Res. Lett. Inf. Math. Sci. (2003) **4**, 113-136
Available online at http://iims.massey.ac.nz /research/letters/

Automated Schema Matching Techniques: An Exploratory Study

Xiao Long Sun
Institute of Information and Mathematical Sciences
Massey University, Auckland, New Zealand
xiaosun@xtra.co.nz

Ellen Rose
Institute of Information and Mathematical Sciences
Massey University, Auckland, New Zealand
e.a.rose@massey.ac.nz

Abstract.

Manual schema matching is a problem for many database applications that use multiple data sources including data warehousing and e-commerce applications. Current research attempts to address this problem by developing algorithms to automate aspects of the schema-matching task. In this paper, an approach using an external dictionary facilitates automated discovery of the semantic meaning of database schema terms. An experimental study was conducted to evaluate the performance and accuracy of five schema-matching techniques with the proposed approach, called SemMA. The proposed approach and results are compared with two existing semi-automated schema-matching approaches and suggestions for future research are made.

1 Introduction

The rapid development of computer information systems over the past three decades has resulted in a number of heterogeneous data sources. Database application domains such as data warehousing (Stohr et al., 1999), data integration (Bergamaschi et al., 1999), e-commerce and semantic query processing (Heflin, 2001) all rely on schema matching to achieve interoperability (Rahm and Bernstein, 2001a). These new database applications demand integration of independently developed data sources, making interoperability increasingly important (Heiler, 1995). Heterogeneity can be classified into two main types: 1) information heterogeneity and 2) system heterogeneity. This classification of concerns can be further expanded as shown in table 1 Ouksel and Sheth (1999).

Heterogeneity Problems		**Interoperability Concerns**
Information Heterogeneity	Semantic heterogeneity	Semantic interoperability
	Structural heterogeneity	Structural interoperability
	Syntactic, format heterogeneity	Syntactic interoperability
System Heterogeneity	Information system heterogeneity: 1) Digital media management system (unstructured, semi-structured data) 2) Database management systems (structured data)	System Interoperability
	Platform heterogeneity: 1) Operating systems Hardware/system	

Table 1. Heterogeneity Problems and Corresponding Interoperability Concerns.

Currently, schema matching to solve these heterogeneity problems is typically performed manually. This manual process is tedious, time consuming, error-prone and expensive. A less labour-intensive approach is desirable. Such an approach would provide automated support to identify relationships and generate

mappings between a source schema and a target schema. A classification of existing approaches is summarized in figure 1.

Most work on automated schema matching has been done during the past decade. Much of this work was primarily focused in the context of a particular application domain and applied to a particular schema format. For example, SemInt (relational schema) (Li and Clifton, 1994), LSD (XML schema) (Doan et al., 2001) and SKAT (XML schema) (Mitra et al., 1999) were used in the data integration domain. ARTEMIS (relational schema) (Castano et al., 2001) and DIKE (ER model) (Palopoli et al., 1999) were used for schema integration. Cupid (XML) (Madhavan et al., 2001) and TranScm (SGML object-oriented class schema) (Milo and Zohar, 1998) were used for data translation tasks as found in the e-commerce domain. Success in schema matching depends on understanding the semantics of the schema elements, such as attributes, relations, entity set, etc., and the ability to reason with these semantics (Rahm and Bernstein, 2001b).

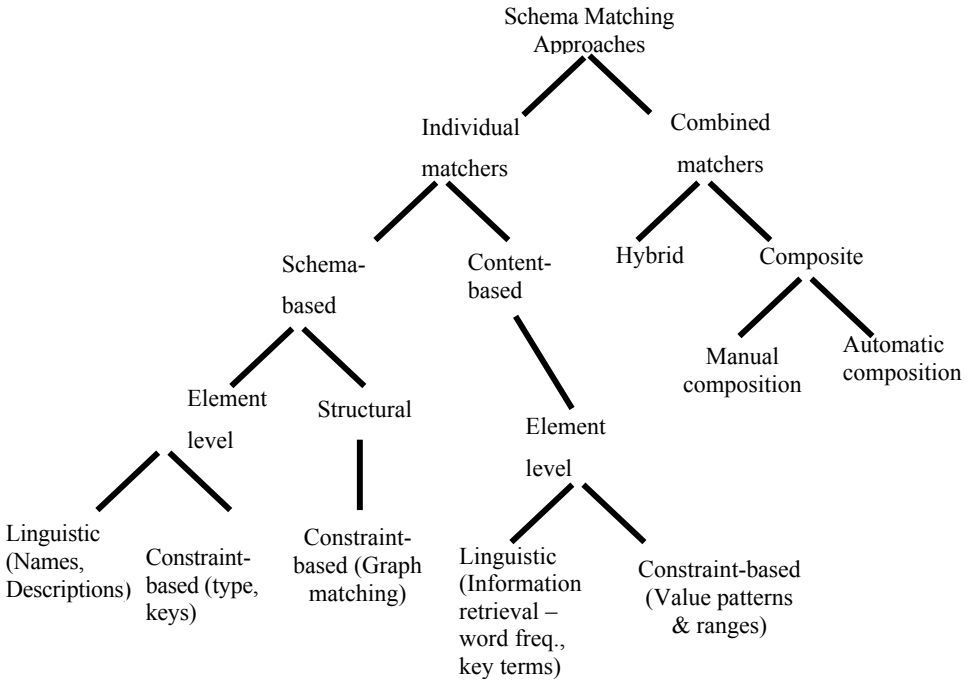


Figure 1: Classification of Existing Schema Matching Approaches.

While significant progress has been achieved in system, syntactic, and structural interoperability, comprehensive semantic interoperability still remains as an open problem (Goh 1997)(Hull, 1997). To achieve cost effective, efficient semantic interoperability, semi-automated schema matching mechanisms are needed.

This research analysed current semi-automated schema matching approaches used in relational data warehouse environments. The core research question is: *What are the critical criteria of schema matching that lead to improved semantic matching between two relational data schemas?* The study examined the relationship between the independent variables that is the matching criterion of relational schemas and the measurable dependent variables of completeness, precision and overall matching quality. To answer these questions, the relative performance and precision of a set of five schema-matching algorithms using two relational source schemas and one target data warehouse schema were examined. A schema matcher was designed and implemented in C++ and is referred to here as SemMa (Semantic Matcher).

The contributions of this study are: 1) using an external dictionary to construct an internal thesaurus to detect the semantics of database terms, 2) using an experimental evaluation of schema matching criteria, and 3) using object-oriented techniques to simplify the schema matching process. The following objectives were set:

- Measure the effects of schema matching criteria such as name, type and structure matching on the performance of SemMa.
- Evaluate the impact on matching performance of using an external dictionary WordNet (WordNet, 2002).
- Compare the SemMa approach to existing approaches to identify areas that need improvement

Section 2 of the paper discusses the design and implementation of the Semantic Matcher, SemMa. Section 3 discusses the experiments conducted to validate the performance of SemMa. Section 4 presents and analyses the results of executing SemMa on a sample test data set. Section 5 concludes the paper with a summary of the contributions of the study and suggestions for further improvements to the semantic matching operation.

2 The Proposed Semantic Match Approach

2.1 Design of the Semantic Matcher

In this project, an approach to semantic-matching called (SemMa) was designed and implemented. SemMa constructs an internal database thesaurus and uses an external dictionary as in (Lawrence and Barker, 2001) to discover the semantic meaning of database schema terms. Access to the semantics (i.e. meaning) of database terms allows the computer system to automatically create a mapping between fields of the target database schema and fields of the source database schema that are semantically equivalent, thereby providing automated support for the matching task. As in other schema matching approaches such as SemInt (Lee and Clifton, 1994), ARTEMIS (Castano, et al. 2001) and Cupid (Madhavan, et al. 2001), we compute the similarity of field names from two schema definitions. The similarity is based on comparisons of field name, structure, and data type and is scored on a [0…1] interval. If the similarity of a field pair is greater than a pre-defined threshold value, the two fields are taken to be a match. The three types of similarity aspects or match criteria used by SemMa are described below:

2.1.1 Name Match

Name match is based primarily on schema field names assuming that field names represent the most useful source of information for matching. The proper matching of two fields is determined by comparing the field name strings and by comparing their meaning. The name match algorithm to resolve conflicts used the following processes:

1) **Tokenisation** was used to parse a field name into a number of tokens based on punctuation, abbreviation, case, etc. For example, “ClientTypeID” would be parsed into “client”, “type” and “id”. The infix (such as “of”, “-“), and suffixes (such as “ing”, “s”) were discarded. For example, “DayOfWeek” would be parsed into “day” and “week”. These tokens are atomic elements for finding the similarity of two fields.

