Towards Data Dependency Detection in Web Information Systems

Jörg P. Wadsack*[1], Jörg Niere*[1], Holger Giese[1], Jens H. Jahnke[2]

[1] Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100
D-33098 Paderborn, Germany
{maroc, nierej, hg}@upb.de

[2] Department of Computer Science
University of Victoria
PO Box 3055
Victoria B.C.
Canada V8W3P6
jens@cs.uvic.ca

Abstract. Over the last decade globalisation has taken place and in parallel, the World Wide Web has broadly been adopted for electronic data interchange within and among corporate organizations. Distributed information management is therefore of critical importance in modern societies. A sound understanding of the inter-dependencies among the integrated Web information systems is a key prerequisite for the ability to efficiently evolve these systems step-by-step with rapidly changing requirements. Unfortunately, distributed data dependencies are rarely well-documented and existing database reverse engineering tools do little to support the recovery of these kinds of dependencies. In this paper we classify distributed data dependencies and propose an approach to extract them from existing networked systems.

1 Introduction

Electronic data interchange and Web-centric applications have experienced a rapid growth during the late 90s. This leads to the emergence of distributed, internet-based information systems as also referred to as *Web information systems*. Those systems integrate various heterogeneous information systems to enable combined effective processes and information collection for publishing. Often this integration has evolved in an ad-hoc manner and dependencies between the locally autonomous information systems have rarely been planned and documented in a systematic way. Today, industry faces the challenge of maintaining and adapting these systems with-

* This work is part of the Finite project funded by the German Research Foundation (DFG), projectno. SCHA 745/21.

out complete documentation of all dependencies involved although many of the systems have become indispensable. Consequently, recovery of dependencies between distributed databases in Web information systems becomes increasingly important.

Databases used in Web information systems built from numerous previously independent systems are naturally distributed. To avoid misunderstandings, we use C. J. Date's "working definition" [Dat00] for the notion of distributed database systems.

> A distributed database system consists of a collection of **sites**, connected together via some kind of communications network, in which
>
> a. each site is a full database system site in its own right, but
>
> b. the sites have agreed to work together so that a user at any site can access data anywhere in the network exactly as if the data were all stored at the user's own site.
>
> It follows that the so-called "distributed database" is really a kind of *virtual* database, whose component parts are physically stored in a number of distinct "real" databases at a number of distinct sites (in effect, it is the logical union of these real databases).

Programmers working on maintenance of Web information systems have to face the permanent demand for evolution due to changing requirements. Besides extending the functionality of the system, technological improvements and the evolution of requirements lead to frequent changes. The integration and merging of multiple (Web) information systems to Web information systems are other typical problem scenarios. In contrast to the non-distributed case, no overall structural representations are available and, thus, system understanding is rather difficult. To support redesign and evolution of Web information systems we propose to use the information present in the application to identify the relevant data dependencies. In addition to the reengineering of each single database schema we require the analysis of code fragments coordinating the access to multiple databases.

Hence, the integration of databases takes place within applications and not in a common database management system. In Section 2 we motivate and exemplify the activity of data dependency recovery by means of a case study. Section 3 describes different types of inter-schema dependencies and how they typically occur in application code. Further we present the conceptual representation and our combined process of reengineering data dependencies using the analysis of single databases and application code. The paper closes with related work and conclusions.

2 Case Study

The German ministry of education and research (BMBF) has currently started the UNI-MOBILIS project at several German universities. The aim of this project is to provide students with wireless access to all university-related services. Those planned services comprise management of personal data, study schedule planning, course and

exam registration, library account handling and multimedia course materials. Such services will first be established as pilot projects at a small number of universities in Germany, before a country-wide solution is planned in a second step. Common to all sites is the existence of distributed information systems that have to be understood before installing the student-services.

At Paderborn University currently exists a wireless infrastructure with access to the internet, which has been built up for the last two years and covers most buildings and lecture halls. Installing such an infrastructure is a minor, because it could be newly built, problem compared with the big challenge of integrating the existing information systems of the university services grown over the last decades.

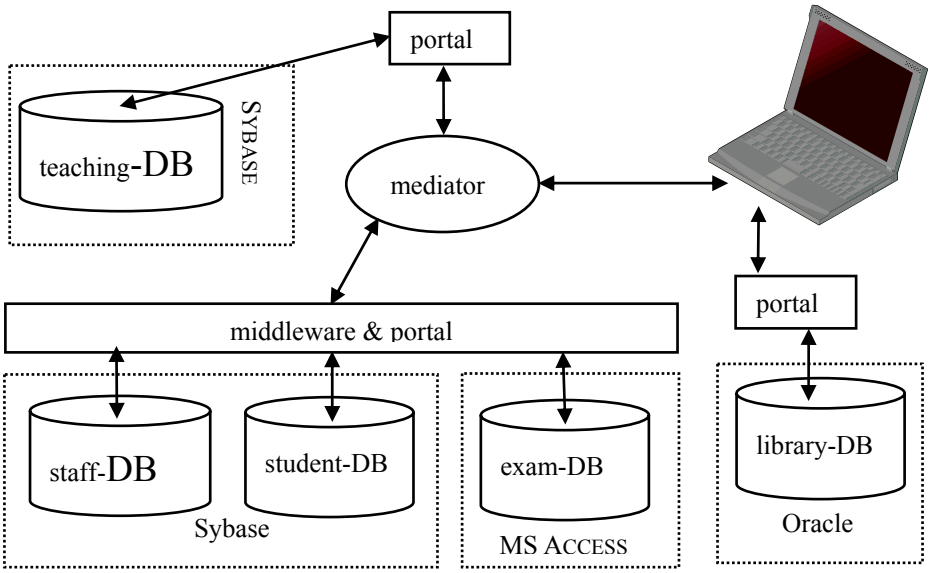


Figure 1: Topology of the Uni-Mobilis project

The different information systems at Paderborn University are quite heterogeneous and are distributed over the different involved departments or institutions. This ranges from an integrated distributed information system based on a common business platform up to paper portfolios, one for each student. Data transfer between different services is often made by passing unformatted text documents or semi-formatted sheets manually and electronically.

