Knowledge Extraction in the SEEK Project Part I: Data Reverse Engineering

Technical Report

J. Hammer, M. Schmalz, W. O’Brien, S. Shekar and N. Haldavnekar
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611-6120
jhammer@cise.ufl.edu

Abstract

In this report we describe our methodology for knowledge extraction in the SEEK (Scalable Extraction of Enterprise Knowledge) project and highlight the underlying technologies supporting SEEK. In particular, we discuss our use of data reverse engineering and code mining techniques to automatically infer as much as possible the schema and semantics of a legacy information system. We have a fully functional prototype implementation, which we use to illustrate the approach using an example from our construction supply chain testbed. We also provide empirical evidence to the usefulness and accuracy of our methodology.

1 SEEK Approach to Knowledge Extraction

The goal of SEEK is to semiautomatically discover and extract enterprise knowledge from legacy sources in order to support the generation of source wrappers. Wrappers, which are “…a type of software "glueware" that is used to attach together other software components …” [26] are essential in enabling the exchange of data between heterogeneous systems. Such systems employ different data models, representations, and query languages to manage their data: a (source) wrapper converts a query that is formulated in the language of one system into one or more queries understood by a second system. Analogously, it converts the resulting data, which is expressed in the model of the second system, into the format used by the first. Existing wrapper development tools (see, for example, [11, 13, 21, 24]) require significant programmatic set-up with limited reusability of code. Given the diversity and number of information sources available today, the time and investment needed to establish connections to legacy sources has imposed severe limitations on the scalability and maintainability of such systems. These requirements have generally acted as a significant barrier to information integration in enterprise development (e.g., supply chain automation). Efforts are now under way to develop languages and tools for describing resources in a way that can be processed by computers (e.g., Semantic Web). However, they do not address the problem of how to collect this enterprise knowledge, or how to maintain it efficiently for the continuously increasing number of legacy sources.

A high-level view of the SEEK knowledge extraction architecture is shown in Figure 1. SEEK applies Data Reverse Engineering (DRE) and Schema Matching (SM) processes to legacy database(s), to produce a source wrapper for a legacy source. The source wrapper will be used by another component (not shown in Figure 1) wishing to communicate and exchange information with the legacy system. We assume that the legacy source uses a database management system for storing and managing its enterprise data.
First, SEEK generates 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 enterprise knowledge. The extracted enterprise knowledge forms a knowledgebase that serves as input for subsequent steps outlined below. In order to extract this enterprise knowledge, the DRE module shown on the left of Fig. 1 connects to the underlying DBMS to extract schema information (most data sources support at least some form of Call-Level Interface such as JDBC). The schema information from the database is semantically enhanced using clues extracted by the semantic analyzer from available application code, business reports, and, in the future, perhaps other electronically available information that may encode business data such as e-mail correspondence, corporate memos, etc. It has been our experience (through visits with representatives from the construction and manufacturing domains) that such application code exists and can be made available electronically.

![Diagram](image)

**Figure 1:** Schematic diagram of the conceptual architecture of SEEK’s knowledge extraction algorithm.

Second, the semantically enhanced legacy source schema must be mapped into the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema matching process that produces the mapping rules between the legacy source schema and the application domain model. In addition to the domain model, the schema matching module also needs access to the domain ontology (DO) describing the model.

Finally, the extracted legacy schema and the mapping rules provide the input to the wrapper generator (not shown), which produces the source wrapper.

The three preceding steps can be formalized as follows. At a high level, let a legacy source \( L \) be denoted by the tuple \( L = (DB_L, S_L, D_L, Q_L) \), where \( DB_L \) denotes the legacy database, \( S_L \) denotes its schema, \( D_L \) the data and \( Q_L \) a set of queries that can be answered by \( DB_L \). Note, the legacy database need not be a relational database, but can include text, flat file databases, and hierarchically formatted information. \( S_L \) is expressed by the data model \( DM_L \).

We also define an application via the tuple \( A = (S_A, Q_A, D_A) \), where \( S_A \) denotes the schema used by the application and \( Q_A \) denotes a collection of queries written against that schema. The symbol \( D_A \) denotes data that is expressed in the context of the application. We assume that the application schema
is described by a domain model and its corresponding ontology (as shown in Figure 1). For simplicity, we further assume that the application query format is specific to a given application domain but invariant across legacy sources for that domain.

Let a legacy source wrapper \( W \) be comprised of a query transformation

\[
f_w^Q : Q_A \mapsto Q_L
\]

and a data transformation

\[
f_w^D : D_L \mapsto D_A,
\]

where the \( Q \)'s and \( D \)'s are constrained by the corresponding schemas.

The SEEK knowledge extraction process shown in Figure 1 can now be stated as follows. Given \( S_A \) and \( Q_A \) for an application wishing to access legacy database \( DB_L \) whose schema \( S_L \) is unknown. Assuming that we have access to the legacy database \( DB_L \) as well as to application code \( C_L \), accessing \( DB_L \), we first infer \( S_L \) by analyzing \( DB_L \) and \( C_L \), then use \( S_L \) to infer a set of mapping rules \( M \) between \( S_L \) and \( S_A \), which are used by a wrapper generator \( WGen \) to produce \( (f_w^Q, f_w^D) \). In short:

\[
\begin{align*}
DRE: (DB_L, C_L) & \mapsto S_L \quad (3.1) \\
SM: (S_L, S_A) & \mapsto M \quad (3.2) \\
WGen: (Q_A, M) & \mapsto (f_w^Q, f_w^D) \quad (3.3)
\end{align*}
\]

In this report, we provide a description of our \( DRE \) algorithm (equation 3.1) which is comprised of schema extraction and semantic analysis. We also provide a description of the \( DRE \) prototype that is installed and running in our testbed and summarize the experimental results validating our approach. We report on \( SM \) and \( WGen \) in subsequent publications.

2 Data Reverse Engineering (DRE)

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. So far in SEEK, we are applying DRE to relational databases only. However, since the relational model has only limited semantic expressability, in addition to the schema, our 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).

More formally, DRE can be described as follows: Given a legacy database \( DB_L \) defined as \( \{ R_1, R_2, \ldots, R_n \} \), where \( R_i \) denotes the schema of the \( i \)-th relation with attributes \( A_{i1}, A_{i2}, \ldots, A_{im(i)} \), keys \( K_{i1}, K_{i2}, \ldots, K_{in(i)} \) and data \( D = \{ r_1(R_1), r_2(R_2), \ldots, r_n(R_n) \} \), such that \( r_i(R_i) \) denotes the data (extent) for schema \( R_i \) at time \( t \). Furthermore, \( DB_L \) has functional dependencies \( F = \{ F_1, F_2, \ldots, F_{k(i)} \} \) and inclusion dependencies \( I = \{ I_1, I_2, \ldots, I_{i(i)} \} \), expressing relationships among the relations in \( DB_L \). The goal of DRE is to extract the relations \( \{ R_1, R_2, \ldots, R_n \} \), \( I \), and \( F \) and then use \( I \), \( F \), \( D \), and \( C_L \) to produce a semantically enhanced description of \( \{ R_1, R_2, \ldots, R_n \} \) 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 \( C_L \).

Our approach to data reverse engineering for relational sources is based on existing algorithms by Chiang [4, 5] and Petit [23]. However, we have improved their methodologies in several ways, most importantly 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.

Our 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-right 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 Encoder (lower right-hand corner) represents the extracted knowledge in the form of an XML document so that it can be shared with other components in the SEEK architecture (e.g., the semantic matcher). 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.

Figure 2: High-level overview of the DRE algorithm.

2.1 Eight-Step Data Reverse Engineering Algorithm

We now describe each of the eight steps and related activities outlined in the DRE algorithm diagram of Figure 2.