2) **Construction of the Thesaurus** was based on the assumption that the database schema relationships and terms provide the starting semantic information of the database. Abbreviations can be expanded by using other related, existing schema terms. For example, association table name “A_B” in the book order schema refers to table names “author” and “book”. The database thesaurus expands abbreviations by storing all existing terms in a more easily searched data structure.

3) **Finding Synonyms** involves using a powerful, external dictionary, WordNet (WordNet, 2002), as a basis for computing the similarity value of two field names. This differs from the approach taken in

Cupid (Madhavan, etal. 2001) and SemInt (Li and Clifton, 1994), which do not use external dictionaries.

4) **Computing the Name Similarity Value** is done as follows:

Name similarity = nameWeight *(*sum of synonyms tokens*) / (*total tokens* / 2)

"*Name similarity*" ranges between 0 and 1 and indicates the strength of name similarity between two fields. A value of 0 implies two fields are not similar at all, while a value of 1 indicates the two fields are identical based on name. "*nameWeight"* is a pre-defined constant value, which determines the contribution of field name to the similarity of two fields. Different values of *nameWeight* (in the range of 0.5 to 1.0) can be used in different matching algorithms. "*sum of synonyms tokens"* is the number of tokens of a field name in a source schema that are identical or are synonyms to the tokens of a field name in the target schema. "*total tokens*" is the number of total tokens of both field names in the source schema and the target schema. For example, *"sum of synonyms tokens"* of "CustomerFirstName" and "ClientName" is 2 ("name" and "client"/"customer"), and *"total tokens*" is 5 (customer, first, name, client, name). The different types of match algorithms are outlined in the following sections.

2.1.2 Type Match

Data type is an important part of schema information. Data type similarity contributes to total similarity. The set of type similarity values used by Cupid (Madhavan, etal. 2001) were also used here. For example, the type similarity between "number" and " string" is 0.4, while the similarity of "number" and "float" is 0.8.

*Type similarity = typeWeight * similarity of two field types*

The "*typeWeight*" is a pre-defined constant, which determines the contribution of field type to the similarity of two fields. Different "*typeWeight*" values (in the range of 0 to 0.3) were used to evaluate the affect of field type on schema matching performance. For example, if we use "*typeWeight*" = 0.2 and "*similarity of two field types",* for the fields "Sales.UnitPrice(float)" of the target schema and "OrderDetails.UnitPrice (number)" of the source schema, then the "*Type similarity"* = 0.2 * 0.8 = 0.16.

2.1.3 Structure Match

Structure similarity is a measure of the similarity of the contexts in which the fields occur in two schemas. Unlike Cupid, SemMa uses table name and field name to compute the structure similarity. Structure similarity consists of table token similarity and the sum of the field similarities in the schema substructure. Structure similarity is computed as shown below:

1. If two table names are synonyms determined by WordNet, and the primary key fields are synonyms, we consider these two tables to be similar and return the full *structureWeight*.

2. However, if at least two field pairs in the two tables are individually synonyms, we say these two tables are similar and return the full *structureWeight* (based on Cupid approach).

3. Or, we compute the field similarities to represent the structure similarity as below:

   Structure similarity = structureWeight * (*sum of field similarities*) / (*total number of field pairs in current two tables*)

   where "*structureWeight"* is a pre-defined constant value, which determines the contribution of structure to the total similarity of two fields. Different "*structureWeight*" values (in the range of 0 to 0.3) were used to evaluate the contribution of structure to schema matching performance. The term "*sum of field similarities"* is the sum of field pair similarities in two compared tables. The field pair similarity is determined by field type and field name using name match and data type

match described previously. The "*total number of field pairs in current two tables"* is the total number of field pairs in the two compared tables.

In summary, SemMa is a hybrid matcher that combines the name match, data type match and structure match. The sum of similarities computed by these three matches is used to determine the similarity of fields in the source schema and the target schema. The next section will detail the implementation of the SemMa matcher that is encapsulated in three classes.

2.2 Implementation Architecture

The three architectural layers and their major components are shown in the architecture diagram depicted in figure 2. The top layer includes two types of input files: 1) an XML file representation of relational schemas generated using the Microsoft BizTalk editor software (BizTalk, 2002) and 2) a dictionary input file that was retrieved from the WordNet online dictionary (WordNet 2002). The later file contains all words used in the program for schema matching. The output file contains mappings between source schemas and the target schema.

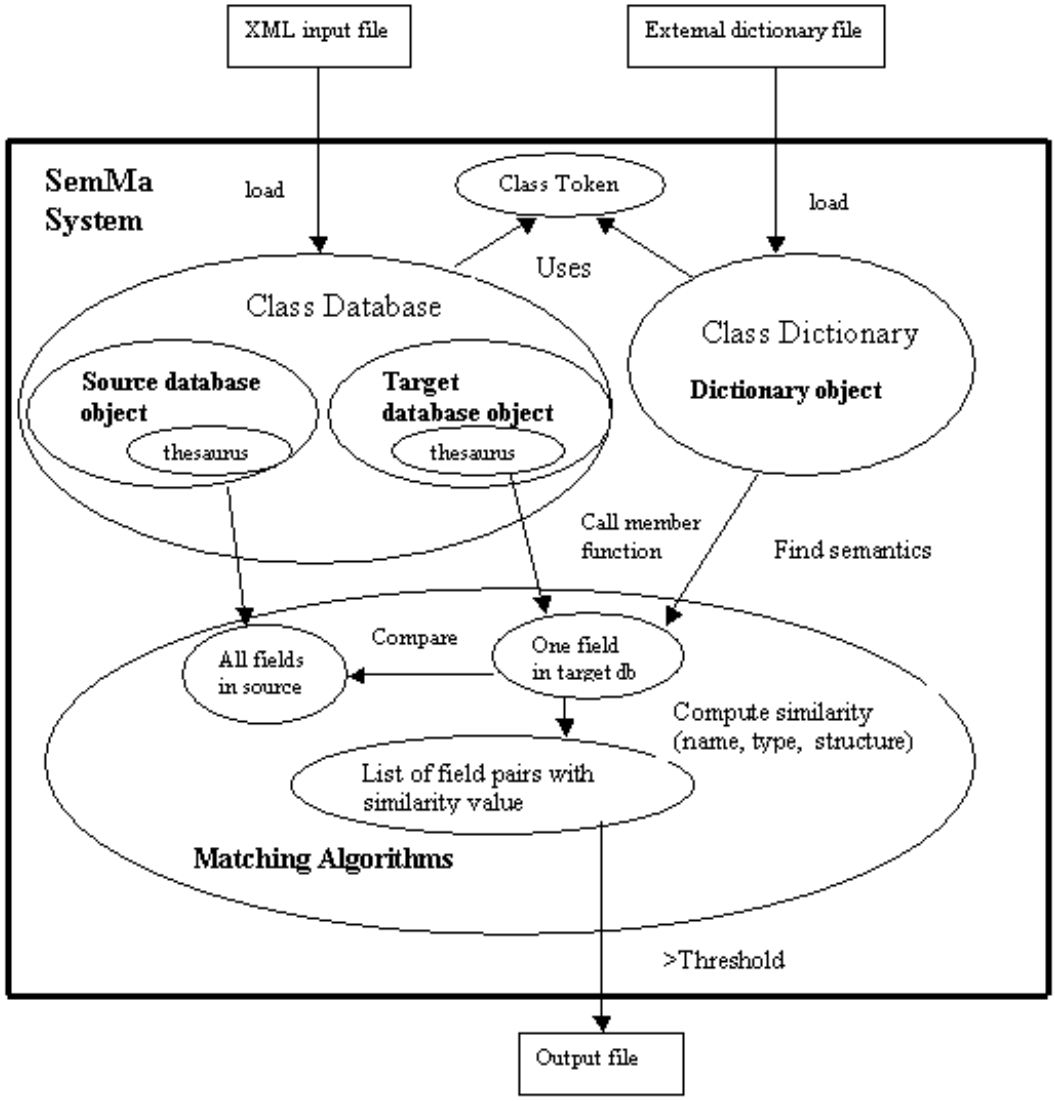


Figure 2. Architecture and Main Components of the Schema Matcher, SemMa.

