ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2006, Vol.35, No.3

BUSINESS KNOWLEDGE EXTRACTION FROM LEGACY INFORMATION SYSTEMS

Bronius Paradauskas, Aurimas Laurikaitis

Kaunas University of Technology, Department of Information Systems
Studentų St. 50, LT-51368 Kaunas, Lithuania

Abstract. This article discusses the process of enterprise knowledge extraction from relational database and source code of legacy information systems. Problems of legacy systems and main solutions for them are briefly described here. The uses of data reverse engineering and program understanding techniques to automatically infer as much as possible the schema and semantics of a legacy information system is analyzed. Eight step data reverse engineering algorithm for knowledge extraction from legacy systems is provided. A hypothetical example of knowledge extraction from legacy information system is presented.

Keywords: knowledge extraction, relational database, data reverse engineering, legacy information systems.

1. The problems of legacy information systems

Legacy information system is any information system that significantly resists modification and evolution to meet new and constantly changing business requirements 0. Legacy systems typically contain incredible detailed business rules and form the backbone of the information flow of organization that consolidates information about its business 0. A failure in one of these systems may have a serious business impact. Legacy information systems are currently posing numerous and important problems to their host organizations. The most serious of these problems are:

- systems usually run on obsolete hardware which is slow and expensive to maintain;
- maintenance of software is generally expensive; tracing faults is costly and time consuming due to the lack of documentation and a general lack of understanding of the internal workings of the system;
- integration efforts are greatly hampered by the absence of clean interfaces;
- legacy systems are very difficult, if not impossible, to expand.

In response to these problems, several approaches to change or replace legacy systems have been proposed. They are classified into the following three categories 0: redevelopment, wrapping, and migration. Redevelopment involves process of developing system from scratch, using a new hardware platform, architecture, tools and databases. Wrapping involves developing a software component called wrapper that allows an existing software component to be accessed by other components. Migration allows legacy systems to be moved to new environments that allow information systems to be easily maintained and adapted to new business requirements, while retaining functionality and data of the original legacy systems without having to completely redevelop them.

Usually, in the process of replacing legacy systems the above three approaches are combined in varying degrees. First thing needed when solving the problems of legacy systems is the exact picture of information system. So reverse engineering must be accomplished in order to discover and extract as much as possible business knowledge from legacy sources.

2. Knowledge extraction

The goal of knowledge extraction from legacy systems is to semiautomatically discover and extract enterprise knowledge from legacy sources, i.e. generate a detailed description of the legacy source, including entities, relationships, application-specific meanings of the entities and relationships, business rules, data formatting and reporting constraints, etc. We collectively refer to this information as business knowledge.

The schema information is extracted (Figure 1) from legacy system database management system (DBMS). Then this schema information can be semantically enhanced using clues extracted by the semantic analyzer from available application code, business reports and other electronically available information that may encode business data such as e-mail correspondence, corporate memos, etc.

In this article, data reverse engineering algorithm (DRE) which is comprised of schema extraction (SE) and semantic analysis (SA) is provided. An example that illustrates the DRE operation and confirms the power of the approach is also provided.