Figure 1 shows a possible solution topology built upon the actual information system architecture of each university service. Services strongly related to each other are accessible over a common middleware. A mediator controls access to the different portals of the university's information services. Such an approach allows us to integrate the services incrementally and not all at once. To build up such a topology we need to understand the existing dependencies between the services.

3 Types of Data Dependencies

Web information systems usually employ a *three tier architecture* [Fow96], which separates the data, application and graphical user interface (GUI) tier (see Figure 2). The data tier manages persistency using multiple databases and accesses the databases using SQL. The application tier includes the functionality, the core business logic and required coordination between the different databases. The GUI tier or front end is typically realized using available Web browser or network-centric languages, e.g. Java.

Web information system *maintenance* has to face the permanent demand for *evolution*. Besides extending the functionality of the system, technological improvements and the evolution of business rules lead to frequent changes. The three-tier architecture especially takes the requirement for evolution and changeability into account: each tier covers a different concern that might evolve separately, namely data requirements (data tier), business logic (application tier), and representation (GUI tier). Despite this separation of concerns, changes to one tier often imply changes in other tiers.

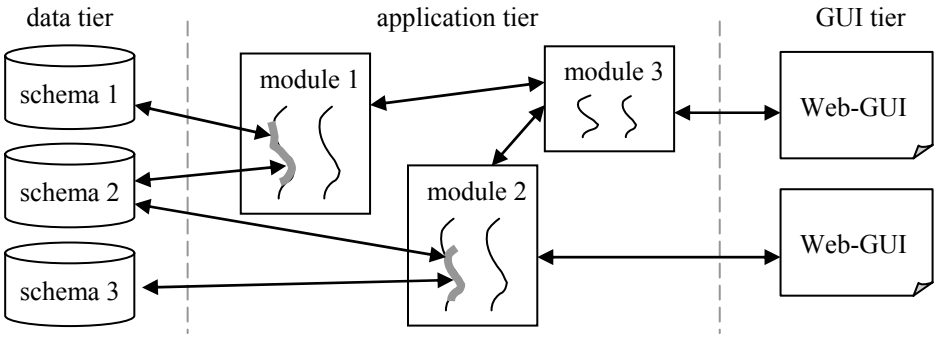


Figure 2: Code fragments (grey short bars) and schema access

The *integration* and *merging* of multiple (Web) information systems to Web information systems is another frequent maintenance scenario. Typically the flexibility of the application tier (using several proprietary middleware) is used to realize the additionally required coordination between the various applications and databases. While a "deep" integration of the database schemas would permit to handle the resulting dependencies in a declarative manner, application mediation via transactional middleware in practice better facilitates the federation of legacy databases, which have to retain certain autonomy.

Frequent changes and integration are therefore crucial maintenance scenarios Web information systems have to face. In contrast to the non distributed case no overall structural representations are available and thus *system understanding* is rather difficult. A set of database entries which is spread over the different physical and logical

databases might depend on each other in various different ways and the intended coupling can have different semantic properties. The relevant data dependencies, however, will all manifest themselves in the application tier. The corresponding code fragments are generally delimited by transactions boundaries. This is depicted with the grey short bars in Figure 2. In this paper, we consider only code fragments which are encapsulated in transactions but we do not look across transaction boundaries. We have chosen this limitation to keep the complexity of fragment detection manageable.

We characterize a set of basic (inter-) database dependencies relevant for evolution and integration of Web information systems. We illustrate each data dependency type with a sample code fragment, which has to be analysed to reveal a dependency between distributed databases.

We will also discuss how to choose the most relevant code fragment from a set of related ones. An often occurring case is that data dependencies in form of stored foreign keys or attributes have been used. Another case is complex functional dependencies, which are used when the relevant dependencies are computed on-the-fly by combining the values of multiple stored attributes. While revealing the first case is relatively simple, instances of the latter case are rather hard to detect. Thus, identifying data dependencies can become quite complex.

Next, we will give an informal description of data dependency types. The data dependency types are described by corresponding pairs of attributes. Relations (joins) between them have to be considered. In this paper, we classify three basic kinds of dependencies. We call them "inter-schema dependencies" which we refine in:

- redundancy dependency:
  the same information is held - and maintained - (at least) twice

- inclusion dependency:
  an (a set of) attribute in one database table holds a part or the same information as an (a set of) attribute of a second database table

- constraint dependency:
  condition(s) over two or more data dependencies to assign information

Fahrner and Vossen [FV95] proposed similar classifications for inclusion dependencies in *single* database schemas. In general, the reverse engineering results are presented in a conceptual model to the reengineer. We choose UML [UML], i.e. classes and associations to represent the revealed data dependencies.