Step 1: Generation of an Abstract Syntax Tree (AST) for the Application Code

The DRE process begins with the generation of an abstract syntax tree (AST) for the legacy application code. The AST will be used by the semantic analyzer for code exploration during Step 3. The following discussion references Figure 3, which is an expansion of the AST Generator representation shown in Figure 2. The process flow on the left side is for building ASTs for C code, the one on the right side is for developing ASTs for Java code.

The AST generator for C code consists of two major components: the lexical analyzer and the parser. The lexical analyzer for application code written in C reads the source code line-by-line and breaks it up into tokens. The C parser reads in these tokens and builds an AST for the source code in accordance with language grammar (see Appendix A for a listing of the grammar for the C code that is accepted by the semantic analyzer). The above approach works nicely for procedural languages such as C. However, when applied directly to object oriented languages (e.g., Java), it greatly increases the
complexity of the problem due to required resolution of issues such as ambiguity induced by multiple inheritance, diversity resulting from specialization of classes and objects, etc.

**Figure 3:** Generation of an AST for either C or Java code.

Unfortunately, most application code written for currently-available databases is written in Java making it necessary to develop an algorithm to infer semantic information from Java code. As previously implied, the grammar of an object-oriented language is complex when compared with procedural languages like the C language. Building a Java lexical analyzer and parser would require the parser to look ahead multiple tokens before applying the appropriate production rule. Thus, building a Java parser from scratch does not seem a feasible solution. Instead, tools like lex or yacc can be employed to do the parsing. These tools generate N-ary AST’s. N-ary trees, unlike binary trees, are difficult to navigate using standard tree traversal algorithms. Our objective in AST generation is to be able to extract and associate the meaning of selected partitions of application code 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. Standard Java language grammar does not put the format string information on the AST, which defeats the purpose of generating AST’s for the application code.

The above stated reasons justify the need to use an alternate approach for analyzing Java code. Our Java AST builder (depicted on the right-hand side of Figure 3) has three major components, the first of which is a pattern matching process. The objective the pattern matching is twofold. First, it reduces the size of the application code being analyzed. Second, while generating the reduced application code file, it performs selected text replacements that facilitate easier parsing of the reduced source code. The pattern matcher works as follows: It scans the source code line by line looking for patterns such as `System.out.println` that indicate output statements or `ResultSet` that indicate JDBC statements. Upon finding such a pattern, it next replaces the pattern with an appropriate pre-designated string. For example, in the following line of code:

```java
System.out.println("Task Start Date" + aValue);
```

the pattern `System.out.println` is replaced with `printf`, and following line is generated in a reduced source code file:

```java
printf("Task Start Date" + aValue);
```

After one pass of the application code, the pattern matcher generates a reduced source code file that contains only JDBC and output statements, which more closely resemble a procedural language.
In writing a lexical analyzer and parser for this reduced source code, we can re-use most of our C lexical analyzer and parser. The lexical analyzer reads the reduced source code line by line and supplies tokens to the parser that builds an AST in accordance with the Java language grammar.

**Step 2: Extracting Schema Information using the Dictionary Extractor.**

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 legacy database $DB_L$ in the form of one or more system tables. The details of this step are outlined in Figure 4.

![Figure 4: Implementation detail for dictionary extraction.](image-url)

In order to determine key attributes, the algorithm proceeds as follows: For each relation, it first identifies the set of candidate key attributes, which are those attributes whose values are restricted through NON-NULL and UNIQUE constraints. If there is only one candidate key per entity, then that key is the primary key. 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 $a_1$ … $a_n$ form a primary key. Three sample rule-out patterns are:

```sql
SELECT DISTINCT <selection>
FROM T
WHERE a_1=<scalar_expression_1> AND a_2=< scalar_expression_2> AND ... AND a_n =<scalar_expression_n>

SELECT <selection>
FROM T
WHERE a_1=<scalar_expression_1> AND a_2=< scalar_expression_2> AND ... AND a_n =<scalar_expression_n>
GROUP BY ...

SELECT <selection>
FROM T
WHERE a_1=<scalar_expression_1> AND a_2=< scalar_expression_2> AND ... AND a_n =<scalar_expression_n>
ORDER BY ...
```
Following code analysis, if a primary key cannot be identified, the reduced set of candidate keys is presented to the user for final primary key selection.

**Step 3: Code Analysis.**

The objective of the application code analysis that comprises Step 3 is threefold: (1) augment entities extracted in Step 2 with domain semantics, (2) extract queries that help validate the existence of relationships among entities in Step 2, and (3) identify business rules and constraints not explicitly stored in the database, but which may be important to the wrapper generator or application program accessing legacy source \( L \). Our approach to code analysis is based on code mining, which includes code slicing [15] and pattern matching [22].

The mining of semantic information from source code assumes that in the application code there are output statements that support report generation or display of query results. From output message string(s) that usually describe a displayed variable \( v \), semantic information about \( v \) can be obtained. This implies location (tracing) of the statement \( s \) that assigns a value to \( v \). Since \( s \) can be associated with the result set of a query \( q \), we can associate \( v \)’s semantics with a particular column of the table being accessed in \( q \).

![Figure 5: Implementation detail for code analysis.](image)

The details of the code analyzer are shown in Figure 5. The first step is the pre-slicer, which extracts slicing variables from input, output and SQL statements from the AST. For each of the slicing variables identified by the pre-slicer, the code slicer and analyzer are invoked on the AST. The code slicer traverses the AST in pre-order and retains only those nodes that contain the slicing variable in their sub-tree. The reduced AST constructed by the code slicer is then sent to the semantic analyzer that extracts the data type, meaning, business rules, column name, and table name which can be associated with the slicing variable. The results of semantic analysis are appended to a result file and the slicing variable is stored in the metadata repository. Note that the code slicer and analyzer could have been invoked once with all the slicing variables generated by the pre-slicer rather than for each slicing variable in an iterative fashion. The second approach of invoking the code slicer and analyzer separately for each slicing variable (which is the approach used here) yields a smaller reduced AST and, therefore, more focused results. Since the code analyzer is part of a build time activity, accuracy of the results rather than time is the critical factor.
After the code slicer and analyzer have been invoked on all slicing variables, the result generator examines the result file and replaces the variables in the extracted business rules with the semantics extracted from their associated output statements (where possible). The results of code analysis obtained thus far are presented to the user, who has a chance to view the results and decide if further code analysis is needed. Should the user indicate that further analysis is needed, a list of variables is displayed that occur in input, output, and SQL statements as well as the slicing variables used in previous passes. In addition, the user can enter a new slicing variable until the user determines that no new knowledge can be obtained from the code, or until there are no further candidate slicing variables. The results of this user-controlled code analysis are appended to the result file.

From the AST created in Step 1, the code analyzer identifies all nodes corresponding to input, output and embedded SQL statements. The node number, statement node name, and identifier list is appended to a symbol table during pre-order traversal of the AST.

In practice, the slicing variables are the program identifiers (variables) identified by the pre-slicer and are classified based on whether they appear in input, output, or SQL statements. The code slicer module takes each of these slicing variables, traverses the source AST and retains only nodes that satisfy the slicing criteria. Here satisfying the slicing criteria would mean occurrence of the slicing variable in a subtree rooted at the node in question. A reduced AST is produced, as follows:

1. Locate the AST start/end node for slicing. Forward slicing completely traverses the AST in pre-order from the start node. Backward slicing traverses the AST in pre-order, stopping when the pre-specified end node is reached.
2. Traverse the AST in pre-order starting at the root. Each time a declaration (dcln), statement, or assign node is encountered, search the node’s subtree for occurrences of user-specified slicing variable(s). If such variable(s) is(are) found, then push the subtree onto a stack.
3. If an assign statement with the slicing variable on its right-hand side is found, then add the variable on the left-hand side (i.e., the identifier name) to the list of slicing variables.
4. Repeat Steps 2 and 3, above, until the pre-specified end node is reached.
5. Pop all elements off the stack, building a tree using the left-child/right-sibling representation of N-ary trees.