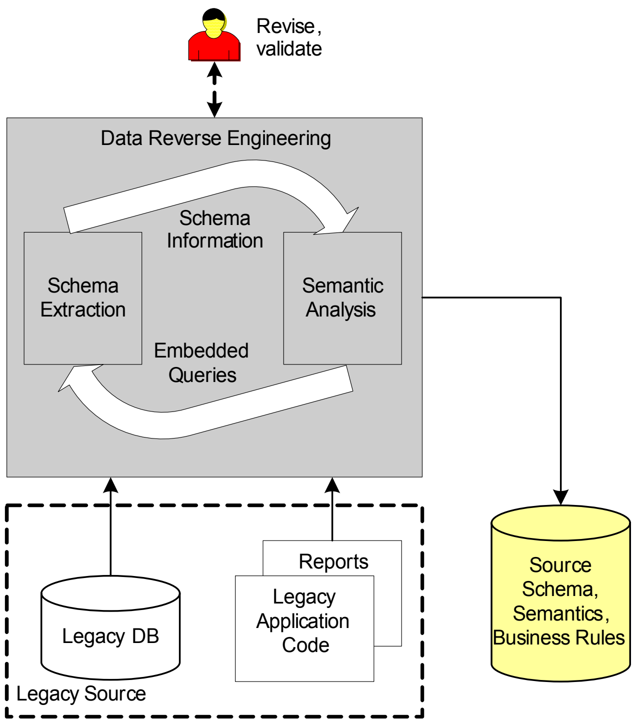

Figure 1. Knowledge extraction conceptual diagram

3. Data Reverse Engineering

Data reverse engineering is defined as the application of analytical techniques to one or more legacy data sources to elicit structural information (e.g., term definitions, schema definitions) from the legacy source(s) in order to improve the database design or produce missing schema documentation 0. In this article, DRE is applied to relational databases only. However, since the relational model has only limited semantic expressability, in addition to the schema, DRE algorithm generates an E/R-like representation of the entities and relationships that are not explicitly defined in the legacy schema (but which exist implicitly).

Formally DRE can be described as follows. We are given a legacy database DB_L defined as ({R_1, R_2, ..., R_n}, D), where R_i denotes the schema of the i-th relation with attributes A_1, A_2, ..., $A_{m(i)}$, keys K_1, K_2, ..., $K_{m(i)}$, data $D = \{r_1(R_1), r_2(R_2), ..., r_n(R_n)\}$, such that $r_i(R_i)$ denotes the data (extent) for schema R_i at time t. Furthermore, DB_L have functional dependencies $F = \{F_1, F_2, ..., F_{k(i)}\}$ and inclusion dependencies $I = \{I_1, I_2, ..., I_{l(i)}\}$ expressing relationships among the relations in DB_L. The goal of DRE is to first extract $\{R_1, R_2, ..., R_n\}$, I, and F and then use I, F, D, and C_L (program code) to produce a semantically enhanced description of $\{R_1, R_2, ..., R_n\}$ that includes all relationships among the relations in DB_L (incl. those that are implicit), semantic descriptions of the relations as well as the business knowledge that is encoded in DB_L and C_L.

This approach to data reverse engineering for relational sources is based on the existing algorithms by Chiang 0,0 and Petit et al. 0. However, in order to reduce the dependency on human input, to eliminate some of the limitations of their algorithms (e.g., consistent naming of key attributes and the requirement that the legacy schema be modeled in 3NF), and to produce a semantically richer schema description at the end, their methodologies were improved in several ways.

The DRE algorithm is divided into two parts: schema extraction and semantic analysis, which operate in interleaved fashion. An overview of the two algorithms, which are comprised of eight steps, is shown in Figure 2. In addition to the modules that execute each of the eight steps, the architecture in Figure 2 includes three support components: the configurable Database Interface Module (upper-left hand corner), which provides connectivity to the underlying legacy source. Note that this component is the only source-specific component in the architecture: in order to perform knowledge extraction from different sources, only the interface module needs to be changed. The Knowledge Encoding (lower right-hand corner) represents the extracted knowledge in the form of an XML document. The Metadata Repository is internal to DRE and used to store intermediate run-time information needed by the algorithms including user input parameters and the abstract syntax tree for the code (e.g., from a previous invocation), etc.

We provide a trivial example of legacy system in order to highlight each of the eight steps and related activities outlined in Figure 2. Assume that the underlying legacy database DB_L is managed by a relational database management system. For simplicity, we assume without lack of generality or specificity that only the following relations exist in DB_L, whose schema will be discovered using DRE:

Projects [<u>Proj_ID</u>, ...]

Availability [<u>Proj_ID</u>, <u>Avail_UID</u>, ...]

Resources [<u>Proj_ID</u>, <u>Res_UID</u>, ...]

Tasks [<u>Proj_ID</u>, <u>Task_UID</u>, ...]

Assignment [<u>Proj_ID</u>, <u>Assn_UID</u>, ...]

In order to illustrate the code analysis and how it enhances the schema extraction, the following C code fragment is used representing a simple, hypothetical interaction with a legacy database:

```
char *aValue, *cValue;
int bValue = 0;
```

```
..........
/* more code */
..........
EXEC SQL SELECT T_StD, T_FinD INTO
:aValue, :cValue
FROM Tasks WHERE T_Dur = :bValue;
..........
/* more code */
..........
```

```
if (*cValue < *aValue)
{ cValue = aValue; }
..........
/* more code */
..........
printf("Task Start Date %s ",
aValue);
printf("Task Finish Date %s ",
cValue);
```