A mapping is written to the output file when the similarity of field pairs in the source exceeds the defined threshold. The bottom layer is an output file, containing matched field pairs and their similarity values. The middle layer is the SemMa system. SemMa has four components: a source database, a target database, an external dictionary to capture data semantics and a set of schema matching algorithms. Three classes of objects (class D*atabase*, class D*ictionary* and class *Token*) are used to encapsulate these four components in one programming package. The main advantages of using object-orientation here are: 1) encapsulation of data and functions within a class and 2) maintainability through localisation of changes in small modules of code. The three classes are discussed in the following sub sections.

2.2.1 Dictionary Class and Finding Synonyms

The class *Dictionary* creates and maintains a dictionary of words. It takes a text formatted dictionary file as input. The file used was the WordNet external dictionary (WordNet, 2002). WordNet is an online lexical reference system whose design was inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. The D*ictionary* class consists of a structure called *words*, which has a term as root and an array of possible synonyms. Arrays were chosen to simplify the process. The alternative is using a linked list of node pointers. The class *Dictionary* defines various methods for adding words, adding synonyms, tidying strings and tidying synonyms into a standard format. The structure of the class *Dictionary* is shown below:

```
Struct words{
   char root[51];
   int num_of_syns;
   char synonyms[num][51];
};

class Dictionary{
    private:
         word words[num];
         int num_of_words;
     public:
          Dictionary() ; //constructor
          void AddRootWord(char* rootword);
          void AddSynonyms(char* synonyms);
          void PrintAll();
          void TidySameWords();
          void TidyString(char* word, char ch);
          void TidyWord(char* word);
          bool FindRootBySyn(char* syn, char* root);
          bool FindSynByRoot(char* root, char* syn);
};
```

Figure 3. Structure of the Dictionary Class.

The *Dictionary* class uses two member functions to find the synonym of a given term.

1) FindRootBySyn (char* syn, char* root) finds synonyms by locating the root word of a given term. For example, the root "customer" has synonyms "client" and "consumer". If given the term "client" in the source and "consumer" in the target, the *Dictionary* class object, *mydict*, will call this function to determine if "client" and "consumer", are synonyms.

```
Dictionary  mydict;
 char*  root1, root2;
 if    mydict .FindRootBySyn ("client", root1)  = true and
 if    mydict .FindRootBySyn ("consumer", root2)  = true
          then   root1 = root2              // they are synonyms;
          else                              // they are not synonyms.
```

Figure 4. Finding the Root Using a Synonym .

2) FindSynByRoot(char* root, char* syn) will find synonyms of the given root words. For example, the root "sale" has a synonym "agreement", and the root "order" has a synonym "agreement". The function below checks to see "sale" and "order" are synonyms.

```
Dictionary  mydict;
 char* syn1, syn2,
      for each syn1 in mydict .FindSynByRoot ("sale", syn1)  = true and
           for each syn2 in mydict .FindSynByRoot ("order", syn2)  = true
                if  syn1 = syn2
                then "sale" and "order" are synonyms
           end of for
       end of for

       if  syn1 != syn2, "sale" and "order" are not synonyms.
```

Figure 5. Finding a Synonym Using the Root.

2.2.2 Token Class and Term Tokenisation

The *Token* class tokenises a given table name or field and stores the results. A token is a data member in both the structure *field* and structure *table*. *Token* parses a given term (field name or table name) into tokens based on abbreviation, case, punctuation, etc. The structure of the *Token* class is shown below:

```
class Token {
   private:
        int tokenNum;
        char tkn[MAX][51];
   public:
        token();                          //constructor
        token ( const token &tk) ;        //copy constructor
        void AddTokens(char * str);       // add str to token array
        bool Find(char* str);             //find weather str in the token array
        char* GetToken(int index) ;       //get token in position index
        int GetNumOfToken();              //return total num of tokens
        void PrintAll();                  //print all tokens
        bool IsEmpty();                   //return true if empty
};
```

Figure 6. Structure of the Token Class.

2.2.3 Database Class and Schema Matching Algorithms

The class *Database* stores all data schema information and consists of data member structures *table*, *field*, *pairsoffields* and *thesaurus*.

```
struct field{
    char fieldName[51];
    char type[21];
     token field_tokens;          // one field consist of number of tokens
};
struct table{
     char tableName[51];
     field fields[MAX];
      int num_of_fields;
      token table_tokens;        //a table name consist of number of tokens
};
struct pairoffields{
     char dbNameA[51];
     char tableA[51];
     char fieldA[51];
     char dbNameB[51];
     char tableB[51];
     char fieldB[51];
     float sim;
};
```

Figure 7. Key Member Structures of the Database Class.

Data structures *field* and *table* store field and table information for a schema. Structure *table* stores table name, number of *fields*, and the *tokens* of the table. Structure *field* has three data members: *fieldname*, *type* and *field tokens*. Pre-processing of the *field* tokens and *table* tokens, and storing the field name tokens in the structure *field* makes the field matching process more efficient. The data structure *pairoffields* stores the similarity information for one field in the target database and all of its corresponding fields in the source database. The field names along with their respective table name, database name, and similarity value of the pair are represented here.

The *Database* class also contains an array of the thesaurus terms. The thesaurus is used to help discover the semantically equivalent words and phrases in a schema according to the context. For example, in the book order database, to determine the meaning of the abbreviations "A" and "B" in the association table named "A_B", all semantic words in the database thesaurus that contain the words "author" and "book" will be checked since "A_B" is an association table between the tables author and book.

The class *Database* is organized as a hierarchy of database concept terms. A database instance contains tables and a thesaurus that corresponds to terms in its schema. The benefits of using a directed hierarchal graph structure include:

- The thesaurus contains all terms for expanding abbreviations and detecting their semantic meaning.
- Hierarchical organization shows how terms are related to each other and can be easily navigated to retrieve information about the schema instance
- The encapsulation of the schema-matching algorithm enables schema stability and user flexibility. Flexibility means a user can find a specified field mapping by changing the parameter of a member function. For example, a user can find out which field in the Book Order source database is matched to the field "Sales.PostalCode" by calling the member function: compareTwoFields ("Sales. PostalCode", field2).

The *Database* class encapsulates a schema-matching algorithm as a list of member functions as shown below. The database schema information is stored in the following data structure as an instance of the *Database* class.

```
class Database{
    private:
          char dbaName[51];
          table tables[MAX];
           int num_of_tables;
           int size_thesaurus;
           char thesaurus[200][51];
            token db_tokens;
            pairoffields field_pairs[1000];
            int num_of_pairs;
    public:
          Database();
          void AddDbaname(char * dName) { strcpy( dbaName, dName);}
          void AddTableName(char* tableName);//ok
          void AddFieldName(char* fieldName);//ok
          void AddFieldType(char* fieldType);//ok
          void AddTabletokens(char* tableName);
          void AddFieldtokens(char* fieldName);
          void AddToThesaurus( char* word );
          bool ReadTable(int tableIndex, table &oneTable);
          bool ReadField(int tableIndex, int fieldIndex, field oneField);
          void PrintAll();
          void PrintThesaurus();
          void PrintDBTokens();
          //-------------------------------------------------------------------
          void DisplayMostSimilarPair(FILE *outfile);
          float CompareFieldsByName(char* name1, char* name2);
          float CompareFieldsBySys(field f1,field f2);
          float CompareFieldsByType(char* type1, char* type2);
          float CompareFieldsByToken(field f1, field f2);
          float CompareTwoFields(field f1, field f2);
          float CompareTwoTables(table &t1, table &t2);
           //-------------------------------------------------------------------
           int CompareFields( database & db, FILE *outfile);

};
```

Figure 8. Structure of the Database Class.