The dependency types are mostly based on attribute indicators, i.e., relations between attributes properties. Attribute properties are name similarity (ns) or name equivalence (ne), and type compatibility (tc) or type equivalence (te). Building the cross product, we get four different similarity properties for attributes, i.e., attribute

similarity (ns&tc), attribute type equivalence (ns&te), attribute name equivalence (ne&tc) and attribute equivalence (ne&te).

3.1 Redundancy Dependency

The first inter-schema dependency type is called *redundancy dependency*. Redundancy dependencies might be implemented using *synonyms* or three other alternatives based on attribute equivalence (ne&te).

Synonyms are attributes that hold the same information but have different names. Therefore, synonyms imply data dependencies, where joins (at least one insert and one update), but no attribute similarity or equivalence dependency, relate the dependent data. In some cases data is never updated because the information system uses keys that are never altered (e.g. a student id). For this case further more complex investigation has to take place which is based on attribute semantic resemblance. Semantic resemblance and in consequence semantic equivalence (synonym) is hard to determine and subject of our current work.

```
1: ... // Adding the same value in two attributes
2: ... "UPDATE student_DB.person SET name VALUES (newName)"
3: ...
4: ... "UPDATE exam_DB.student SET name VALUES (newName)"
5: ...
```

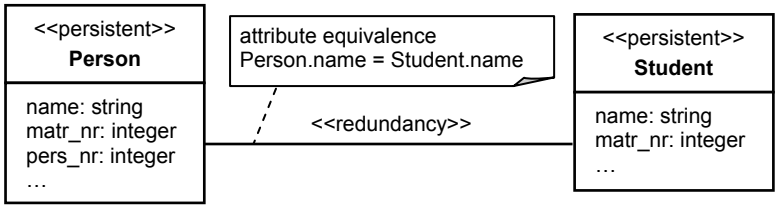


Figure 3: Redundancy Dependency

We classify the other three redundancy dependencies as *redundancy*[2], *duplication* and *replication*. In addition to attribute equivalence, redundancy is revealed when at least an update-join but no insert-join relates the data. We have "real" redundancy if the same information is maintained but not inserted together. Otherwise we talk about *duplication* if an explicit copy is made at specific points in the application but the copied information is not kept consistent. Thus, we define it as attribute equivalence with at least an insert-join but no update-join. Finally, *replication* is an explicit copy, which is held consistent, i.e., a controlled redundancy. Thus it is the occurrence of attribute equivalence, and at least one insert-join and one update-join.

[2] Note that redundancy is used twice in this paper; first as an inter-schema dependency type and second as an instance of the redundancy dependency type.

In Figure 3 we show an occurrence of a redundancy dependency in the middleware, namely redundancy. Lines 2-4[3] show that the attributes *name* in *Person* and *name* in *Student* are updated with the same value *newName*. These two attributes maintain the same information which indicates the occurrence of redundancy. In this paper, we consider only code fragments which are encapsulated in transactions but we do not look across transaction boundaries. We have chosen this limitation to keep the complexity of fragment detection manageable.

In a conceptual (reengineered) UML representation [UML] of the database schemas we will have a one-to-one association with stereotype <<redundancy>> between Person and Student. We keep both attributes in the classes because we cannot automatically decide how this redundancy can be resolved. Thus, the attributes are still represented twice until the reengineer resolves the redundancy. In addition, we append a note to the association with a comment to store details about the redundancy.

For the other redundancy dependencies we use different stereotypes. We represent synonyms, i.e. a consistently maintained copy of attributes with no similarity dependency relating them, as an association with stereotype <<synonyms>>. A duplication, the identification of an insert-join but no consistency maintenance (no update-join), is represented as an association with stereotype <<copy>>. In the case that an insert-join as well as an update-join occurs, i.e. we have a replication; we represent it as an association with stereotype <<replica>>.

3.2 Inclusion Dependency

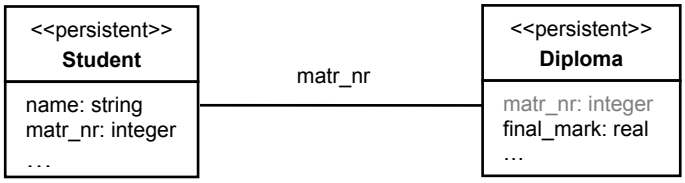
Inclusion dependencies are known from (single) relational databases [FV95]. An inclusion dependency is a data dependency where the types of the attributes are compatible and at least one select-join exists. They also form the basis for interpreting the semantics of foreign keys. Each foreign key implies an inclusion dependency where the included attribute (set of attributes) is a key of the corresponding data (table). Here we use the classical definition of the inclusion dependency in data reengineering.

We will show three examples of foreign keys. We start with a dependency between two database tables in one database. Then, we show the difference to inter-schema dependencies. The upper part of Figure 4 shows an occurrence of a select-join in the middleware. Along with the information that the student id *matr_nr* in Student is a primary key, we can identify *matr_nr* in *Diploma* as a foreign key. In a conceptual view this will be represented as an association with the name of the foreign key (*matr_nr* in this case). The attribute *matr_nr* in Diploma is in grey colour because in a conceptual representation it is not necessary (we list it for better understanding). This "foreign key attribute" is accessible through the association. Note that cardinal-

[3] We use pseudo SQL queries in quotes for the database access for readability reason, instead of real ‘executeQuery’ statements of a certain API.