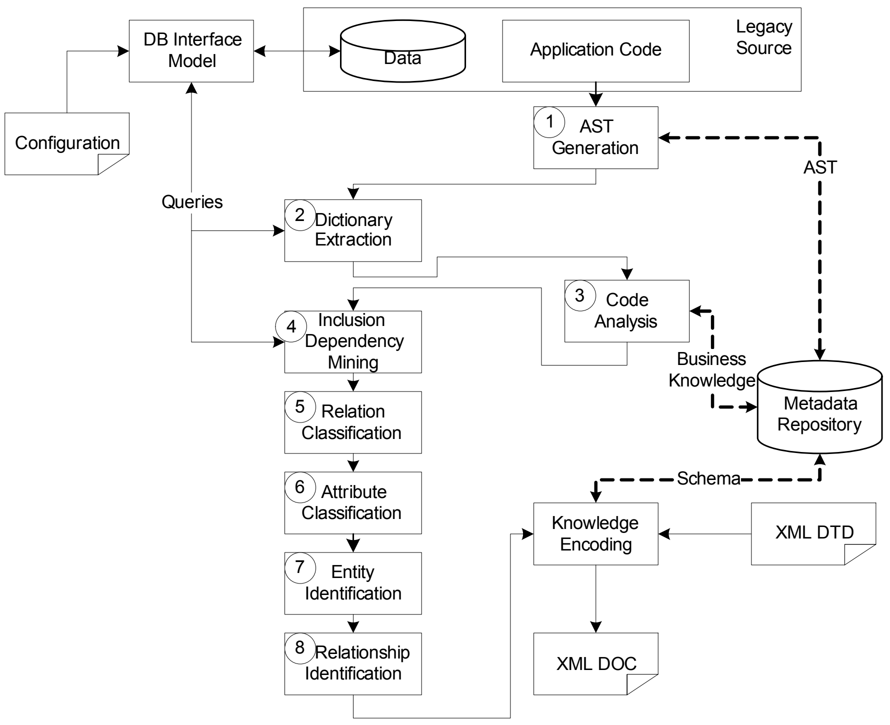

Figure 2. Conceptual view of the data reverse engineering algorithm

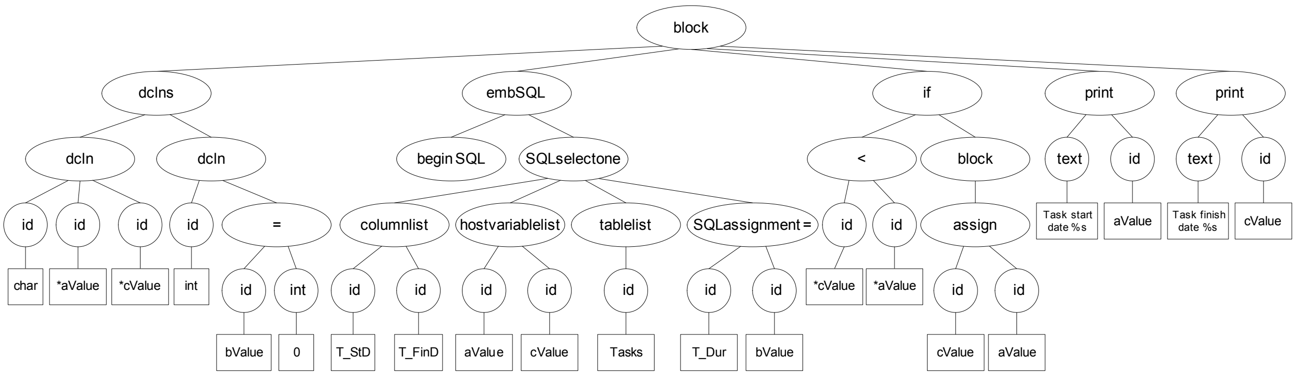

Figure 3. Abstract syntax tree for the legacy application code

3.1. Abstract syntax tree generation

The DRE process begins with the generation of an abstract syntax tree (AST) (AST is described in 0, 0, 0) for the legacy application code (Figure 3). The AST will be used by the semantic analyzer for code exploration during Step 3. Our objective in AST generation is to be able to associate "meaning" with program variables. For example, format strings in input/output statements contain semantic information that can be associated with the variables in the input/output statement. This program variable in turn may be associated with a column of a table in the underlying legacy database.