The schema-matching algorithm shown on the next page uses a structure similarity value stored in tables_sim , which is based on the following assumptions:

- Two tables are similar if two tables are linked by a foreign key to their own entry table, and the table names are synonyms. For example, take the “Sales” table in the target schema and the “Book_Order” table in the source schema as the entry tables. The “Client” table is linked to the “Sales” table by a foreign key, and the “CUSTOMER” table in the source schema is linked to the “Book_Order” table by a foreign key. Client and Customer are also synonyms, so these tables are similar.
- Two tables are similar if there are at least two fields in each table that are similar (based on Cupid) as follows:

  type_sim(“id”, “idref”) = 0.9;
  type_sim(“number”, “int”) = 0.9;
  type_sim(“float”, “int”) = 0.8;
  type_sim(“string”, “int”) = 0.4;
  type_sim(“date”, datetime”) = 0.9;

The token similarity and synonym similarity are computed by calling the *Dictionary* class member functions boolean dictionary :: FindSynByRoot(char* root, char* syn) and boolean dictionary :: FindRootBySyn (char* syn, char* root).

The schema-matching algorithm is a straightforward term matching algorithm as shown below.

```
void Database::CompareFields( database & db, FILE *outfile){
   for each table of this database {
      for each field f1 in this table {
         inicialize  fields_sim =0.0; tables_sim = 0.0;
         for each table in source database {
              tables_sim = compare structure similarity of current two tables
              for each field f2 in source table {
                   fields_sim = tables_sim +
                                compare type similarity (f1, f2) +
                                compare name similarity (f1, f2) +
                                compare token similarity (f1, f2) +
                                compare synonym similarity (f1, f2);
                 store these two field information and the fields_sim;
            }
          }
         for all fields pair {
                if fields_sim > threshold then write to output file
         }
       }
   }
}
```

Figure 9. Schema Matching Algorithm.

In summary, SemMa uses an external dictionary WordNet to generate an internal database thesaurus to determine the semantic meaning of database terms. SemMa automatically computes the similarity of two fields in the source database and target database by computing name similarity, type similarity and structure similarity. The name similarity is defined by equality of names and synonyms of names. The structure similarity is determined by field similarity and table similarity. SemMa is supported by three main classes, which encapsulate the data and operations that operate on the data. The types of similarity can be combined into different algorithms. The effect of this on matching is discussed in the following experimental study.

3 Experimental Validation Study

An empirical evaluation of the schema-matching algorithm (discussed in the last section) was conducted. Five schema-matching algorithms were evaluated by measuring the test results using the same two source schemas and same target schema in all five cases. The proper values for weighting the contribution of name similarity, data type similarity, and structure similarity were evaluated to determine which ones led to better matching performance.

3.1 Test Data

Two relational source schemas (see appendix, based on Bernstein and Rahm, 2000) and one relational target schema were used to test five schema-matching algorithms. A list of all *true matches* between the source and target schemas is given in the appendix. The true matches were identified manually, and serve as a target for 100% correctness for the automated match for the purposes of the evaluation. The source schema in Figure 1 contains 9 tables and 40 fields. It represents a Book Order schema. The source schema in Figure 2 contains 13 tables and 47 fields. It represents a general Product Orders schema. The data

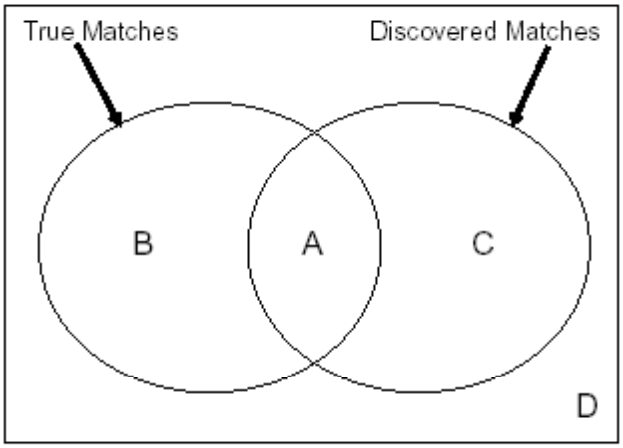
warehouse target schema has one fact table *Sales*, four dimension tables (*Products, Time, Clients and Geography*), and contains 30 fields. There are 27 true matches with respect to the Book Order schema and the target schema and 35 true matches between the Product Orders schema and the target schema. A field in the target schema may match more than one field in the source schema. For example, the field *Sales.OrderID* from the target schema matches both *Book_Order.OrderID* and *B_Ordering.OrderID* in one of the source schemas. These relational schemas were parsed into XML files using the Microsoft BizTalk editor (BizTalk, 2002).

3.2 Threshold Values

As previously discussed, schema matching algorithms return a similarity value between 0 and 1. Therefore a threshold must be selected such that the similarity of two fields above the threshold is considered a match. The choice for a matching threshold should detect most real matches. If a lower threshold is chosen, then the number of false positives will increase, while a higher threshold will find fewer true matches. Previous work with Cupid found that 0.8 was a suitable value for the threshold. This value is used in this study as well.

3.3 Accuracy Measures

To determine the matching accuracy, we used < matching completeness %>, <matching precision %> and <overall quality of the match %> from (Do and Rahm, 2002) based on bounded areas A, B, C and D below.



A = True Positives B = False Negatives C = False Positives
D = True Negatives

Figure 10. A+B=Possible True Matches based on Manual Matching, A+C= Matches Discovered by the Matching Algorithm (D excludes areas A+B+C).

<Matching completeness %> = A / (A+B), is the proportion of true matches detected by SemMa among all the actual true matches determined manually. It measures the percentage of true positives found. This is known in the field of information retrieval as "recall".

<Matching precision %> = A / (A+C), is the proportion of true matches among all the matches the algorithms found. It measures the reliability of SemMa in terms of its consistency. Essentially, it is the percentage of true positives out of all positives detected so it drops if lots of false positives are found. This is known in the field of information retrieval as "precision".

<Overall quality of the match %> = <matching completeness %> * (2 – (1 / <matching precision %>)), measures the overall quality of the schema matching approach as a function of both precision and recall. This prevents bias in construction of the algorithm to favour recall at the expense of precision and vice-versa. Overall quality can be negative if the number of false positive matches exceeds the number of true positive matches (i.e. matching precision < 50%). Such a result would indicate that the post match effort to correct the results will be higher than the gain from the automatic match operation, indicating the match operation was not effective.

As in COMA (Do and Rahm, 2002), if the overall quality of the match exceeds 50%, the automated matcher has had a positive effect on the task. Ideally, all three measures will be 1.0 meaning all true matches are found all the time. In all other cases, precision and recall are larger than the overall quality measure and it becomes difficult to get an overall quality measure higher than 0.5 (Do and Rahm, 2002).

3.4 Matching Algorithms

Five matching algorithms made up of different combinations of name, tokenised name strings, data type, and structure similarity comparisons are discussed with examples in this section. The similarity threshold used was 0.8 in all cases. If the similarity of two fields in the source and target schemas exceeds 0.8, these two fields are considered to match (or map to one another). The fifth algorithm also uses the external dictionary.

Algorithm 1 (field name comparison only): For example, the field name "OrderID" in the source schema and field name "OrderID" in the target schema will map, but "ClientID" and "CustomerID" will not map. Therefore the weight of name is 1.0, and the weights for type and structure are 0.0.