ities have to be recovered through further investigations, which we omit in this paper due to the lack of space.

```
6: ...
7: ... // in exam_DB in MS Access
8: ... mark = "SELECT d.final_mark FROM student s, diploma d
               WHERE s.matr_nr = d.matr_nr"
9: ...
```




```
10: ...
11: ... // between student_DB and staff_DB in Sybase
12: ... amount = "SELECT s.amount FROM student_DB.person p,
                  staff_DB.salary s WHERE p.pers_nr = s.pers_nr
13: ...
```

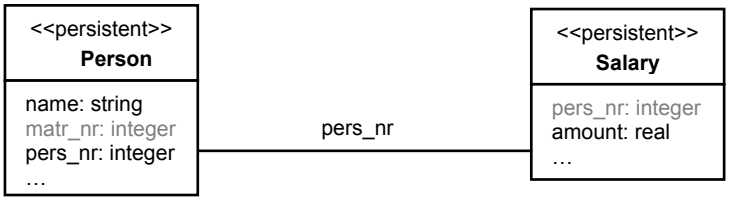


Figure 4: Inclusion Dependencies (hidden in SQL statements)

The next example of a foreign key dependency is a join of two tables in two separate databases, but handled by one DBMS. Line 12 is nearly identical to line 8 in Figure 4. Again this code fragment lies in the middleware and we have the information that *pers_nr* in Person is a primary key. The difference to the preceding example here is that the databases are listed in front of the dependencies. Note that in DBMS, where the tables are uniquely identifiable, there is no need to list the dependencies explicitly. The attribute *matr_nr* is in grey shape because it belongs to another association, cf. Figure 5.

The last example of a foreign key dependency is a dependency between databases in separate DBMS. Consequently, the join will be coded in the application's programming language, e.g. Java, and not in the database query language (SQL). Figure 5 shows such an occurrence. In line 16 and 17 sets are assigned the return values from the SQL queries. We already know that *matr_nr* in Student is a primary key. The actual join is encoded in the nested while-loops (line 19 and 22).

```
14: ... // between exam_DB in MS Access and
15: ... // student_DB in Sybase
16: ... students = "SELECT matr_nr FROM student_DB.person";
17: ... examinees = "SELECT matr_nr FROM exam_DB.student";
18: ... studIter = students.iterator();
19: ... while (studIter.hasNext() )
20: ...  { student = studIter.next();
21: ...    examIter = examinees.iterator();
22: ...    while (examIter.hasNext() )
23: ...      { examinee = examIter.next();
24: ...        if ( student.equals(examinee) )
25: ...          { … }
26: ...      } …
27: ...  } …
28: ...
```

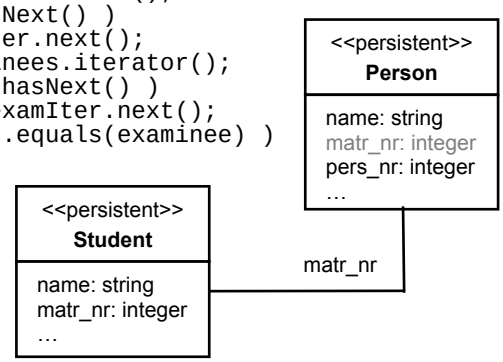


Figure 5: Inclusion Dependency (hidden in application code)

3.3 Constraint Dependency

All remaining dependencies are classified as *constraint dependencies*. A sub-classification of the constraint dependencies is our current work. We illustrate the constraint dependencies by the following example.

Figure 6 shows an occurrence of a constraint dependency. Again, we choose an example of a dependency between database schemas of separate DBMS. Therefore, the database constraint manifests itself not in a database constraint but in the application code. We start with only one assignment to a set and one while-loop (lines 30, 32-41). Furthermore, there exists the foreign key/association *pers_nr* between Person and Salary (line 34). Moreover, a concatenation of the return value from a selection query (line 35) and the foreign key/association *matr_nr* between Student and Diploma is used (line 38).In our UML representation, we encapsulate this constraint in an own class with stereotype <<constraint>>. The relation between the <<constraint>> class and the classes which attributes are used in the constraint is represented by associations. Finally, class <<constraint>> Check is annotated with the code (lines 29-41) as a comment for the reengineer.

The notion of a constraint can be used in further analysis and re-design steps to identify the essential business rules and the business logic in the system that effect multiple databases.

```
29: ... // constraint for checking
30: ... persons = "SELECT pers_nr FROM student_DB.person";
31: ... personsIter = persons.iterator();
32: ... while ( personsIter.hasNext() )
33: ...  { person = personsIter.next();
34: ...    salary = getAmount(person);
35: ...    stud_id = "SELECT matr_nr
36: ...               FROM student_DB.person p
37: ...               WHERE p.pers_nr = person";"
38: ...     mark = getMark(stud_id);
39: ...     if ( salary > 0.0 && !(mark==0.0) )
40: ...       { exception.warning(person); }
41: ...   } …
42: ...
```