3.2. Dictionary Extraction

The goal of Step 2 is to obtain the relation and attribute names from the legacy source. This is done by querying the data dictionary, stored in the underlying database DB_L in the form of one or more system tables. Otherwise, if primary key information cannot be retrieved directly from the data dictionary, the algorithm passes the set of candidate keys along with predefined rule-out patterns to the code analyzer. The code analyzer searches for these patterns in the application code and eliminates those attributes from the candidate set, which occur in the rule-out pattern. The rule-out patterns, which are expressed as SQL queries, occur in the application code whenever programmer expects to select a SET of tuples. By definition of primary key, this rules out the possibility that the attributes $a_1 \ldots a_n$ form a primary key. Three sample rule-out patterns are:

```
1. SELECT DISTINCT <selection> FROM
   table
   WHERE a1=<expression1> AND
   a2=<expression2> AND … AND
   an=<expressionn>
2. SELECT <selection> FROM table
   WHERE a1=<expression1> AND
   a2=<expression2> AND … AND
   an=<expressionn>
   GROUP BY …
3. SELECT <selection> FROM table
   WHERE a1=<expression1> AND
   a2=<expression2> AND … AND
   an=<expressionn>
   ORDER BY …
```

Following code analysis, if a primary key cannot be identified, the reduced set of candidate keys must be presented to the user for final primary key selection.

Result. From the example legacy system, the following relations and their attributes were obtained:

Projects [Proj_ID, ...]

Availability [Proj_ID, Avail_UID, ...]

Resources [Proj_ID, Res_UID, ...]

Tasks [Proj_ID, Task_UID, ...]

Assignment [Proj_ID, Assn_UID, ...]

3.3. Code analysis

The objective of Step 3, code analysis, is twofold: (1) augment entities extracted in Step 2 with domain semantics, and (2) identify business rules and constraints not explicitly stored in the database, but which may be important to the process of reverse engineering. This approach to code analysis is based on program understanding, which includes slicing 0, 0, 0and pattern matching 0.

The first step is the construction of program dependency graph (PDG) from the abstract syntax tree. The PDG is constructed in the following three steps 0: 1) augmented control flow graph (ACFG) construction from the AST; 2) computation of the post dom graph from ACFG; 3) construction of the PDG using ACFG and the post dom graph. The PDG for the AST in Figure 3 is presented in Figure 4.

The next step is the pre-slicing. From the AST of the application code, the pre-slicer identifies all the nodes corresponding to input, output and embedded SQL statements. If an identifier node (which corresponds to the occurrence of a variable in that statement) exists in the subtree in that statement node, then it appends the actual variable name to the list of slicing variable. For example, for the AST in Figure 3, the array contains the following information depicted in Table 1. The identifiers that occur in this data structure maintained by the pre-slicer form the set of slicing variables.

Table 1. Information maintained by pre-slicer for slicing variables

Slicing Variable	Type of Statement	Direction of Slicing	Text string (only for print nodes)
aValue	Output	Backwards	“Task Start Date”
cValue	Output	Backwards	“Task Finish Date”

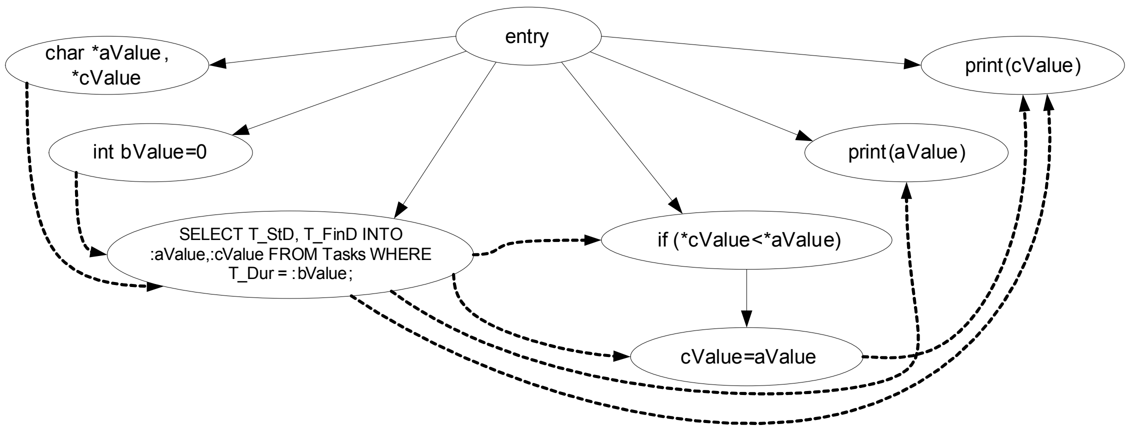

Figure 4. Program dependency graph for the legacy application code

The code slicer and analyzer, which represent sub-steps three and four, respectively, are executed once for each slicing variable identified by the pre-slicer. In the above example, the slicing variables that occur in SQL and output statements are aValue and cValue. The direction of slicing is fixed as backwards or forwards depending on whether the variable in question is part of an output (backwards) or input (forwards) statement. The slicing criterion is the exact statement (SQL or input or output) vertex that corresponds to the slicing variable.