Algorithm 2 (field names and data type comparison): The similarity value of a field pair is the sum of field name similarity and data type similarity. If the field names of two fields are identical, the full nameWeight is assigned to this field pair. The data type similarity depends on the data types of the field pair. The structure similarity is not considered in this case. As in Cupid, the similarity weights of name, data type and structure used were:

```
const float nameWeight = 0.8 ;
const float typeWeight   = 0.2;
const float StructWeight = 0.0;
```

Algorithm 3 (field names, the tokens of field names and field type comparison): The similarity weights of name, data type and structure used were:

```
const float nameWeight = 0.8 ;
const float typeWeight   = 0.2;
const float StructWeight = 0.0;
```

In this case, the similarity of name is considered as the sum of token similarity in the field pair. For example the name similarity of "ClientName" and "ClientFirstName" would be:

name similarity = nameWeight * k / ((m+n) / 2) = 0.8 * 2 / ((2+3) / 2) = 0.65

where: k = the number of tokens that are identical in the field pair
m = the number of the field tokens in the source schema
n = the number of the field tokens in the target schema

The total weight for "ClientName" and "ClientFirstName" is: similarity = 0.65 + 0.2 = 0.85

Algorithm 4 (field names, tokens of field names, field type and structure similarity): This algorithm is the same as algorithm 3 except for structure similarity. The similarity of a field pair is the sum of name similarity, data type similarity and structure similarity. We use the follow parameters in this case:

```
const float nameWeight = 0.6 ;
const float typeWeight    =0.2;
const float StructWeight= 0.2;
```

Algorithm 5 (field names, tokens of field names, field type, structure similarity and an external dictionary WordNet): The same weights used with algorithm 4 are used here. The similarity of field names is determined by finding the token synonyms of field names.

The name similarity = nameWeight * (sum of token synonyms) / (num of tokens / 2)

3.5 Limitations

This exploratory study only used two sample relational schemas to test the performance of SemMa. More extensive studies involving the use of larger scale schemas and schemas with varying actual similarity is also desirable. The use of larger schemas affects the search space for match candidates (Do and Rahm, 2002). The SemMa program currently does not read other schema formats, such as text files, other than BizTalk (BizTalk, 2002) formatted XML schema. This study did not attempt to define a new external dictionary and did not involve the improvement or redesign of any existing schema matching tools. The use of other external inputs such as ontologies (Wache et al., 1999) would also be desirable but has not been explored here. Other orderings and combinations of similarity aspects were not examined within the scope of this study. This study only looked at similarity in terms of schema properties and did not look at data content similarity. The speed of performing matches with SemMa was not a concern in this study.

4 Analysis of Results

For each of the five algorithms shown on the x-axis in figures 11 and 12, the value of each of the three

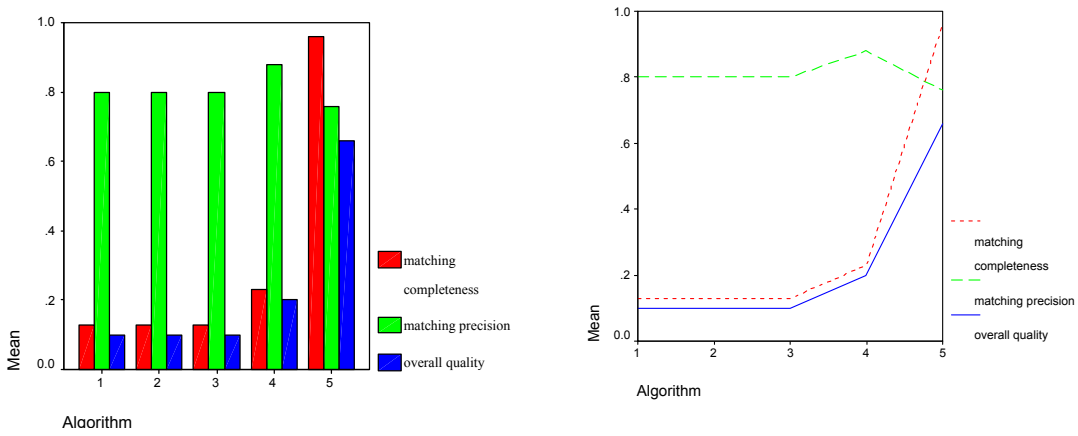


Figure 11. Match Results by Algorithm Using the Book Order Source Schema.

metrics (completeness (aka recall), precision and overall quality) is given on the y-axis. Figure 11 shows the results for the Book Order Schema match and figure 12 shows the results for the Product Orders schema match.

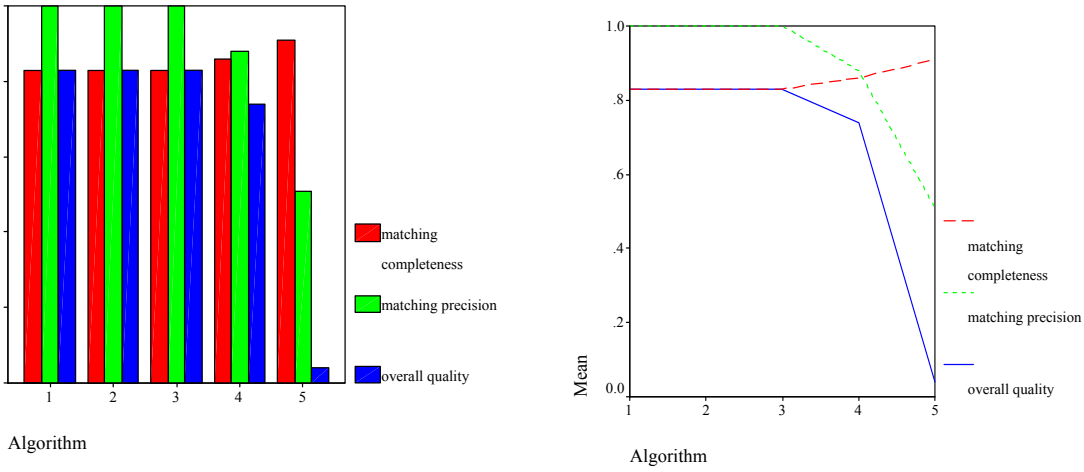


Figure 12. Match Results by Algorithm Using the Product Orders Source Schema.

Overall quality can also be expressed as (True Positives – False Positives)/(Real Matches) so in other words, the overall quality value is reduced based on the number of matches found to be true which were not actually real matches (i.e. false positives). In terms of the manual match, the Book Order schema was less similar to the target than the Product orders schema.

Precision was consistently high for both cases, except for algorithm 5 in the case of using the Product Orders schema as input. In this instance, the precision was near 0.5 resulting in low overall quality of the match despite a high recall level. Recall that algorithms 5 used an external dictionary, WordNet, to find synonyms. Algorithm 5 had the highest recall (i.e. ability to detect true matches) among the five algorithms with a matching completeness of 96% for the Book Order schema and 93% for the Product Orders schema.

Poor precision for algorithm 5 appears to be due to mismatching of synonyms of the field tokens (5 out of 9 for the Book Order schema, and 21 out of 33 for the Product Orders schema), which resulted in a large number of false positives. For example, SemMa treated "Region" and "Territory" as synonyms but they actually have different semantics according to the manual match. The different semantics between "Year" in the target schema and "Year" in the Book Order schema were also not detected. A mismatch on ClientTypeID of the target with CustomerID in the source also occurred because most of the tokens of these two fields are highly similar although the fieldnames have different semantics in the manual match. In order to reduce false positives, an ontology may be needed to supplement or replace the WordNet dictionary.

No difference in the three metrics was found with respect to algorithms 1 (name), 2 (name and data type) and 3 (name, data type and tokens). However, overall quality of the match was low as shown by low matching completeness (13.3%) for the Book Order schema, which used many different field names that had the same semantics as those found in the target schema but was unable to detect many of the true matches. Because the Product Orders schema and target have high duplication of field names, these three algorithms achieved a high matching completeness (83%), a high matching precision (100%) and high overall quality of match (83%) indicating that SemMa is not a good matcher if the names of many fields differ as name match was the common factor between these three algorithms.

Algorithm 4 added structure similarity to the criteria of algorithm 3. It matched more fields for both source schemas, but resulted in only a small increase in matching completeness (from of 13.3% to 23.3% for the Book Order schema, and from 83% to 86% for the Product Orders schema). Structure similarity did detect matches such as: TIME.Date in the target mapping to BOOK_ORDER.OrderDate in the Book Order schema. The table "Sales" in the target schema and the table "BOOK_ORDER" are also similar since their two fields "OrderID" and "OrderDate" are identical and "TIME.Date(id)" is linked with "Sales.OrderDate" by a foreign key. Matching precision remained high due to the identical field names in the source and target schemas.