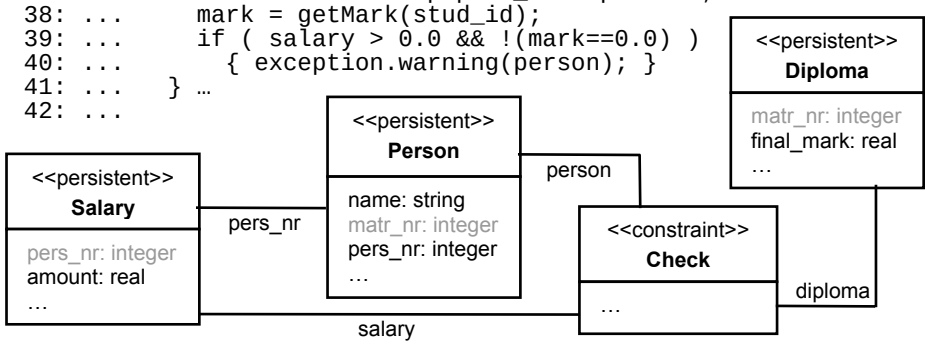


Figure 6: Constraint Dependency

3.4 Resulting Enriched Conceptual UML Representation

A detailed of the extracted schemas of the Web information system example of Figure 1 is presented in Figure 7. We notice that almost all inter-schema dependencies in our example could not be inferred from the data base schema but we needed to analyse the application code (fragments) in order to detect them (except for the association *matr_nr* from Student to Diploma). Figure 7 shows different kinds of inter-schema dependencies without, even in this simple example, an understanding of the distributed database systems is realistic.

In addition to the data dependencies it is useful for the reengineer to recover relationships between the persistent and transient parts of the Web information system, i.e., relations between the data-tier and the application-tier. We classify this kind of relationship as *usage* relationship.

- usage relationship
  is the usage of data by transient parts of the Web information system, i.e., an interface using the data dependencies for the connection to the "transient world" (the Web).

Reverse engineering usage relationships are a first step towards overall Web information system analysis. Such a relationship with stereotype <<usage>> is shown

in Figure 7 in the servlet *Account* which connects the GUI with the database *library_DB*.

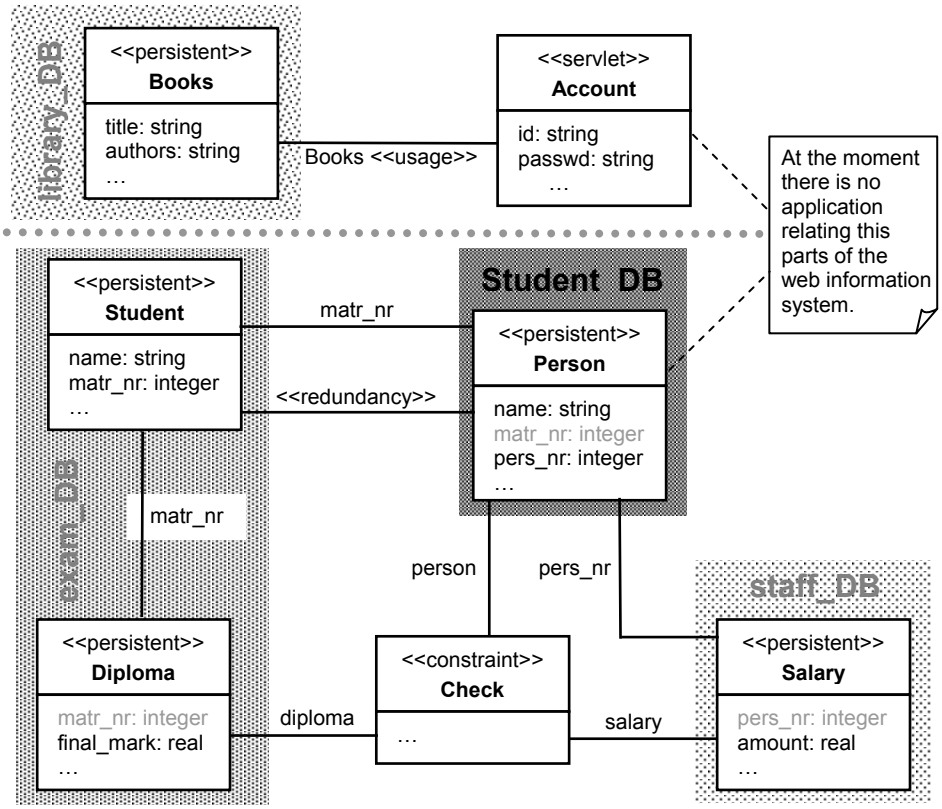


Figure 7: Cut-out of the resulting enriched conceptual schemas

The understanding can be further improved with a distinction of the (inner database versus inter-schema) inclusion dependencies. A first simple way would be additional stereotypes for associations, e.g. <<distributed>> for inter-schema inclusion dependencies.

3.5 Inter-Schema Dependency Recovery

The problem of recovering inter-schema dependencies among distributed Web databases is quite complex in general. The reason for this complexity is the great heterogeneity present in such system and the various mechanisms used to integrate them. Therefore, we suggest the separation of the recovery problem into three smaller problems, namely (1) extracting code fragments of interest from the Web information system and (2) the reverse engineered single schema information and (3) the inference of knowledge about inter-schema dependencies based on the extracted code

fragments and the single schemas. The resulting process is sketched in Figure 8. The refinement of the reengineering process is current work consequently we provide only a short description of it as it stands so far.