During code slicing sub-step the flow and control edges of the PDG for the source code are followed and only those vertices are retained that were reached by traversal. The result of slice consists of the set of vertices and the set of edges induced by this vertex set that are relevant to the slice with respect to the slicing variable. Figure 5 shows backward slice for the PDG in Figure 4 with respect to `printf(cValue)` vertex. The reduced AST that correspond to the PDG in Figure 5 is shown in

Figure **6**.

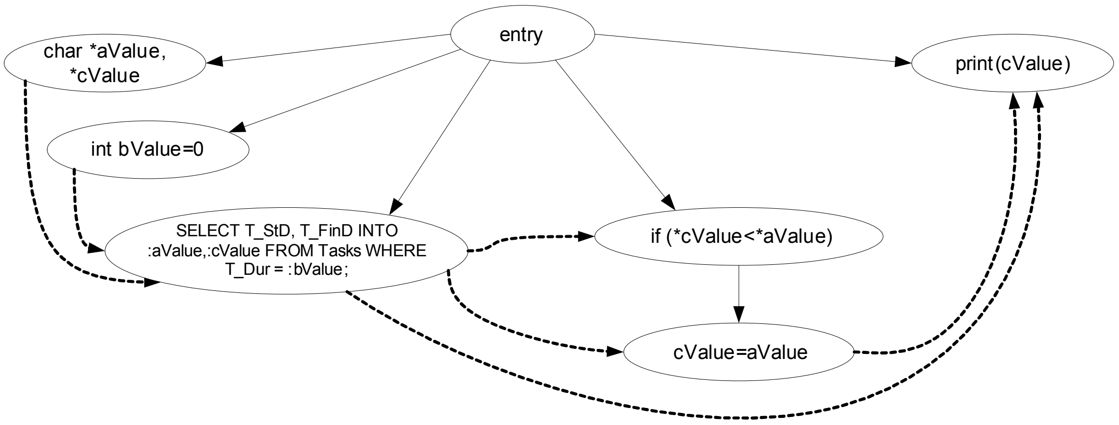

Figure 5. Backward slice for the legacy application code

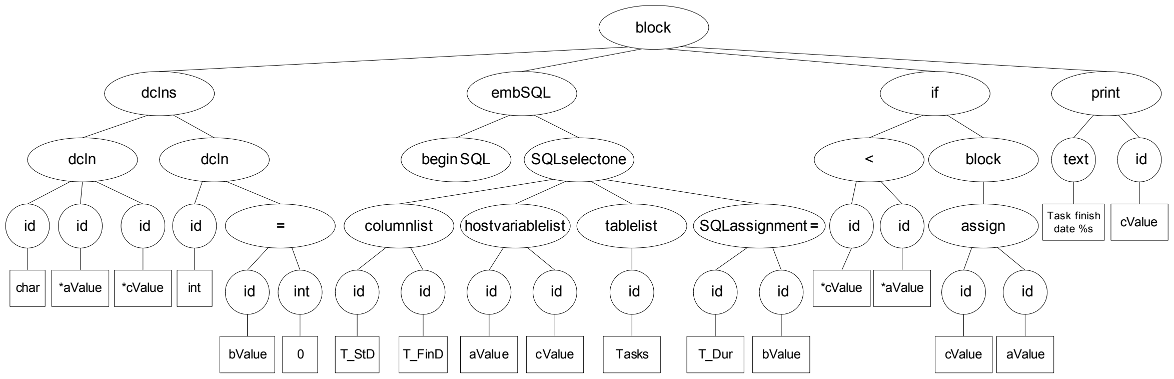

Figure 6. Reduced abstract syntax tree

Table 2. Information inferred during analysis sub-step

Identifier Name	Meaning	Possible Business Rule	Data type	Column Name	Table Name
aValue	Task Start Date		Char * => string	T_St_D	Tasks
cValue	Task Finish Date	if (*cValue < *aValue) { cValue = aValue; }	Char * => string	T_Fin_D	Tasks

During the analysis sub-step, algorithm extracts the information shown in Table 2, while traversing the reduced AST.