Figures 13 and 14 examine the results of varying the type weight and structure weight used by the algorithms that include type (algorithms 2 to 5) and structure (algorithms 4 and 5).

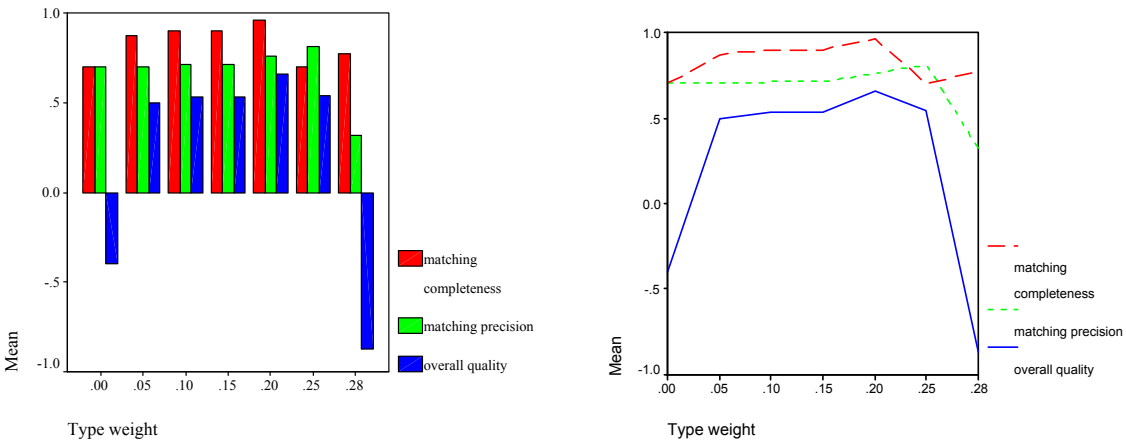


Figure 13. Overall Match Quality by Type Weight for the Book Order Schema**.**

Overall quality of the match became negative when either type weight or structure weight exceeded 0.25 and required a type weight of at least .05 to remain positive. The sum of type weight and structure weight should not exceed 0.5 in order to maintain a positive overall quality level. Many fields can have the same data type in tables that have high structure similarity, but the fields may have different semantics. The field pair similarity mostly depends on the semantics of the field names. Overall quality of the match becomes negative (matching precision < 50%) when too many of the detected matches are false positives due to a high weight on data type or data structure. This means the manual effort to resolve false positives exceeds the benefit of automatic match.

The ideal value for each of the 3 measures (matching completeness, matching precision and overall quality) would be a value of 1.0. This would mean all true matches were discovered and no false positives were found. Generally, this is not the case but if overall quality >= 50%, for example, matching completeness >= 75% and matching precision >= 75%, the matcher has a positive effect.

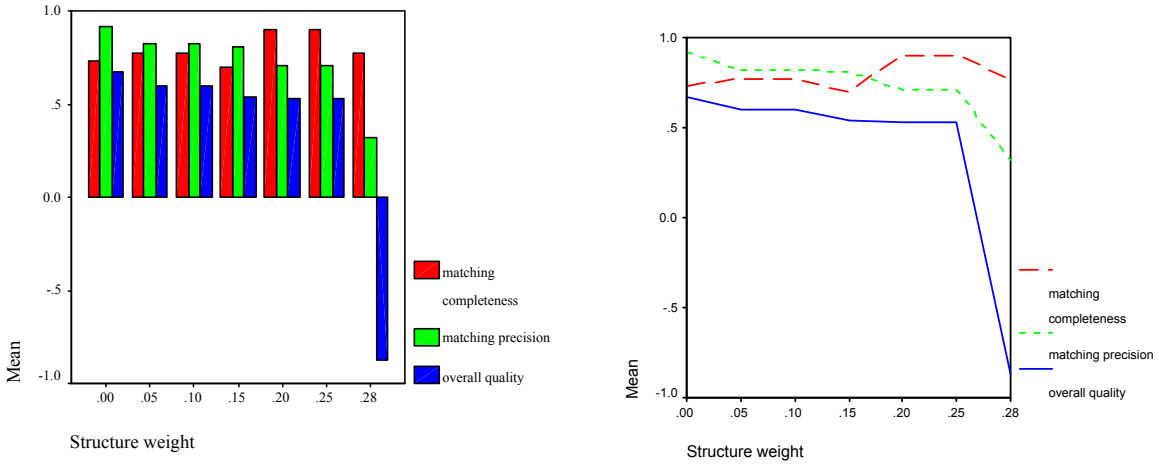


Figure 14. Overall Match Quality by Structure Weight for the Book Order Schema**.**

In summary, the following conclusions about SemMa can be drawn from the results.

- The addition of structure similarity in algorithm 4 and of the dictionary in algorithm 5 resulted in higher recall values but this gain was offset by a drop in precision and thereby a drop in overall quality of the match. The use of structure and the dictionary helped in finding more of the true matches but had the side effect of increasing the number of false positives. Increase in false positives means a decrease in precision since the number of true matches discovered is small relative to the number of matches detected as positive.
- Like previous studies, this study found the contribution of data type should be between 0.15 and 0.2 in a hybrid matcher.
- More effort to reduce false positives is needed.

4.1 Comparing SemMa with SemInt and Cupid

In this section, we briefly compare SemMa with two other schema matching approaches, SemInt (Li and Clifton, 1994) and Cupid (Madhavan, et al., 2001). SemMa was also compared with Cupid via experimental evaluation.

To compare these three schema matching approaches, the input, output, implementation and matching performance measurement aspects were summarized in table 2 (based on Do et al., 2002).

- **Input**: What kind of input data has been used? The simpler the test sample and the more auxiliary information given, the more likely the system can achieve greater effectiveness but less automation is achieved.
- **Output**: What information has been included in the output? How much post-processing is needed?
- **Implementation**: What language is used to implement the approach? Is a GUI available?
- **Matching Performance Measurement**: What measures have been used to evaluate the matching performance?

			SemInt	**CUPID**	**SemMa**
Input schema types			relational, files	XML, relational	relational, XML
Output format				links with similarity values	link with similarity values
Match Performance	Metadata representation		unspecified (attribute-based)	extended ER	relational data model
	Match granularity		element-level: attributes (attribute clusters)	element and structure-level	element level and structure level
	Match cardinality		1:1	1:1 and n:1	1:1
	Schema-level match	Name-based	-	name equality, synonyms, hyponyms, homonyms, abbreviations	name, token equality, synonyms, hyponyms, abbreviations
		Constraint-based	several criteria: data type, length, key info,…	data type and domain compatibility, referential constraints	data type, and referential constraints
		Structure matching	-	matching sub trees, weighted by leaves	table and field similarity
	Instance-level matchers	Text-oriented	-		
		Constraint-oriented	character / numerical data pattern, value distribution, averages		
	Reuse /auxiliary information used		-	thesauri, glossaries	database thesaurus and external dictionary WordNet
	Combination of matchers		hybrid	hybrid	hybrid
	Manual work /user input		selection of match criteria (optional); selection of matching attributes from attribute clusters	user can adjust threshold weights	user can vary type and structure weights
	Application area		data integration	data translation, but intended to be generic	schema integration
	Matching Quality measurement			by looking correspondences elements	completeness, precision and overall quality
Implementation			C (non OO)	VB (object-based)	C++ (object-oriented)
Remarks			neural networks	tree matching	external dictionary

Table 2. Summary of Key Aspects of SemInt, Cupid and SemMa.

SemMa was based on a study of Cupid. Both are purely schema based, and both combine element level and structural level matching. SemMa and Cupid use the same input format, generated by Microsoft BizTalk.

Identical source and target schemas, converted to XML files were used with both Cupid and SemMa to evaluate their relative performance. The results are shown in figures 15 and 16 below.

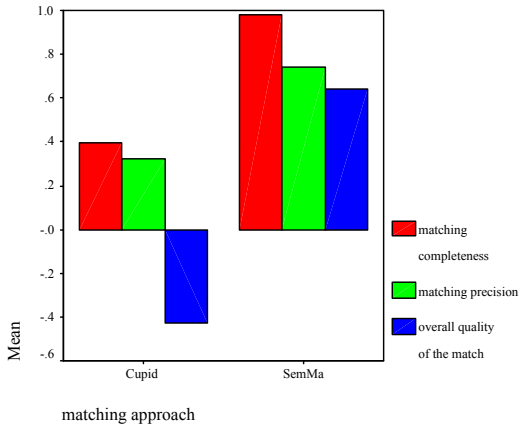


Figure 15. Recall, Precision & Overall Quality with Cupid and SemMa (Book Order Schema).