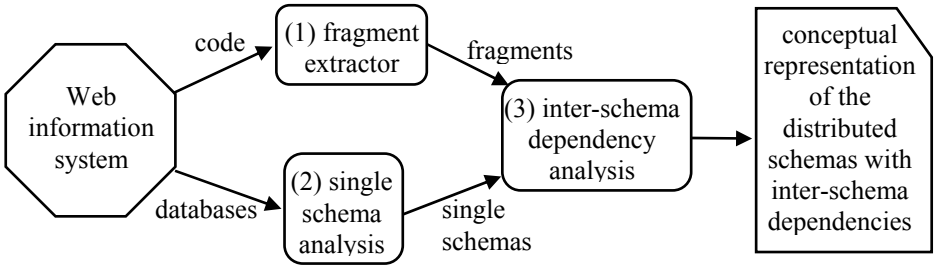


Figure 8: Reengineering Process

To automate the extraction of code fragments of interest, we make use of parser technology to import the legacy code into some internal representation that can be further processed. This is not new and has been done for decades in numerous reverse engineering tools. Today, parsers, or parser generating grammars, are available for a wide variety of languages, including database languages like SQL and programming languages like COBOL, C, and Java. These parsers have been used to build extractors for reverse engineering homogeneous systems, i.e., systems implemented in a single language.

In case of heterogeneous systems, however, the reverse engineering problem is more difficult because it deals with multiple different programming languages. Using multiple different parsers solves the problem only partially, because this approach fails to capture the inter-relationships between software artefacts written in different languages. This problem gets worse for multi-language systems where certain languages are embedded in other languages. This situation is typical for many information systems: most database management systems provide proprietary data manipulation languages embedded in various host languages like C, COBOL, Java, etc. In addition, the code fragments may contain code pieces from multiple modules integrated in the application tier via interoperable interfaces [COR99, Cha96, Ib96].

Parsers for extracting inter-schema dependencies among distributed databases have to deal with code fragment, that are amalgamations of different languages including proprietary dialects. This feature renders the reuse of existing parsers highly unlikely. In addition, our experiences show that the construction of multi-lingual custom parsers might become a fairly complex task. The reduction of reverse engineering effort achieved by the resulting extractor may be lost when building and adapting a multi-lingual custom parser. However, the code fragments of interest are not arbitrary amalgamations of different languages but rather the well separated code fragments executed within distributed transactions. Therefore, we can simplify the task by looking at its specific characteristics.

As shown in Figure 5, the inter-schema dependencies are included in the application code where the database access is done by using API's provided by the database itself. For example, in Java the *Java DataBase Connectivity interface* (JDBC) is the standard interface to access various kinds of relational databases. The interface provides data structures which can be accessed and modified in the Java programming language. Hence a complete analysis of an application is too expensive regarding the analysis time, we use the JDBC interface declarations as starting points to extract fragments of interest only.

In general, distributed transactions are used to ensure data integrity for the access and manipulation of multiple databases. The code used within such a distributed transaction may be spread over a set of procedures. In practice besides local procedure calls also remote procedure calls are used. Within the application tier the current transaction context is propagated either in an implicit [COS98, OTS98] or explicit [AOS+99] manner. The resulting transaction boundaries can determine the relevant excerpt of code fragments for the analysis of data dependencies. This reduces the lines of code dramatically and allows us to handle also large applications.

Still, it is important to note that the code fragments of interest for extracting relationships among distributed databases are typically in a fairly small subset of the multi-language grammar. Therefore, it is viable to construct more simple parsers that filter out only those "interesting" parts of the multi-language syntax and ignore everything else. A naïve way of performing this filtering is using a pre-processing step with a lexical analyser like the Unix *grep* command. However, this simple approach has severe limitations as outlined in [Moo01].

Therefore, reverse engineering researchers have started to investigate more powerful approaches, one of them being *Island Grammars*. An Island Grammar can informally be defined as *a set of production rules that describe the language fragments of interest (so-called islands) plus another set of production rules that catch the rest (so-called water)*. Obviously, the idea behind the concept of island grammars is to make the water significantly less descriptive than the islands in order to decrease the complexity of the associated parser. For a formal definition of island grammars, we refer to [Moo01].

Recently, Moonen has presented Mangrove [Moo01], a parser generator that takes an island grammar in SDF [HHKR89] format and produces an extractor for this (partial) language. Moonen demonstrated the usefulness of this approach with several case studies. Still, the definition of an island grammar for our application, the extraction of relationships among distributed, heterogeneous databases is a task that requires an intimate understanding of the concept of island grammars and a highly explorative process.

To simplify this process, we have developed a tool for interactively creating island grammars based on code examples of interest identified by the user. This tool, called *Buffy*, initially assumes that the entire input code represents *water*. When the user

identifies instances of interesting code fragments, *Buffy* suggests a set of *island* productions for this instance. Subsequently, the user can interactively correct and refine these productions to characterize the associated *island*. Then, the user can generate a prototype extractor and run it against other parts of the input code in order to verify if this *island* recognizes other instances of this pattern. Depending on the result of this verification step the user might iteratively refine the description of the *island*. The extractors that *Buffy* generates are then used to add mark-up information in terms of XML tags to the input code. The mark-up information transforms the source code into semi-structured data with unstructured text (tagged as water) and structured text (islands). This marked-up version of the source code can then easily be parsed into a DOM tree using a standard XML parser to perform the actual detection of the inter-schema relationships implied.

Resulting, the fragment extractor yields islands (code fragments) delimited by transaction boundaries including the indicators for the inter-schema dependencies encoded in XML.

Analysis approaches of single database systems are based on schemas, access code, and the data itself. This field is well-explored and understood and do not underlay our main focus. In principle, our process depends only on the reverse engineered single schemas and not on the methodology. We [Jah99, JSWZ02] use a so-called cliché library which categorize typical database queries and cover nearly all variants. Unfortunately, during database analysis inconsistencies occur where an appropriate analysis approach has to deal with. Uncertainty has also to be covered in order to get a correct abstract representation of the database. We cover uncertainty with Generic Fuzzy Reasoning Nets [JSZ97, Jah99] and a semi-automatic approach. The outcomes of this analysis are the single schemas with their inner-schema dependencies represented as UML.