1. If a *dcln* node is encountered, the data type of the identifier can be learned.
2. *embSQL* contains the mapping information of identifier name to the corresponding column name and table name in the database.
3. *print*/*scanf* nodes contain the mapping information from the text string to the identifier. In other words, we can extract the meaning of the identifier from the text string.

It is important to note that enterprise knowledge is identified by matching templates against code fragments in the AST. So, patterns for discovering business rules must be developed which are encoded in loop structures and/or conditional statements and mathematical formulae, which are encoded in loop structures and/or assignment statements. Note, the occurrence of an assignment statement itself does not necessarily indicate the presence of a mathematical formula, but the likelihood increases significantly if the statement contains one of the slicing variables.

3.4. Inclusion Dependency Mining

After extraction of the relational schema in Step 2, the goal of Step 4 is to identify constraints to help classify the extracted relations, which represent both the real-world entities and the relationships among them. This is done using inclusion dependencies (INDs), which indicate the existence of inter-relational constraints including class/subclass relationships.

Let A and B be two relations, and X and Y be attributes or a set of attributes of A and B, respectively. An inclusion dependency $A.X << B.Y$ denotes that a set of values appearing in $A.X$ is a subset of $B.Y$. Inclusion dependencies are discovered by examining all possible subset relationships between any two relations A and B in the legacy source. Inclusion dependencies can be identified in an exhaustive manner as follows: for each pair of relations A and B in the legacy source schema, compare the values for each non-key attribute combination X in A with the values of each candidate key attribute combination Y in B (note that X and Y may be single attributes). An inclusion dependency $A.X << B.Y$ may be present if:

1. X and Y have the same number of attributes.
2. X and Y must have pair wise domain compatibility (matching data types and matching maximum length of attributes).
3. $A.X \subseteq B.Y$.

In order to check the subset criteria (3), the following generalized SQL query templates are provided, which are instantiated for each pair of relations and attribute combinations and run against the legacy source:

```
C1 = SELECT count (*) FROM A
WHERE X NOT IN (SELECT Y FROM B);
C2 = SELECT count (*) FROM B
WHERE Y NOT IN (SELECT X FROM A);
```

If $C1$ is zero, we can deduce that there may exist an inclusion dependency $A.X << B.Y$; likewise, if $C2$ is zero, there may exist an inclusion dependency $B.Y << A.X$. Note that it is possible for both $C1$ and $C2$ to be zero. In that case, we can conclude that the two sets of attributes X and Y are equal.

The worst-case complexity of this exhaustive search, given N tables and M attributes per table (NM total attributes), is $O(N^2M^2)$. However, the search space can be reduced in those cases where equi-join queries in the application code could be identified (during semantic analysis). Each equi-join query allows to deduce the existence of one or more inclusion dependencies in the underlying schema. In addition, using the results of the corresponding count queries the "direction" of the dependencies can also be determined. This allows to limit exhaustive searching to only those relations not mentioned in the extracted queries.

Result: Inclusion dependencies are as follows:

Assignment [Task_UID, Proj_ID] << **Tasks** [Task_UID, Proj_ID]

Assignment [Res_UID, Proj_ID] << **Resources** [Res_UID, Proj_ID]

Availability [Res_UID, Proj_ID] << **Resources** [Res_UID, Proj_ID]

Resources [Proj_ID] << **Projects** [Proj_ID]

Tasks [Proj_ID] << **Projects** [Proj_ID]

Assignment [Proj_ID] << **Projects** [Proj_ID]

Availability [Proj_ID] << **Projects** [Proj_ID]

The last two inclusion dependencies can be removed since they are implicitly contained in the inclusion dependencies listed in lines 2, 3 and 4 using the transitivity relationship.

3.5. Relation Classification

When reverse-engineering a relational schema, it is important to understand that due to the limited expressability of the relational model, all real-world entities are represented as relations irrespective of their types and role in the model. The goal of this step is to identify the different types of relations, some of which correspond to actual real-world entities while others represent relationships among them.

In this step, all the relations in the database are classified into one of four types – strong, regular, weak or specific. Identifying different relations is done using the primary key information obtained in Step 2 and the inclusion dependencies from Step 4. Intuitively, a strong entity-relation represents a real-world entity whose members can be identified exclusively through its own properties. A weak entity-relation represents an entity that has no properties of its own that can be used to identify its members. In the relation model, the primary keys of weak entity-relations usually contain primary key attributes from other (strong) entity-relations. Both regular and specific relations are relations that represent relationships between two entities in the real world (rather then the entities themselves). However, there are instances when not all of the entities participating in an (n-ary) relationship are present in the database schema (e.g., one or more of the relations were deleted as part of the normal database schema evolution process). While reverse engineering the database, such relationships must be identified as special relations.