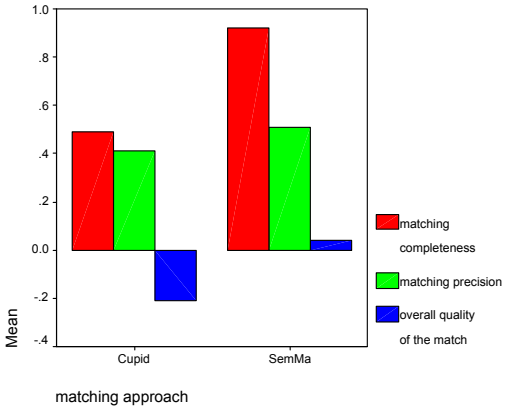


Figure 16. Recall, Precision & Overall Quality with Cupid and SemMa (Product Orders Schema).

In both cases, SemMa achieved higher overall quality of match relative to Cupid. Algorithm 5 of SemMa was used to produce the graphs shown in figures 15 and 16.

- The matching precision and matching completeness of Cupid for both source schemas were less than 50%. Therefore the overall quality of the match for Cupid was negative, while the overall quality of the match for SemMa was 66% for the Book Order schema and 4% for the Product Orders schema. Cupid had a greater false positive effect for these two sample schemas than SemMa did but SemMa also needs to reduce detection of false positives.

- The use of an external dictionary increased the recall values for SemMa allowing it to exceed the performance of Cupid in this experiment. Cupid could not match “product” and “book” in the case of the first source schema, because the internal thesaurus of Cupid does not list these two terms as synonyms. An ontology may provide additional semantics to address this problem.

5 Conclusions

In this project, an automated solution for the well-known problem of semantic schema matching has been examined. Identifying semantically similar terms in independently designed schemas is a critical step towards achieving interoperability and integration in many applications, such as data warehousing, schema integration for distributed systems, e-commerce message translation, and semantic query processing.

A number of solutions to automate the schema matching operation have been proposed in the literature. Unlike these existing solutions, SemMa used three object-oriented classes to encapsulate an external dictionary WordNet, a target database schema, source database schemas and a schema-matching algorithm. SemMa, used Microsoft BizTalk (BizTalk, 2002) to parse the relational schema sources into XML. Next, SemMa transformed the XML files into Database class object instances. Schema matching algorithms were encapsulated as *Database* class member functions that performed field match tasks. The matching algorithms were based on finding semantic similarity in terms of database field name, field data type and table structure similarity. Synonym sets were used to represent the semantics of database field terms. The semantic meanings of the field terms were determined using an independently defined, external dictionary. Different weightings were also examined for the contribution of name, data type and structure to determining a match.

In the experimental validation step of this study, the matching ability of five combinations of similarity factors was used to identify strengths and limitations, and to provide suggestions for future work. The five algorithms were composed from the similarity aspects (name, data type, etc.) and tested via a series of experimental runs to evaluate performance with respect to overall match quality, recall and precision. The experimental runs showed that using an external dictionary such as WordNet had a positive effect on recall, but that this effect was offset by an increase in false positives for schemas for which the dictionary was not adequate in terms of making finer distinctions and that the weight given to name similarity may require adjustment. The experimental runs also showed that using suitable similarity weights for name, data type and structure was important to schema matching performance.

Proposed future work includes improving the schema matching algorithms, considering the use of externally defined ontologies (Wache et al., 1999) as semantic input in addition to the use of a more comprehensive external dictionary, designing a graphical user interface to complement the SemMa architecture and testing larger scale schemas. Further experimental evaluation of automated match algorithms is an essential way of making progress on this hard problem of semantic schema matching.

In the case of schema matching for data warehousing applications, the deterministic structural characteristics of a data warehouse star schema may prove useful in detecting semantic similarity. For example, each instance of the central fact table of a star schema acts as an entry point into the matching instance in each dimension table. These composite primary key to foreign key links could be used to detect similar relationships in the source schema(s) to provide additional match information.

More powerful algorithms and additional semantic discovery knowledge sources such as ontologies are needed to determine the distinctions between less obvious terms, such as “Territory” vs. “Region”, and “ClientTypeID” vs. “CustomerID” to reduce false positives. Domain ontologies may be useful in avoiding such mismatches since they provide greater information on term relationships, which could be exploited here.

Improving semantic matching is an extremely interesting area for future research. Additional experiments will provide greater sophistication in identifying appropriate similarity weights for name, data type structure and additional kinds of similarity factors. Scalability analysis and further testing on large-sized real-word schemas will contribute to further improvements. The work reported here is an exploratory step in this direction.

References

Bergamaschi, S., Castano, S. and Vicini, M. (1999). Semantic integration of semi- structured and structured data sources. ACM *SIGMOD Record* 28(1), 54-59.

Bernstein, P. A. and Rahm, E. (2000). Data warehouse scenarios for model management. In: *ER 2000 Conference Proceedings*. http://research.microsoft.com/~philbe/ER2000wSpringer.pdf. (retrieved in June 2002)

BizTalk Editor. Microsoft. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/bts_2002/htm/lat_xmltools_editor_intro_cyvg.asp. (Jan – Dec 2002)

Castano, S., DeAntonellis, V. and DeCapitani, S. (2001). Global viewing of heterogeneous data sources. *IEEE Transactions on Data and Knowledge Engineering* 13(2), 277-297.

Do, Hong-Hai, Melnik, S. and Rahm, E. (2002). Comparison of schema matching evaluations. *Proceedings of GI-Work "Web and Database"*, Erfurt, Oct. 2002. http://www.cs.umd.edu/projects/plus/SHOE/pubs/heflin-thesis.pdf. (retrieved March 2001).

Do, Hong-Hai and Rahm, E. (2002). COMA-A system for flexible combination of schema matching approaches. *Proceedings of the 28th VLDB Conference*, Hong Kong, China.

Doan, A. H., Domingos, P. and Halevy, A. (2001). Reconciling schemas of disparate data sources: a machine-learning approach. *ACM SIGMOD, May 2001,* 509-520.

Goh, C. H. (1997). Representing and reasoning about semantic conflicts in heterogeneous information Sources. *Ph.D. Thesis, MIT Sloan School of Management.* http://context2.mit.edu/coin/publications/goh-thesis/goh-thesis.pdf (retrieved in March 2002).

Heflin, J. D. (2001). Towards the semantic Web: knowledge representation in a dynamic, distributed environment. *Ph.D. Thesis, University of Maryland, 2001*. http://www.cs.umd.edu/projects/plus/SHOE/pubs/heflin-thesis.pdf. (retrieved in March 2001).

Heiler, S. (1995). Semantic interoperability. *ACM Computing Surveys* 27(2), 271-277.

Hull, R. (1997). Managing semantic heterogeneity in databases: a theoretical perspective. *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* May 1997.

Lawrence, R. and Barker, K. (2001). Integrating relational database schemas using a standardized dictionary. *SAC'2001 - 16th ACM Symposium on Applied Computing March 2001,* Las Vegas, USA, 225-230.

Li, W. and Clifton, C. (1994). Semantic integration in heterogeneous databases using neural networks. *Proceedings 20th VLDB Conference 1994,* 1-12.

Madhavan, J., Bernstein, P. A. and Rahm, E. (2001). Generic schema matching with Cupid. *Proceedings of the 27th VLDB Conference 2001,* 49-58.

Milo, T. and Zohar, S. (1998). Using schema matching to simplify heterogeneous data translation. *Proceedings of the 24th VLDB Conference 1998,* 122-133.

Mitra, P., Wiederhold, G. and Jannink, J. (1999). Semi-automatic integration of knowledge sources. *Proceedings of Fusion 1999, Sunnyvale, USA*.