The result of this step is a reduced AST. Program semantics are determined from pre-order traversal of the reduced AST, as follows:

- If a dcln node is encountered, the data type of the identifier can be discovered.
- embSQL nodes contain mapping information of identifier name to the corresponding column name and table name in the database.
- print nodes contain mapping information from the text string to the identifier. In other words, one can extract the ‘meaning’ of the identifier from the text string.

In addition, to the preceding program semantics, we identify other enterprise knowledge by matching templates against code fragments in the AST. So far, we have developed patterns for discovering business rules which are encoded in loop structures and/or conditional statements and mathematical formulae, which are encoded in loop structures and/or assignment statements. Note however, that the occurrence of an assignment statement does not necessarily indicate the presence of a mathematical formula, but the likelihood increases significantly (for example) if the statement contains one of the slicing variables. We illustrate the extraction of simple business rules in an example in Section 3.

**Step 4: Discovering Inclusion Dependencies.**

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.
Inclusion dependencies are defined as follows. 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.

As depicted in Figure 6, the inclusion dependency detection module obtains its input from two sources: (1) the dictionary extraction module (via the send/receive module) provides the table name, column names, primary keys and foreign keys (if available); and (2) the equi-join query finder is a part of the code analyzer that operates on the AST and provides pairs of relations and corresponding attributes, which occur together in equi-join queries in the application code. The fact that two relations are used in a join operation is evidence for the existence of an inclusion dependency between them. The inclusion dependency detection algorithm operates as follows:

1. Create a set X of pairs of relations: e.g. if we have relations P, Q, R and S then X = {(P,Q), (P,R), (P,S),(Q,R),(Q,S),(R,S)}. Intuitively, this set will contain pairs of relations for which inclusion dependencies have not been determined yet. In addition, we maintain two (initially empty) sets of possible (POSSIBLE) and final (FINAL) inclusion dependencies.

2. If foreign keys have been successfully extracted, do the following for each foreign key constraint:
   a) Identify the pair of participating relations, i.e. the relation to which the FK belongs and the relation to which it is referring.
   b) Eliminate the identified pair from set X.
   c) Add the inclusion dependency involving this FK to the set FINAL.

3. If equi-join queries have been extracted from the code, do the following for each equi-join query:
   a) Identify the pair of participating relations.
   b) Check the direction of the resulting inclusion dependency by querying the data. In order to check the direction of an inclusion dependency, we use a subset test described in Appendix B.
   c) If the above test is conclusive, eliminate the identified pair from set X and add the inclusion dependency involving this FK to the set FINAL.
   d) If the test in step b) is inconclusive (i.e. the direction cannot be finalized) add both candidate inclusion dependencies to the set POSSIBLE.
4. For each pair \( p \) remaining in \( X \), identify attributes or attribute combinations that have the same data type. Check whether the subset relationship exists by using the subset test described in Appendix B. If so, add the inclusion dependency to the set POSSIBLE. If, at the end of Step 4, no inclusion dependency is added to the possible set, delete \( p \) from \( X \); otherwise, leave \( p \) in \( X \) for user verification.

5. For each inclusion dependency in the set POSSIBLE, do the following:
   a) If the attribute names on both sides are equal, then assign the rating value of high.
   b) If the attribute name on left side of the inclusion dependency is related (based on common substrings) to the table name on the right hand side, then assign the rating value of high.
   c) If both conditions are not satisfied, then assign the rating value of low.

6. For each pair in \( X \), present the inclusion dependencies and their ratings in the set POSSIBLE to the user for final determination. Based on the user input, append the valid inclusion dependencies to the set FINAL.

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, we have reduced the search space in those cases where we can identify equi-join queries in the application code. This allows us to limit our exhaustive searching to only those relations not mentioned in the extracted queries. As a result, the average case complexity of the inclusion dependency finder is much smaller: The detection of one foreign key constraint in the data dictionary or one equi-join query in the application code allows the algorithm to eliminate the corresponding relation(s) from the search space. Hence, if \( K \) foreign key constraints and \( L \) equi-join queries (involving pairs different from the pairs that are part of the foreign key constraints) are detected, the average complexity is \( O((N^2 - K - L)M^2) \). In the best-case scenario when the \( K + L \) equals all possible pairs of relations, then the inclusion dependency detection can be done in constant time i.e. \( O(1) \).

In addition, the factors such as matching data types and matching maximum length of attributes (e.g. \texttt{varchar(5)} ) are used to reduce the number of queries to be made to the database (Step 4) to check subset relationship between attributes. If the attributes in a pair of relations have \( T \) mutually different data types then the \( N^2 \) part reduces to \( N(N-T) \). Finally, it is important to note that the DRE activity is always considered a built-time activity and hence the performance complexity is not a crucial issue.

**Step 5: Classification of the Relations.**

When reverse engineering a relational schema, it is important to understand that due to the limited expressibility 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 real-world entities while others represent relationships among such entities or among relations.

Identifying different relations is performed with the primary key information obtained in Step 2 and the inclusion dependencies from Step 4. Specifically if the consistent naming is used, the primary key of each relation is compared with primary keys of other relations to identify strong or weak entity-relations and specific or regular relations. Otherwise we have to use inclusion dependencies to give vital clues. Intuitively, a strong entity-relation represents a real-world entity whose members can be identified exclusively through its own properties. A weak entity-relation on the other hand, 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.

Intuitively, both regular and specific relations are relations that represent relationships between two entities in the real world (rather than 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, we identify such relationships as special relations. Specifically, the primary key of a specific relation is only partially formed by the primary keys of the participating (strong or weak) entity-relations, whereas the key of a regular relation is made up entirely of the primary keys of the participating entity-relations.

More formally, Chiang [4] defines the four relation types as follows:

• A strong entity relation is a relation whose primary key does not properly contain a key attribute of any other relation.

• A weak entity relation $\rho$ is a relation which satisfies the following three conditions:
  1. A proper subset of $\rho$'s PK contains key attributes of other strong or weak entity relations;
  2. The remaining attributes of $\rho$'s PK do not contain key attributes of any other relation; and
  3. $\rho$ has an identifying owner and properly contains the PK of its owner relation. User input is required to confirm these relationships.

• A regular relation has a PK that is formed by concatenating the PKs of other (strong or weak) entity relations.

• A specific relation $\tau$ is a relation which satisfies the following two conditions:
  1. A proper subset of $\tau$'s PK contains key attributes of other strong or weak entity relations;
  2. The remaining attributes of $\tau$'s PK do not contain key attributes of any other relation.

Classification proceeds as follows: First, strong and weak entity-relations are classified. For weak entity-relations, the primary key must be composite and part of it must be a primary key of an already identified strong entity-relation. The remaining part of the key must not be a primary key of any other relation. Finally, regular and specific relations are discovered. This is done by checking the primary keys or the remaining un-classified relations for full or partial presence of primary keys of already identified entity-relations.

**Step 6: Classification of the Attributes.**

Each relation must have its attributes classified into one of four groups to depending on whether they can be used as keys for entities, weak entities, relationships etc. Attribute classification is based on the type of relation and inclusion dependencies to which they belong:

• **Primary key attributes** (PKA) are attributes that uniquely identify the tuples in a relation.

• **Dangling key attributes** (DKA) are attributes belonging to the primary key of a weak entity-relation or specific relation that do not appear as the primary key of other relations.

• **Foreign key attributes** (FKA) are attributes in R1 referencing R2 if
  o the attributes of FK have the same domains as the primary key attributes PK of R2
  o for each $t_1$ in $r(R1)$ and $t_2$ in $r(R2)$, either $t_1[FK] = t_2[PK]$, or $t_1[FK]$ is null.