Inferring knowledge about inter-schema dependencies is based on the extracted code fragments and the reverse engineered schema information about each single integrated information system. Data structures provided by a certain API, e.g. JDBC, and modifications on the structure are usually implemented in the same way. Note, that this includes encoding in SQL and Java. Usually, different developer teams implement several modules in the application tier, where particularly one developer-team is responsible for a subset of the inter-schema dependencies. Each developer in a team uses the same language constructs based on programming style guides or code reuse. Those specific language constructs can be used to define a catalogue of patterns like design pattern to analyse the application code fragments and combined with the single conceptual schemas we are able to recover inter-schema dependencies. The outcome of this task is a conceptual representation of the distributed schemas. A detailed description can be found in [NSW+02]. The described approach presents the recovery of design patterns, whereas the patterns are stored in a catalogue.

4 Related Work

To our best knowledge, a catalogue for inter-database/-schema dependencies has not been published yet. Rusinkiewisz et al. [RSK91] present two examples of inter-database dependencies. First, replicated data characterized as "identical copies of data in two or more databases" for which "we can tolerate inconsistencies (...) for no more than one day" by the authors. Second, existential constraints which are e.g. referential integrity constraints requiring immediate updates. Both correspond to the redundancy dependency with respect to copies that have to be held consistent. Our approach also discovers possible redundant data by duplicated schema elements.

A theory of attribute equivalence in databases on a semantic basis is presented in [LNE89]. The approach uses semantic attribute equivalence for integration of database schemas. Therefore the characteristics of the attribute equivalence are very detailed and restrictive. In contrast to schema integration, schema reverse engineering needs flexible and general attribute property (characteristic) definitions.

Identifying and solving conflicts in inter-schema knowledge in cooperative information systems has been presented in various references, e.g. [BLN86, TGF00, CL93]. In these approaches the discovery and representation of inter-schema assertions is studied to "make explicit the knowledge which a human integrator uses implicitly to identify semantic similar schema concepts" [TGF00]. This is different from the inter-schema knowledge, i.e. explicit dependencies between the distributed databases, we want to recover.

Schema matching provides a mapping between two schemas that semantically correspond to each other. A detailed overview over existing partially automated schema matching techniques is given in [RB01]. Schema matching techniques are needed for data integration and can be used to detect schema overlapping. To obtain a complete picture of a Web information system additionally inter-schema dependencies are required.

Like Moonen, we suggest the use of partial parsers generated by Island Grammars for extracting particular code fragments of interest from legacy code [Moo01]. In addition, we point out that the development of these Island Grammars for multi-language systems might still be a complex task. We suggest supporting this task by an interactive environment (Buffy). Buffy lets the user develop such an extractor grammar driven by input examples.

An approach and a tool for database (single schema) reengineering activities is DB-Main [EH99]. In this approach the reverse engineering process is invoked by predefined scripts which look at the application code [HHH+99] to extract data structures. The DB-Main tool [THB+98] integrates the construction of abstract interfaces to access independent heterogeneous distributed databases. This approach is limited to inter-schema dependency recovering, due to the low flexibility of the predefined scripts.

Concerning pattern detection, Keller et. al. present an approach [KSRP99] to recover design patterns. Patterns are defined using UML and a pattern matching algorithm matches patterns on an abstract syntax graph representation of the source code, also using the UML notation. The matching process is executed using scripts and adoption of patterns is hard to follow especially when patterns are highly interrelated. In addition Seemann and von Gudenberg [SvG98] present an approach to recover design patterns starting with inheritance relations, call graphs, naming conventions, and programming guide lines. The pattern definition of higher order patterns allows a reverse engineer to compose patterns out of subpatterns and reduces thereby the number of definitions. Both approaches are also feasible for the pattern based analysis task, but cannot deal with large programs.

5 Conclusions and Future Work

Web information systems gain more and more importance and have become the fastest growing information system area. To protect the investments in large Web information systems their maintainability and the ability to adapt them to changing requirements is of crucial importance. Today, a great variety of different technologies such as XML, Java, script languages and databases are used to realize the required overall Web information structures. While most of the current used techniques facilitate the fast development of new Web information systems (e.g., script languages), their support for evolution is rather limited. Even when the Web front ends are replaced by new ones, build using new technologies, a proper understanding of the former system and its inherent data dependencies is helpful. Therefore, reengineering techniques for these systems are required, which facilitates their *understanding* and adoption to new requirements.

As underlying foundation for such reengineering techniques, we defined several types of data dependencies which can be found frequently in distributed database systems. We distinguish between three data (inter-schema) dependency types. Redundancy dependency which is typically for distributed databases. Inclusion dependency which is well known from the relational database field. And finally constraint dependency which represents complex relations between data. We have illustrated these inter-schema dependencies with examples from our case study.

Moreover, we sketch the different steps, which are required to recover those inter-schema dependencies in a distributed database systems. Based on the extraction of code fragments of interest and the reengineering of the single "real" database schemas, we use pattern based analysis process to recover the dependencies between those schemas.