Result:

Strong entity: **Projects**

Weak entity: **Resources**, **Tasks**, **Availability**

Regular relation: **Assignment**

3.6. Attribute Classification

Attributes are classified as (a) primary key (PK) or foreign key (FK), (b) dangling key (DK) or general key (GK), or (c) non-key (NK) (rest).

Result: Table 3 illustrates attributes obtained from the example legacy source.

Table 3. Attribute classification example

	PKA	DKA	GKA	FKA	NKA
Projects	Proj_ID				All remaining attributes
Resources	Proj_ID	Res_UID			
Tasks	Proj_ID	Task_UID			
Availability	Proj_ID	Avail_UID		Res_UID+Proj_ID	
Assignment	Proj_ID		Assn_UID	Res_UID+Proj_ID, Task_UID+Proj_ID	

3.7. Entity Identification

Strong (weak) entity relations obtained from Step 5 are directly converted into strong (resp. weak) entities.

Result: The following entities were classified:

Strong entities: **Projects** with Proj_ID as its key.

Weak entities:

Tasks with Task_UID as key and **Projects** as its owner.

Resources with Res_UID as key and **Projects** as its owner.

Availability with Avail_UID as key and **Resources** as its owner.

3.8. Relationship Identification

The inclusion dependencies discovered in Step 4 form the basis for determining the relationship types among the entities identified above. This is a two-step process:

1. Identify relationships present as relations in the relational database. The relation types (regular and specific) obtained from the classification of relations (Step 5) are converted into relationships. The participating entity types are derived from the inclusion dependencies. For completeness of the extracted schema, a new entity may be created when conceptualizing a specific relation.

   The cardinality between the entities is $M : N$.
2. Identify relationships among the entity types (strong and weak) that were not present as relations in the relational database, via the following classification:

- IS-A relationships can be identified using the PKAs of strong entity relations and the inclusion dependencies among PKAs. If there is an inclusion dependency in which the primary key of one strong entity-relation refers to the primary key of another strong entity-relation, then an IS-A relationship between those two entities is identified. The cardinality of the IS-A relationship between the corresponding strong entities is $1 : 1$.
- Dependent relationship: For each weak entity type, the owner is determined by examining the inclusion dependencies involving the correspon-ding weak entity-relation. The cardinality of the dependent relationship between the owner and the weak entity is $1 : N$.
- Aggregate relationships: If the foreign key in any of the regular and specific relations refers to the PKA of one of the strong entity relations, an aggregate relationship is identified. The cardina-lity of the aggregate relationship between the strong entity and aggregate entity (a $M : N$ relationship and its participating entities at the conceptual level) is as follows: if the foreign key contains unique values, then the cardinality is $1 : 1$, else the cardinality is $1 : N$.
- Other binary relationships: Other binary relation-ships are identified from the FKAs not used in identifying the above relationships. The cardina-lity of the binary relationship between entities is as follows: if the foreign key contains unique values, the cardinality is $1 : 1$, else the cardinality is $1 : N$.

Result: The following $1 : N$ binary relationships between the following entities were discovered:

Between **Projects** and **Tasks**

Between **Projects** and **Resources**

Between **Resources** and **Availability**

Since two inclusion dependencies involving Assignment exist (i.e., between Tasks and Assignment and between Resources and Assignment), there is no need to define a new entity. Thus, Assignment becomes an $M : N$ relationship between Tasks and Resources.

3.9. Knowledge Representation

8 steps of DRE enable the extraction of the following schema information from the legacy database:

- Names and classification of entities;
- Names of attributes;

- Primary and foreign keys;
- Data types;
- Simple constraints (e.g., Null, Unique) and explicit assertions;
- Relationships and their cardinalities;
- Business rules.

A conceptual overview of the extracted schema is represented by the entity-relationship diagram shown in Figure 7.