• **Non-key attributes** (NKA) are those attributes that cannot be classified as PKA, DKA, or FKA.

**Step 7: Identification of Entity Types.**

Starting with this step, we begin to map relational concepts into corresponding E/R model concepts. Specifically, the strong and weak entity relations identified in step 5 are classified as either strong or weak entities respectively. Furthermore, for each weak entity we associate its owner entity. The mapping, which includes the identification of proper keys, is done as follows:
• Each weak entity relation is converted into a weak entity type. The dangling key attribute of the weak entity relation becomes the key attribute of the entity.
• Each strong entity relation is converted into a strong entity type.

Step 8: Identification of Relationship Types.

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, we may decide to create a new entity 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 corresponding weak entity-relation as follows: we look for an inclusion dependency whose left hand side contains the part of the primary key of this weak entity-relation. When we find such an inclusion dependency, the owner entity can be easily identified by looking at the right hand side of the inclusion dependency. As a result, a binary relationship between the owner (strong) entity type and weak entity is created. 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. An inclusion dependency must exist from this (regular or specific) relation on the left-hand side that refers to some strong entity-relation on the right-hand side. The aggregate relationship is between the relationship (which must already be conceptualized in Substep 8.1 from a regular/specific relation) and the strong entity on right hand side.
     The cardinality 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 relationships are identified from the FKAs not used in identifying the above relationships. When an FKA of a relation refers to a primary key of another relation, then a binary relationship is identified. The cardinality 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.

2.2 Knowledge Representation

At the end of Step 8, DRE has extracted the following schema information from the legacy database:
• Names and classification of all entities;
• Names of all attributes;
• Primary and foreign keys;
• Data types;
• Simple constraints (e.g., Null, Unique) and explicit assertions;
• Relationships and their cardinalities; and
• Business rules.

The schema extraction and semantic analysis collectively generate information about the underlying legacy source. In each step of the DRE algorithm, knowledge is extracted from the source. This knowledge has to be represented in the format that is not only computationally tractable and easy to manipulate but also easy and intuitive for human understanding. The best choice in current scenario is XML as it is simple yet robust language to represent and manipulate data. The knowledge encoder takes an XML DTD as input (shown in Appendix C) and encodes the extracted information to produce XML document. Instead of encoding extracted information after every step, which may result in inconsistencies, as DRE algorithm refines some of its intermediate outcomes in the process, the encoding is done at the last because that works as a consistency check as well. We next exemplify the DRE process using an example.

3 Illustrative Example

Project management is a key application in building construction, hence the sample legacy source is based on a Microsoft Project application from our construction supply chain testbed. 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 (for a description of the entire schema refer to [19]):

- $Proj \left[ P_ID, P_NAME, DES_S, DES_F, A_S, A_F, \ldots \right]$
- $Avail \left[ PROJ_ID, AVAIL_UID, RES_ID, AVAIL_FROM, AVAIL_TO, UNITS \right]$
- $Res \left[ PROJ_ID, RES_UID, RES_NAME, R_ACWP, R_BCWP, R_BCWS, \ldots \right]$
- $T \left[ PROJ_ID, T_UID, T_ID, T_NAME, T_PRITY, T_DUR, T_ST_D, T_FIN_D, \ldots \right]$
- $Assn \left[ PROJ_ID, ASSN_UID, T_UID, R_ID, ASSN_BASE_C, ASSN_ACT_W, \ldots \right]$

In order to illustrate the code analysis and how it enhances the schema extraction, we refer the reader to the following C code fragment representing a simple, hypothetical interaction with a legacy database (only those statements are shown which are relevant to the example).

```c
char *aValue;
char *cValue;
int bValue = 0;
/* more code */
EXEC SQL SELECT T_ST_D, T_FIN_D INTO :aValue,:cValue FROM T
WHERE T_PRITY = :bValue;
/* more code */
int flag = 0;
if (cValue < aValue)
{
    flag = 1;
}
/* more code */
printf("Task Start Date %s ", aValue);
printf("Task Finish Date %s ", cValue);
```

13
Step 1: Creation of AST

We start by creating an Abstract Syntax Tree shown in Figure 7.

![Figure 7](image)

Figure 7: Application-specific code analysis via AST decomposition and code slicing. The direction of slicing is backwards (forward) if the variable in question is in an output (resp. input or declaration) statement.


Step 2 extracts table and attribute names from the legacy source. A decision step directs control to the pattern matcher if PKs cannot be obtained. Using the AST from Step 1, the pattern matcher examines embedded SQL queries (e.g., SELECT and WHERE clauses), yielding a reduced set of keys to be presented to the user for selection of PKs. The code analyzer then searches for these patterns in the application code and rules out selected candidate patterns. At the end of this step, all the primary keys are finalized.

Result. In the example DRE application, the following relations and their attributes were obtained from the legacy database:

- **Proj** `[P_ID, P_NAME, DES_S, DES_F, A_S, A_F, ...]`
- **Avail** `[PROJ_ID, AVAIL UID, Res_ID, AVAIL FROM, AVAIL TO, ...]`
- **Res** `[PROJ_ID, RES UID, RES NAME, R ACWP, R BCWP, R BCWS, ...]`
- **T** `[PROJ_ID, T UID, T ID, T NAME, T PRITY, T DUR, T ST D, T FIN D, ...]`
- **Assn** `[PROJ_ID, ASSN UID, T UID, R ID, ASSN_BASE_C, ...]`


As described above and shown in Figure 5, code analysis is conducted in five substeps. The first substep is the pre-slicing. From the Abstract Syntax Tree (AST) of the application code, the pre-slicer identifies all the nodes corresponding to input, output and embedded SQL statements. It appends the statement node name, and identifier list to an array as the AST is traversed in pre-order. For example, for the AST in Figure 7, 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 the pre-slicer

<table>
<thead>
<tr>
<th>Node number</th>
<th>Statement</th>
<th>Text String (only for print nodes)</th>
<th>Identifiers</th>
<th>Direction of Slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>embSQL</td>
<td>------</td>
<td>aValue, cValue</td>
<td>Backwards</td>
</tr>
</tbody>
</table>
The code slicer and analyzer, which comprise Substeps 2 and 3 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) node that corresponds to the slicing variable.

During the code slicing sub-step we traverse the entire AST for the source code and retain only those nodes that have an occurrence of the slicing variable in the corresponding subtree. The output of this substep is a reduced AST, which is produced as follows:

1. Locate the appropriate node on the tree where the slicing should begin/end. For forward slicing, we start at a certain node (identified by the slicing criteria) and walk the tree nodes in pre-order till the end of the tree. For backward slicing, we start at the root node and walk the tree in pre-order stopping when we arrive at the node corresponding to the slicing criteria.

2. Do a pre-order traversal of the entire AST starting at the root. Each time a declaration (dcln), statement, or assign node is encountered, search the sub-tree for occurrences of the slicing variable that we are looking for. If present, push the sub-tree with root dcln or statement or assign node onto a stack.

3. If there is an assign statement with the slicing variable on the right hand side, add the left hand side variable (identifier name) also to the list of slicing variables.

4. Repeat Steps 2 and 3 until the appropriate end node is reached (as defined in Step 1).

5. Pop out all the elements from the stack and build a tree of them using the left child-right sibling representation of N-ary trees.

For the preceding example, the reduced AST is depicted in Figure 8.

```
Figure 8: Reduced AST.
```

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

1. If a dcln node is encountered, the data type of the identifier can be learned.

2. embSQL contain the mapping information of identifier name to corresponding column name and table name in the database.

3. Printf/scanf nodes contain the mapping information from the text string to the identifier. In other words we can extract the contextual meaning of the identifier from the text string.
Table 2: Information inferred during the analysis sub-step.

