of Timişoara, 2004.

[25] D. Yeh and Y. Li. Extracting entity relationship diagram from a table-based legacy database. In *Proc. European Conference on Software Maintenance and Reengineering*, 2005.