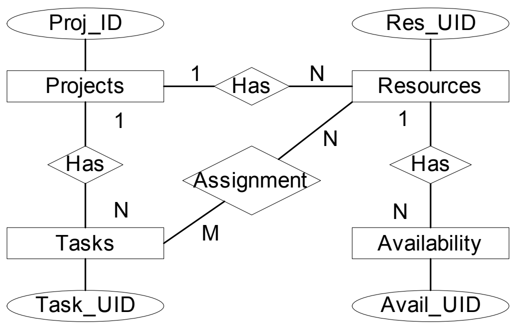

Figure 7. E/R diagram representing the extracted schema

4. Conclusion

Legacy information systems contain incredible detailed business rules and form the backbone of the information flow of organization, but their maintenance is very expensive and it is very difficult, if not impossible, to expand them. Reverse engineering is the essential part of process of changing and replacing legacy systems. Its main objective is to discover and extract business knowledge from legacy sources. Reverse engineering builds the powerful foundation for renovation of IT systems that enables the application of new technologies and programs.

Data reverse engineering algorithm provided in this article is comprised of schema extraction and semantic analysis. The most important techniques of semantic analysis as program slicing, and pattern matching, together with the more conventional approaches of lexical/syntactic analysis semantically enhance the extracted schema from database. Two algorithms which operate in interleaved fashion lead to reduced need of human input compared to other existing methods.

8 steps of data reverse engineering enable the extraction of names and classification of entities, names of attributes, primary and foreign keys, data types, simple constraints and explicit assertions, relationships and their cardinalities, business rules. This knowledge then could be used when changing or replacing legacy systems, i.e. when redeveloping, wrapping, or migrating legacy systems.

References

[1] **G. Bakehouse, T. Wakefield**. Legacy Information Systems. *ACCA*, 2005, *internet resource*: *<http://www. accaglobal.com/>*.

[2] **T. Ball, S. Horwitz**. Slicing Programs with Arbitrary Control Flow. *Technical Report* 1128, 1992, *internet resource: <http://citeseer.ist.psu.edu/>*.

[3] **J. Bisbal, D. Lawless, B. Wu, and J. Grimson**. Legacy Information System Migration: A Brief Review of Problems, Solutions and Research Issues. *Technical Report TCD-CS*1999-38, 1999, *internet resource: <https://www.cs.tcd.ie/>*.

[4] **S. Carr**. An Abstract Syntax Tree for Nolife. 2002, *internet resource: <http://www.cs.mtu.edu/>*.

[5] **R.H. Chiang**. A Knowledge-Based System for Performing Reverse Engineering of Relational Database. *Decision Support Systems*, 1995, 13, 295-312.

[6] **R.H.L. Chiang, T.M. Barron, V.C. Storey**. Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database. *Data and Knowledge Engineering*, 1994, 12:1, 107-142.

[7] **J. Hammer, M. Schmalz, W. O. Brien, S. Shekar, and N. Haldavnekar**. Knowledge Extraction in the SEEK Project Part I: Data Reverse Engineering. *Technical Report TR*-0214, 2002, *internet resource*: *<http://www.cise.ufl.edu/>*.

[8] **J. Henrard**. Program Understanding in Database Reverse Engineering. *Thesis submitted for the degree of Doctor of Science*, 2003, *internet resource*: *<http://edoc.bib.ucl.ac.be:61/>*.

[9] **S. Horwitz, T. Reps**. The Use of Program Dependence Graphs in Software Engineering. *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992, *internet resource*: *<http://www.cs.wisc.edu/>*.

[10] **S. Horwitz, T. Reps, D. Binkley**. Interprocedural Slicing Using Dependency Graphs. *ACM Transactions on Programming Languages and Systems*, 1990, 12, *internet resource*: *http://www.cs.wisc.edu/>*.

[11] **J. L. Nhampossa**. Strategies to Deal with Legacy Information Systems: A Case Study From the Mozambican Health Sector. *IRMA*, 2004, *internet resource*: *<http://heim.ifi.uio.no/>*.

[12] **S. Paul, A. Prakash**. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 1994, 20(6), 463-475, *internet resource*: *<http://citeseer.ist.psu.edu/>*.

[13] **J.M. Petit, F. Toumani, J.F. Boulicaut, J. Kouloumdjian**. Towards the Reverse Engineering of Denormalized Relational Databases. *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, 1996, 218-227.

[14] **C.H. Stork, V. Haldar**. Compressed Abstract Syntax Trees for Mobile Code. *Workshop on Intermediate Representation Engineering*, 2001, *internet resource*: *<http://www.ics.uci.edu/>*.

[15] **D. S. Wile**. Toward a Calculus for Abstract Syntax Trees. *Proceedings of a Workshop on Algorithmic Languages and Calculii*, 1997, 324-352, *internet resource*: *<http://mr.teknowledge.com/>*.

Received June 2006.