<table>
<thead>
<tr>
<th>Identifier Name</th>
<th>Meaning</th>
<th>Possible Business Rule</th>
<th>Data type</th>
<th>Corresponding Column Name in Database</th>
<th>Corresponding Table Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>aValue</td>
<td>Task Start Date</td>
<td>if (cValue &lt; aValue)</td>
<td>Char * =&gt; string</td>
<td>T_ST_D</td>
<td>T</td>
</tr>
<tr>
<td>cValue</td>
<td>Task Finish Date</td>
<td>if (cValue &lt; aValue)</td>
<td>Char * =&gt; string</td>
<td>T_FIN_D</td>
<td>T</td>
</tr>
</tbody>
</table>

The results of the analysis sub-step are appended to a report file. After the code slicer and analyzer have been invoked on every slicing variable identified by the pre-slicer, the report file is presented to the user at the end of the DRE phase.

The analysis program provides the user with intermediate output, which is used to validate the results. For example, during code analysis, output about the extracted business rules is available. A partial list of business rules extracted from the sample code presented earlier is shown in Figure 9.

![Schema Extraction Output](image)

**Figure 9**: Sample business rule output after SA is complete.

Using output screens such as the one in Figure 9, the user can decide whether or not to perform further analysis. This is exemplified in Figure 10. In this screen, the user can enter a new slicing variable and display the slicing variables used in previous passes as well as identifiers that occur in input, output and SQL statements. The code slicer and analyzer are invoked based on the slicing variable(s) indicated by the user. If the user decides not to perform further analysis, code analysis passes control to the inclusion dependency detection module.
Step 4. Discovering Inclusion Dependencies.

If foreign keys are available, we determine associated inclusion dependencies by examining the relation(s) to which they belong or refer. Control is then transferred to the equi-join detection module to determine if equi-join queries are embedded in the application, via pattern matching applied to FROM and WHERE clauses. Detected equi-join queries are extracted and returned to the inclusion dependency finder, where they support confirmation or augmentation of the existing set of inclusion dependencies. The inclusion dependency finder then tests possible inclusion dependencies between still un-related pairs of relations in an exhaustive manner as described in substeps 4 and 5 of the inclusion dependency finder algorithm. After this exhaustive search, a final set of inclusion dependencies is validated with the help of a domain expert.

Result: Inclusion dependencies are as follows:

\[
\begin{align*}
\text{Assn}[T\_UID,\text{Proj}\_ID] & \ll T[T\_UID,\text{Proj}\_ID] \\
\text{Assn}[\text{Res}\_uid,\text{Proj}\_ID] & \ll \text{Res}[\text{Res}\_uid,\text{Proj}\_ID] \\
\text{Avail}[\text{Res}\_UID,\text{Proj}\_ID] & \ll \text{Res}[\text{Res}\_UID,\text{Proj}\_ID] \\
\text{Res}[\text{Proj}\_ID] & \ll \text{Proj}[\text{P}\_ID] \\
T[\text{Proj}\_ID] & \ll \text{Proj}[\text{P}\_ID] \\
\text{Assn}[\text{Proj}\_ID] & \ll \text{Proj}[\text{P}\_ID] \\
\text{Avail}[\text{Proj}\_ID] & \ll \text{Proj}[\text{P}\_ID]
\end{align*}
\]

If a minimal set is desired, for example, the second to last inclusion dependency can be removed since it is implicitly contained in the second and fourth inclusion dependencies using the transitivity relationship.

Step 5. Classification of the Relations.

As discussed earlier in the paper, in this step all the relations in the database are classified into one of four types – strong, regular, weak or specific. This classification is done using primary key information obtained in step 2 and inclusion dependencies from step 4. We use the test described earlier by which we check for presence of primary key attributes of one relation in the primary key of other relations. For example, in relation \( T \), \( T\_UID \) is not a primary key of any other relation; on the
other hand, \texttt{PROJ\_ID} is a primary key of \texttt{PROJ} (where it is called \texttt{P\_ID}). Hence, \texttt{T} is a weak entity relation and \texttt{PROJ} is a strong entity relation. Inclusion dependencies involving primary key attributes of two relations on both sides also give similar clues.

\textbf{Result:}

\textit{Strong Entities:} \texttt{PROJ, AVAIL}

\textit{Weak Entities:} \texttt{RES, T}

\textit{Regular Relationship:} \texttt{ASSN}

\textbf{Step 6. Classification of the Attributes.}

We classify attributes as (a) PK or FK, (b) Dangling or General, or (c) Non-Key (rest).

\textbf{Result:} Table 3 illustrates attributes obtained from the example legacy source.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
 & PKA & DKA & FKA & All Remaining Attributes \\
\hline
PROJ & P\_ID &  &  &  \\
RES & Proj\_ID + Res\_UID & Res\_UID &  &  \\
T & Proj\_ID + T\_UID & T\_UID &  &  \\
AVAIL & Proj\_ID + Avail\_UID & Avail\_UID & Res\_UID + Proj\_ID &  \\
ASSN & Proj\_ID + Assn\_UID & Assn\_UID & R\_UID + Proj\_ID, T\_UID + Proj\_ID &  \\
\hline
\end{tabular}
\caption{Example of the attribute classification from the MS-Project legacy source.}
\end{table}

\textbf{Step 7. Identify Entity Types.}

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

\textbf{Result:} The following entities were classified:

\textit{Strong entities:} \texttt{PROJ} with \texttt{P\_ID} as its key.

\texttt{AVAIL} with \texttt{AVAIL\_UID} as its key

\textit{Weak entities:} \texttt{T} with \texttt{T\_UID} as key and \texttt{PROJ} as its owner.

\texttt{RES} with \texttt{Res\_UID} as key and \texttt{PROJ} as its owner.

\textbf{Step 8. Identify Relationship Types.}

No decision or analysis is required.

\textbf{Result:}

The following 1:N binary relationships between the following weak entity types were discovered:

Between \texttt{PROJ} and \texttt{T} and between \texttt{PROJ} and \texttt{RES}

In addition, a 1:N regular binary relationship was discovered between following entities:

Between \texttt{RES} and \texttt{AVAIL}

Since two inclusion dependencies involving \texttt{ASSN} exist (i.e., between \texttt{T} and \texttt{ASSN} and between \texttt{RES} and \texttt{ASSN}), there is no need to define a new entity. Thus, \texttt{ASSN} becomes an M:N relationship between \texttt{T} and \texttt{RES}.

At the end of the DRE process, the user has the option to review the extracted knowledge for correctness. If modifications are necessary, the extraction and analysis process can be repeated. Some of the screen snapshots providing system output are shown in Figures 11 and 12.
Figure 11: DRE screen snapshot displaying partial list of extracted entities and their metadata.

Figure 12: DRE screen snapshot displaying partial list of extracted relationships and their metadata.
A conceptual overview of the schema is represented by the entity-relationship diagram\(^1\) shown in Figure 13.

Figure 13: E/R diagram representing the extracted schema.

4 Related Research

Research related to this work includes data reverse engineering and program analysis. The state-of-the-art is summarized in the following paragraphs.

Data reverse engineering (DRE) refers to the inference of structure and meaning (e.g., schema, relations, and semantics) from databases [7]. The DRE literature is divided into three areas: translation algorithms and methodologies, tools, and application-specific experiences. Translation algorithm development in early DRE efforts involved manual rearrangement or reformatting of data fields, which is inefficient and error-prone [7]. The relational data model provided theoretical support for research in automated discovery of relational dependencies [3]. In the early 1980s, focus shifted to translating relations to E/R diagrams [18]. Given early successes with translation using the relational data model, DRE translation was applied to flat file databases [3, 8] in domains such as enterprise schemas [16].