Thus, by revealing the relevant information about the persistent parts of Web information systems, i.e., the single schemas and inter-schema dependencies the presented reengineering techniques support maintaining and extending a Web information system. Based on this information, further investigations can now reveal other

dependencies inherent in the Web information system and improve the maintenance of the overall system as, e.g., in the case of the usage relationship which connects parts of the data tier with parts of the application tier.

The flexibility of our approach further permits to reuse existing reengineering technologies such as extract code fragments of interest and reverse engineering of single schema information. Therefore interoperability of reengineering tools, like discussed last year in Dagstuhl [EKM01], would facilitate integration of improved tool versions for those tasks.

References

[AOS+99] K. Arnold, B. Osullivan, R.W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan. The Jini(TM) Specification. Addison-Wesley, June 1999.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(2):323–364, ACM Press, 1986.

[Cha96] D. Chappell. Understanding ActiveX and OLE - A Guide for Developers and Managers. Microsoft Press, 1996.

[CL93] T. Catarci and M. Lenzerini. Representing and Using Interschema Knowledge in a CooperativeInformation Systems. International Journal of Intelligent and Cooperative Information Systems, 2(4):375–398, IEEE Computer Society Press, 1993.

[COR99] CORBA-2.3.1. The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 2.3.1 Specification. Object Managment Group, October 1999. Revision 2.3.1: OMG Technical Document formal/99-10-07.

[COS98] COSS-2.3. CORBAservices: Common Object Services Specification. Object Managment Group, December 1998. Revision 2.3, OMG technical document 98-12-09.

[Dat00] C.J. Date. An introduction to database systems. Addison-Wesley, 7th edition, 2000.

[EH99] V. Englebert and J.-L. Hainaut. DB-MAIN: A Next Generation Meta-CASE. Journal of Information Systems - Special Issue on Meta-CASEs, 24(2):99–112, Elsevier Science Publishers B.V (North-Holland), 1999.

[EKM01] J. Ebert, K. Kontogiannis, and J. Mylopoulos, editors. Interoperability of Reengineering Tools, volume 296 of Dagstuhl-Seminar-Report. IBFI gem. GmbH, January 2001.

[Fow96] M. Fowler. Accountability and Organizational Structures, Patterns for analysis and design. In Pattern Languages of Program Design, pages 353–370, 1996.

[FV95] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. In Proc. of the 4th Intl. Conference on Deductive and Object-Oriented Databases, 1995.

[HHH+99] J. Henrard, J-L. Hainaut, J-M. Hick, D. Roland, and J. Englebert. Data structure extraction in database reverse engineering. In Proc. of the 1st International Workshop on Reverse Engineering in Information Systems (REIS'99), Paris, France, November 1999.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – Reference manual. SIGPLAN Notices, 24(11):43–75, 1989.

[Ib96] CORBA IDL-binding. Information technology – Information Resource Dictionary System (IRDS) Services Interface. ISO/IEC, 1996. Amendment 3:1996 to ISO/IEC 10728:1993 CORBA IDL binding.

[Jah99] J.H. Jahnke. Management of Uncertainty and Inconsistency in Database Reengineering Processes. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JSWZ02] J.H. Jahnke, W. Schäfer, J.P. Wadsack, and A. Zündorf. Supporting Iterations in Exploratory Database Reengineering Processes. Journal of Science of Computer Programming, Elsevier Science Publishers B.V (North-Holland), 2002. (to appear).

[JSZ97] J.H. Jahnke, W. Schäfer, and A. Zündorf. Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications. In Proc. of European Software Engineering Conference (ESEC/FSE), LNCS 1302. Springer Verlag, September 1997.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. In Proc. of the 21th International Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE Computer Society Press, May 1999.

[LNE89] J.A. Larson, S.B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. IEEE Transactions on Software Engineering, 15(4):449–463, IEEE Computer Society Press, April 1989.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In Proc. of the 8th Working Conference on Reverse Engineering (WCRE), Stuttgart, Germany. IEEE Computer Society Press, October 2001.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, USA, May 2002.

[OTS98] OTS-1.1. Transaction Service Specification. Object Management Group, February 1998. The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 1.1 Specification, Revision 1.1: OMG Technical Document formal/97-12-17.

[RB01] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. The VLDB Journal 10(4), pages 334-350. Springer Verlag. December 2001.

[RSK91] M. Rusinkiewisz, A. Sheth, and G. Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. IEEE Computer, 24(12):46–53, IEEE Computer Society Press, December 1991.

[SvG98] J. Seemann and J.W. von Gudenberg. Pattern-Based Design Recovery of Java Software. ACM SIGSOFT Software Engineering Notes, 23(6), ACM Press, November 1998.

[TGF00] A-R.H. Tawil, W.A. Gray, and N.J. Fiddian. Discovering and Representing Inter-Schema Semantic Knowledge in a Cooperative Multi-Information Server Environment. In M.T. Ibrahim, J. Küng, and N. Revell, editors, Proc. of the 11th International Conference on Database and Expert Systems Applications (DEXA'00), London, UK, LNCS 1873, pages 548–562. Springer Verlag, September 2000.

[THB+98] P. Thiran, J-L. Hainaut, S. Bodart, A. Deflorenne, and J-M. Hick. Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE Support: the InterDB Approach. In Proc. of the 3rd IFCIS International Conference on Cooperative Information Systems (Coopis'98), New York, USA. IEEE Computer Society Press, August 1998.

[UML] Rational Software Corporation. UML documentation version 1.3 (1999). Online at http://www.rational.com.