Due to previous establishment of the E/R model as a conceptual tool, reengineering of legacy RDBMS to yield E/R models motivated DRE in the late 1980s [9]. Information content analysis was also applied to RDBMS, allowing more effective gathering of high-level information from data [2].

DRE in the 1990s was enhanced by cross-fertilization with software engineering. In [6] a taxonomy for reverse engineering included DRE methodologies and also highlighted available DRE tools. DRE formalisms were better defined, motivating increased DRE interaction with users [12]. The relational data model continued to support extraction of E/R and schema from RDBMS [5, 17]. Application focus emphasized legacy systems, including DoD applications [1].

\(^1\) The XML representation of the extracted knowledge in its entirety can be downloaded from http://www.dbcenter.cise.ufl.edu/.
In the late 1990s, object-oriented DRE researched the discovering of objects in legacy systems using function-, data-, and object-driven objectification [27]. Applications of DRE increased, particularly in Y2K bug identification and remediation. Recent DRE focus is more applicative, e.g., mining large data repositories [10], analysis of legacy systems [14] or network databases [20], and extraction of business rules from legacy systems [25]. Current research focuses on developing more powerful DRE tools, refining heuristics to yield fewer missing constructs, and developing techniques for reengineering legacy systems into distributed applications.

An important trend in knowledge discovery research is program analysis or program comprehension. Originally, programmers sought to understand program functionality and impact of proposed program modifications, leading to a close association with reverse engineering. SEEK uses knowledge extraction to discover enterprise knowledge represented in application code. Several automated approaches have been proposed for program comprehension. The most important techniques include program slicing [15], cliché recognition [28], and pattern matching [22], together with the more conventional approaches of lexical/syntactic analysis. Slicing identifies program statements that directly affect variables of interest to the knowledge extraction process (e.g., input and output variables and their dependents).

Slicing, a popular data flow analysis derivative, when coupled with program dependence analysis and a language independent program representation, can extract semantics from application code. A program dependence graph (PDG) is a DAG whose vertices are an assignment statement or a predicate of an if-then-else or while construct. Different edges represent control and data flow dependences. Control flow edges, labeled true or false, respectively lead to an entry point of then or else block statements. During program execution, if the value of the edge matches the predicate, then the node pointed to by the control flow edge represents the next executable statement.

Pattern matching identifies interesting code patterns and their dependencies. For example, conditional control structures such as if..then..else, or case statements may encode business rules, whereas type declarations and class/structure definitions can provide information about names, data types, and structure of concepts represented in a legacy source. Pattern matching is implemented by transforming source code and templates constructed from preselected patterns into syntax trees. Coupled with analysis of program dependency graphs, a language independent program representation, slicing, and cliché recognition, pattern matching is a valuable tool for extracting semantic information from application code.

5 Summary and Discussion

We have manually tested a database reverse engineering approach for a number of scenarios and domains (including construction, manufacturing and health care) to validate our knowledge extraction algorithm and to estimate how much user input is required. In addition, we have conducted experiments using nine different database applications that were created by graduate students during course projects. The experimental results so far are encouraging: the DRE algorithm was able to reverse engineer all of the sample legacy sources encountered so far. When coupled with semantic analysis, human input is reduced compared to existing methods. The approach presented herein provides the user with clues and guidelines that lead to the augmentation of the schema with additional semantic knowledge.

The SEEK prototype is being extended using sample data from a large building construction project on the University of Florida campus, in cooperation with the construction manager (Centex Rooney Inc.) and several subcontractors or suppliers. This data testbed will support more rigorous testing of the SEEK toolkit. Other plans for the SEEK toolkit are:

- Develop a formal representation for the extracted knowledge.
• Develop a matching tool capable of producing mappings between two semantically related yet structurally different schemas. Currently, schema matching is performed manually, which is a tedious, error-prone, and expensive process.

• Integrate SEEK with a wrapper development toolkit to determine if the extracted knowledge is sufficiently rich semantically to support compilation of legacy source wrappers for our construction testbed.

We plan to extend the semantic matching tool to provide machine assistance to domain experts for developing cross-application ontologies that are expected to support further automation of the DRE process. The eventual system concept is that of a large, nearly-automatic system that can (1) acquire large amounts of knowledge from multiple legacy systems, (2) extend and enhance its on-board knowledge representation and characterization capabilities through ontology-based learning, and (3) thus make each successive acquisition of knowledge from a legacy system easier and more accessible to the SEEK user community.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant numbers CMS-0075407 and CMS-0122193.

References


Appendix A. Grammar for C code used in Semantic Analysis Algorithm

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Production</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CProgram</td>
<td>Consts Forwards Dclns</td>
<td>&quot;program&quot;;</td>
</tr>
<tr>
<td>Includes</td>
<td>(&quot;#include&quot; &lt;filename&gt;)*</td>
<td>&quot;include&quot;;</td>
</tr>
<tr>
<td>Consts</td>
<td>(Const ';')*</td>
<td>&quot;consts&quot;</td>
</tr>
<tr>
<td>Const</td>
<td>'#define' Name</td>
<td>&quot;const&quot;</td>
</tr>
<tr>
<td>Forwards</td>
<td>(Forward ';')*</td>
<td>&quot;forwards&quot;</td>
</tr>
<tr>
<td>Forward</td>
<td><code>^</code> Type Name Params</td>
<td>&quot;forward&quot;</td>
</tr>
<tr>
<td>Dclns</td>
<td>(DclnList ';')*</td>
<td>&quot;dclns&quot;</td>
</tr>
<tr>
<td>Type</td>
<td>Id;</td>
<td>&quot;dcln&quot;</td>
</tr>
<tr>
<td>DclnList</td>
<td>Type Dcln list ','</td>
<td>&quot;structdcln&quot;</td>
</tr>
<tr>
<td>Dcln</td>
<td>Id '=' Expression</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>Function</td>
<td>Type Name Params '{' Dclns Statement+ '}'</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>Params</td>
<td>'(' DclnList ? ')'</td>
<td>&quot;params&quot;</td>
</tr>
<tr>
<td>Block</td>
<td>'(' Statement+ ')',</td>
<td>&quot;block&quot;</td>
</tr>
<tr>
<td>Statement</td>
<td>Assignment ';',</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td></td>
<td>'printf' '(' Expression list ',')',</td>
<td>&quot;printf&quot;</td>
</tr>
<tr>
<td></td>
<td>'printf' '(' String? Expression list ',')',</td>
<td>&quot;emptyprint&quot;</td>
</tr>
<tr>
<td></td>
<td>'scanf' '(' String? Id list ',')',</td>
<td>&quot;scanf&quot;</td>
</tr>
<tr>
<td></td>
<td>'if' '(' Expression ')', ('else' Statement)?</td>
<td>&quot;if&quot;</td>
</tr>
<tr>
<td></td>
<td>'while' '(' Expression ')',</td>
<td>&quot;while&quot;</td>
</tr>
<tr>
<td></td>
<td>'for' '(' Assignment ';', Expression ';', Assignment ')',</td>
<td>&quot;for&quot;</td>
</tr>
<tr>
<td></td>
<td>'for' '(' ';', ';', ')',</td>
<td>&quot;for&quot;</td>
</tr>
<tr>
<td></td>
<td>'do' Statement 'while' Expression ';',</td>
<td>&quot;do&quot;</td>
</tr>
<tr>
<td></td>
<td>'switch' '(' Term ')', '{' Case+ 'default' ':' Block ')',</td>
<td>&quot;switch&quot;</td>
</tr>
<tr>
<td></td>
<td>SQLprefix SQLstatement SQLterminator?</td>
<td>&quot;embSQL&quot;</td>
</tr>
<tr>
<td></td>
<td>(DclnList ';')*</td>
<td>&quot;dclns&quot;</td>
</tr>
<tr>
<td></td>
<td>Primary '++'</td>
<td>&quot;++&quot;</td>
</tr>
<tr>
<td></td>
<td>Primary '--'</td>
<td>&quot;--&quot;</td>
</tr>
<tr>
<td>SQLprefix</td>
<td>EXEC SQL DBclause?</td>
<td>=&gt; &quot;beginSQL&quot;</td>
</tr>
<tr>
<td>SQLterminator</td>
<td>END-EXEC</td>
<td>=&gt; &quot;endSQL&quot;</td>
</tr>
<tr>
<td>SQLstatement</td>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist</td>
<td>=&gt; 'SQLselectone'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectone'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' 'EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselectone'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselectone'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'COUNT' '(' '*' ')' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectonecount'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'DISTINCT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectonedistinct'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod</td>
<td>=&gt; 'SQLselecttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselecttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselecttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselecttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'COUNT' '(' '*' ')' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselecttwocount'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'DISTINCT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselecttwodistinct'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'GROUP' 'BY' columnlistgroupby</td>
<td>=&gt; 'SQLselecttwogroupby'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod</td>
<td>=&gt; 'SQLselectttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselectttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselectttwo'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'COUNT' '(' '*' ')' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectttwocount'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' 'DISTINCT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselectttwodistinct'</td>
</tr>
<tr>
<td></td>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'GROUP' 'BY' columnlistgroupby</td>
<td>=&gt; 'SQLselectttwogroupby'</td>
</tr>
<tr>
<td></td>
<td>SQL syntax</td>
<td>Natural representation</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td></td>
<td><code>SELECT</code> columnlist <code>FROM</code> tablelistmod <code>WHERE</code> SQLExpression <code>ORDER</code> <code>BY</code> columnlistgroupby</td>
<td>➞ <code>'SQLselecttwogroupby'</code></td>
</tr>
<tr>
<td></td>
<td><code>INSERT</code> INTO tablelist <code>VALUES</code> <code>(hostvariablelist)</code></td>
<td>➞ <code>'SQLinsert'</code></td>
</tr>
<tr>
<td></td>
<td><code>DELETE</code> Id <code>FROM</code> tablelist <code>WHERE</code> SQLExpression</td>
<td>➞ <code>'SQLdelete'</code></td>
</tr>
<tr>
<td></td>
<td><code>UPDATE</code> tablelist <code>SET</code> (SQLAssignment <code>,</code>) list <code>WHERE</code> SQLExpression</td>
<td>➞ <code>'SQLupdate'</code></td>
</tr>
<tr>
<td></td>
<td>;</td>
<td>➞ <code>'SQLselect'</code></td>
</tr>
<tr>
<td>tablelist</td>
<td>( Name list <code>,</code>)</td>
<td>➞ 'tablelist'</td>
</tr>
<tr>
<td>tablelistmod</td>
<td>( tablename list <code>,</code>)</td>
<td>➞ 'tablelist'</td>
</tr>
<tr>
<td>tablename</td>
<td>Id Id</td>
<td>➞ 'tablename'</td>
</tr>
<tr>
<td>columnlist</td>
<td>( Term list <code>,</code>)</td>
<td>➞ 'columnlist'</td>
</tr>
<tr>
<td>columnlistgroupby</td>
<td>( Name list <code>,</code>)</td>
<td>➞ 'columnlistgroupby'</td>
</tr>
<tr>
<td>Hostvariablelist</td>
<td>(Variable list <code>,</code>)</td>
<td>➞ 'hostvariablelist'</td>
</tr>
<tr>
<td></td>
<td>;</td>
<td>➞ 'hostvariablelist'</td>
</tr>
<tr>
<td>Variable</td>
<td>`:=' Name ;</td>
<td></td>
</tr>
<tr>
<td>SQLExpression</td>
<td>SQLExpression <code>AND</code> SQLAssignment</td>
<td>➞ <code>'SQLExpression'</code></td>
</tr>
<tr>
<td></td>
<td>SQLExpression <code>OR</code> SQLAssignment;</td>
<td>➞ <code>'SQLExpression'</code></td>
</tr>
<tr>
<td>SQLAssignment</td>
<td>Id `=' Name</td>
<td>➞ <code>'SQLAssignment='</code></td>
</tr>
<tr>
<td></td>
<td>Id `&gt;' Name</td>
<td>➞ <code>'SQLAssignment&gt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id `&lt;' Name</td>
<td>➞ <code>'SQLAssignment&lt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id `='= Name</td>
<td>➞ <code>'SQLAssignment&gt;=</code></td>
</tr>
<tr>
<td></td>
<td>Id `&lt;=' Name</td>
<td>➞ <code>'SQLAssignment&lt;='</code></td>
</tr>
<tr>
<td></td>
<td>Id `&lt;&gt;' Name</td>
<td>➞ <code>'SQLAssignment&lt;&gt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id <code>='= Name </code>(' (Expression list <code>,')? ')' </code>)' ;</td>
<td>➞ <code>'SQLAssignment='</code></td>
</tr>
<tr>
<td></td>
<td>Id <code>&gt;'= Name </code>(' (Expression list <code>,')? ')' </code>)' ;</td>
<td>➞ <code>'SQLAssignment&gt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id <code>&lt;='= Name </code>(' (Expression list <code>,')? ')' </code>)' ;</td>
<td>➞ <code>'SQLAssignment&lt;='</code></td>
</tr>
<tr>
<td></td>
<td>Id <code>&lt;&gt;'= Name </code>(' (Expression list <code>,')? ')' </code>)' ;</td>
<td>➞ <code>'SQLAssignment&lt;&gt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id <code>LIKE</code> String</td>
<td>➞ <code>'SQLAssignmentLIKE'</code></td>
</tr>
<tr>
<td></td>
<td>Id `=' SQLStatement</td>
<td>➞ <code>'SQLAssignment='</code></td>
</tr>
<tr>
<td></td>
<td>Id `=' ANY SQLStatement</td>
<td>➞ <code>'SQLAssignment='</code></td>
</tr>
<tr>
<td></td>
<td>Id `&gt;' ANY SQLStatement</td>
<td>➞ <code>'SQLAssignment&gt;'</code></td>
</tr>
<tr>
<td></td>
<td>Id `&lt;=' ANY SQLStatement</td>
<td>➞ <code>'SQLAssignment&lt;='</code></td>
</tr>
<tr>
<td></td>
<td>Id `&lt;&gt;' ANY SQLStatement</td>
<td>➞ <code>'SQLAssignment&lt;&gt;'</code></td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' 'ANY' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '=' 'ALL' SQLStatement</code></td>
<td>&quot;SQLAssignment=&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&gt;' 'ALL' SQLStatement</code></td>
<td>&quot;SQLAssignment&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;' 'ALL' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;=' 'ALL' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;=&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '=&gt;' 'ALL' SQLStatement</code></td>
<td>&quot;SQLAssignment=&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '=' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment=&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&gt;' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;=' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;=&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '=&gt;' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment=&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' 'IN' SQLStatement</code></td>
<td>&quot;SQLAssignment&lt;&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>DB clause</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>BEGIN' 'DECLARE' 'SECTION'</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>END' 'DECLARE' 'SECTION'</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'WARNING' 'CALL' Name '(' (Expression list ',')? ')'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'FOUND' 'CALL' Name '(' (Expression list ',')? ')'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'FOUND' 'DO' 'BREAK'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'FOUND' 'DO' 'CONTINUE'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'ERROR' 'CALL' Name '(' (Expression list ',')? ')'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>DISCONNECT' 'ALL'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>USE' Id</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>CONNECT' Name 'IDENTIFIED' 'BY' Name</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>COMMIT'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>COMMIT' 'WORK' 'RELEASE'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>COMMIT' 'WORK'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>OPEN' Name</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>CLOSE' Name</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>DECLARE' Name 'FOR'</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td><code>FETCH' Name 'INTO' hostvariablelist</code></td>
<td>&quot;DBclause&quot;</td>
<td></td>
</tr>
<tr>
<td>Case</td>
<td>&quot;case&quot;</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>&quot;assign&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '=' Expression</code></td>
<td>&quot;assign&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '+'=' Expression</code></td>
<td>&quot;assign&quot;</td>
<td></td>
</tr>
<tr>
<td><code>Id '-'=' Expression</code></td>
<td>&quot;assign&quot;</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>&quot;?&quot;</td>
<td></td>
</tr>
<tr>
<td><code>LEExpression '?' LEExpression</code></td>
<td>&quot;?&quot;</td>
<td></td>
</tr>
<tr>
<td><code>LEExpression</code></td>
<td>&quot;and&quot;</td>
<td></td>
</tr>
<tr>
<td>`LEExpression '</td>
<td></td>
<td>' Comparison`</td>
</tr>
</tbody>
</table>

27
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>LExpression '*' Comparison</td>
<td>=&gt; &quot;xor&quot;</td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>Term '&lt;=' Term</td>
<td>=&gt; &quot;&lt;=&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '==' Term</td>
<td>=&gt; &quot;==&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '&gt;=' Term</td>
<td>=&gt; &quot;&gt;=&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '!=' Term</td>
<td>=&gt; &quot;!=&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '&lt;' Term</td>
<td>=&gt; &quot;&lt;&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '&gt;' Term</td>
<td>=&gt; &quot;&gt;&quot;</td>
</tr>
<tr>
<td>Term</td>
<td>Term '+' Factor</td>
<td>=&gt; &quot;+&quot;</td>
</tr>
<tr>
<td></td>
<td>Term '-' Factor</td>
<td>=&gt; &quot;-&quot;</td>
</tr>
<tr>
<td>Factor</td>
<td>Exp '*' Factor</td>
<td>=&gt; &quot;*&quot;</td>
</tr>
<tr>
<td></td>
<td>Exp '/' Factor</td>
<td>=&gt; &quot;/&quot;</td>
</tr>
<tr>
<td></td>
<td>Exp '%' Factor</td>
<td>=&gt; &quot;%&quot;</td>
</tr>
<tr>
<td></td>
<td>Exp ;</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td>Exp</td>
<td>Primary '**' Exp</td>
<td>=&gt; &quot;***&quot;</td>
</tr>
<tr>
<td></td>
<td>Primary ;</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td>Primary</td>
<td>'- Primary</td>
<td>=&gt; &quot;-_&quot;</td>
</tr>
<tr>
<td></td>
<td>'+' Primary</td>
<td>=&gt; &quot;_+&quot;</td>
</tr>
<tr>
<td></td>
<td>'!' Primary</td>
<td>=&gt; &quot;!&quot;</td>
</tr>
<tr>
<td></td>
<td>'++' Primary</td>
<td>=&gt; &quot;_++&quot;</td>
</tr>
<tr>
<td></td>
<td>'--' Primary</td>
<td>=&gt; &quot;-_+&quot;</td>
</tr>
<tr>
<td></td>
<td>Primary '++'</td>
<td>=&gt; &quot;_++&quot;</td>
</tr>
<tr>
<td></td>
<td>Primary '--'</td>
<td>=&gt; &quot;-_+&quot;</td>
</tr>
<tr>
<td>Atom</td>
<td>'eof'</td>
<td>=&gt; &quot;eof&quot;</td>
</tr>
<tr>
<td></td>
<td>'&lt;integer&gt;'</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>Id</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>'(' Expression ')';</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>Name '(', (Expression list =&gt; &quot;rhscall&quot; &quot;,&quot;,? ')');</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td>Initializer</td>
<td>'&lt;integer&gt;'</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>'&amp;' Name</td>
<td>=&gt; &quot;&amp;&quot;</td>
</tr>
<tr>
<td>Id</td>
<td>'*' Name</td>
<td>=&gt; &quot;*&quot;</td>
</tr>
<tr>
<td></td>
<td>'&amp;' Name</td>
<td>=&gt; &quot;&amp;&quot;</td>
</tr>
<tr>
<td></td>
<td>Name;</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td>Name</td>
<td>'&lt;identifier&gt;'</td>
<td>=&gt; &quot;&quot;</td>
</tr>
<tr>
<td>String</td>
<td>'&lt;text&gt;'</td>
<td>=&gt; &quot;&quot;</td>
</tr>
</tbody>
</table>
Appendix B. Subset Test for Inclusion Dependency Detection.

We are using the following subset test to determine whether there exists an inclusion dependency between attribute (or attribute set) U of relation R1 and attribute (or attribute set) V of R2. Note, U and V must have the same data type and must includes the same number of attributes. Our test is based on the following SQL query templates, which are instantiated for the relations and their attributes and are run against the legacy source.

\[
\begin{align*}
C1 & = \\
& \text{SELECT count (*)} \\
& \text{FROM } R1 \\
& \text{WHERE } U \text{ NOT IN} \\
& \quad \left( \text{SELECT V} \\
& \quad \text{FROM } R2 \right); \\

C2 & = \\
& \text{SELECT count (*)} \\
& \text{FROM } R2 \\
& \text{WHERE } V \text{ NOT IN} \\
& \quad \left( \text{SELECT U} \\
& \quad \text{FROM } R1 \right); \\
\end{align*}
\]

If C1 is zero, we can deduce that there may exist an inclusion dependency R1.U << R2.V; likewise, if C2 is zero there may exist an inclusion dependency R2.V << R1.U. 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 U and V are equal.
Appendix C. DTD Describing Extracted Knowledge.

<?xml version="1.0"?>
<!DOCTYPE Model [
  <!ELEMENT Database (Relational_Schema, Conceptual_Schema,
   Business_Rules)>]
  <!ELEMENT Relational_Schema (Relation+)>  
  <!ELEMENT Relation (Name, Type, Columns)>  
  <!ELEMENT Name   (#PCDATA)>  
  <!ELEMENT Type   (#PCDATA)>  
  <!ELEMENT Columns (Column+)>  
  <!ELEMENT Column (Name, DataType, Meaning, IsPK, FK,
   NullOption, IsUnique)>  
  <!ELEMENT Name   (#PCDATA)>  
  <!ELEMENT DataType (#PCDATA)>  
  <!ELEMENT Meaning (#PCDATA)>  
  <!ELEMENT IsPK   (#PCDATA)>  
  <!ELEMENT FK    (IsFK, FKTable)>  
  <!ELEMENT IsFK   (#PCDATA)>  
  <!ELEMENT FKTable (#PCDATA)>  
  <!ELEMENT NullOption (#PCDATA)>  
  <!ELEMENT IsUnique (#PCDATA)>  
  <!ELEMENT Conceptual_Schema (Entity+, Relationship+)>  
  <!ELEMENT Entity (Name, Type, Identifier)>  
  <!ELEMENT Name   (#PCDATA)>  
  <!ELEMENT Type   (#PCDATA)>  
  <!ELEMENT Identifier (#PCDATA)>  
  <!ELEMENT Relationship (Name, Type, Participants)>  
  <!ELEMENT Name   (#PCDATA)>  
  <!ELEMENT Type   (#PCDATA)>  
  <!ELEMENT Participants (Participant+)>  
  <!ELEMENT Participant (ParticipantEntity, Cardinality)>  
  <!ELEMENT ParticipantEntity (#PCDATA)>  
  <!ELEMENT Cardinality (#PCDATA)>  
  <!ELEMENT Business_Rules (Rule+)>  
  <!ELEMENT Rule   (#PCDATA)>  
]>

30