Ouksel, A. M. and Sheth, A. (1999). Semantic interoperability in global information systems. *ACM SIGMOD Record* 28(1), 5-12.

Palopoli, L., Sacca, D., Terracina, G. and Ursino, D. (1999). A unified graph-based framework for deriving nominal inter-schema properties, type conflicts and object cluster similarities. *Proceedings of the 4th IFCIS Conference On Cooperative Information Systems, IEEE Computer*, 244-253.

Rahm, E. and Bernstein, P. A. (2001a). On matching schema automatically. *Microsoft Research Publications*. http://www.research.microsoft.com/pubs. (retrieved on 5 Jun 2002).

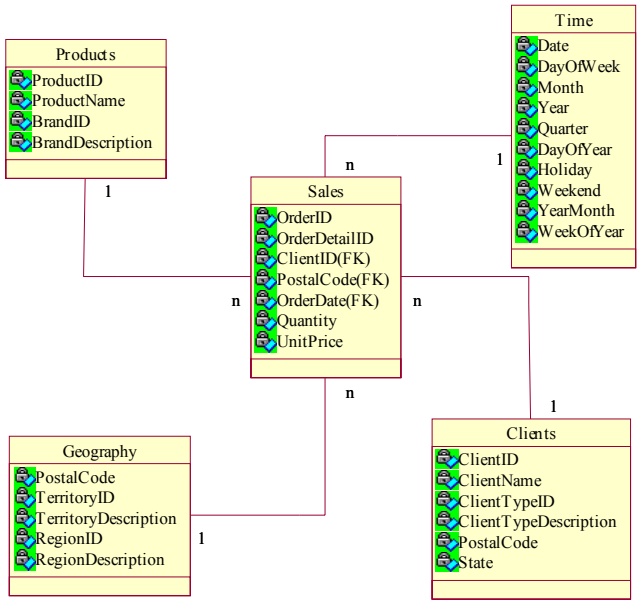
Rahm, E. and Bernstein, P. A. (2001b). A survey of approaches to automatic schema matching. *The VLDB Journal* 10, 334-350.

Stohr, T., Muller, R. and Rahm, E. (1999). An integrative and uniform model for metadata management in data warehousing environments *Proceedings of the International Workshop on Design and Management of Data Warehouses*, Germany, June 1999.

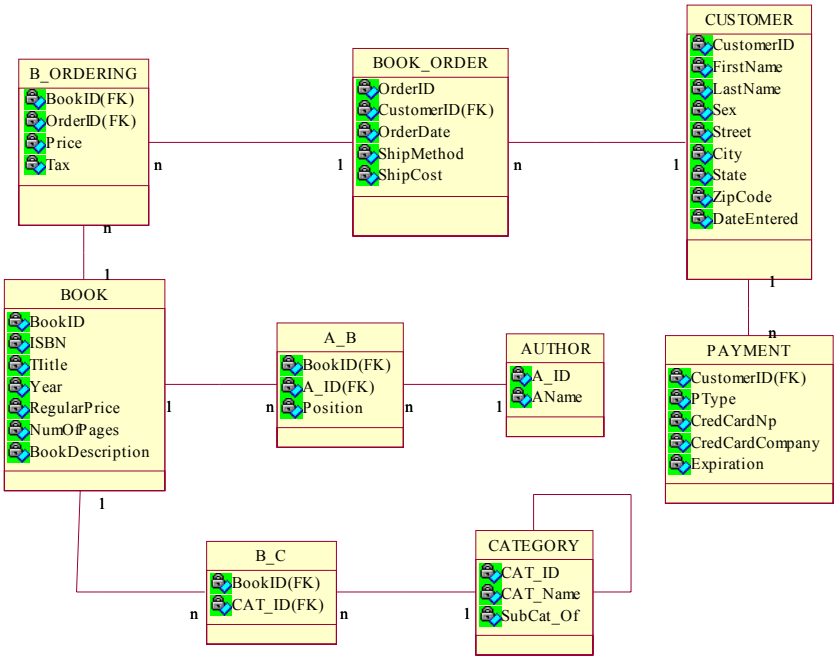
Wache, H., Vogele, T., Visser, U., Stuckenschmidt, H., Schuster,G., Neumann, M. and Hubner, S. (2001). Ontology based integration of information - A survey of existing approaches. *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing, Seattle*, WA 2001, 108-117.

WordNet: Online Lexical Dictionary. Cognitive Science Lab at Princeton. http://www.cogsci.princeton.edu/~wn/. (Jan-Dec 2002)

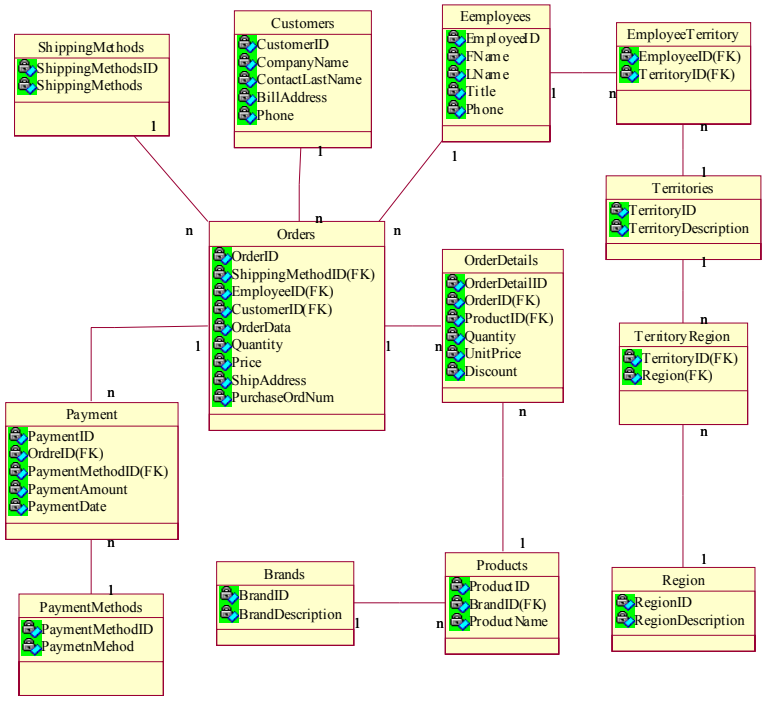
Appendix



Warehouse Star Schema



Relational Book Order Schema



Relaitonal Product Orders Schema

Table 3. Correspondence Table: Manual matches between attributes in the target (star) schema and the source schema, Book Order. In some cases, "id" and "idref" are shown rather than (data type) to indicate the attribute is a table key.

Table Name	**Star Schema (target)**	**Book Order Schema (source)**
Product	Products.ProductID(id)	BOOK.BookID(id)
		B_C.BookID(idref)
		A_B. BookID(idref)
		B_ORDERING. BookID(idref)
	Products.ProductName(string)	BOOK.Title(string)
Sales	Sales.OrderID (id)	BOOK_ORDER.OrderID (idref)
		B_ORDERING.OrderID(idref)
	Sales.ClientID(idref)	BOOK_ORDER.CustomerID(idref)
		PAYMENT.CustomerID(idref)
		Customer. CustomerID(idref)
	Sales.PostalCode(idref)	CUSTOMER.ZipCode(string)
	Sales.OrderDate(idref)	BOOK_ORDER.OrderDate(dateTime)
	Sales.UnitPrice(float)	B_ORDERING.Price(number)
	Sales.OrderDetailsID(string)	B_ORDERING.BookID(idref)
	Products.ProductID(id)	BOOK.BookID(id)
		B_C.BookID(idref)
		A_B. BookID(idref)
		B_ORDERING. BookID(idref)
Clients	Clients.ClientID(id)	BOOK_ORDER.CustomerID(idref)
		PAYMENT.CustomerID(idref)
		Customer. CustomerID(idref)
	Clients.ClientName(string)	CUSTOMER.FirstName(string)
		CUSTOMER.LastName(string)
	Clients.PostCode(string)	CUSTOMER.ZipCode(string)
	Customers.State(string)	CUSTOMER.State(string)
Geography	Geography.PostalCode(id)	Customers.ZipCode(string)
Time	Time.Date(id)	BOOK_ORDER.OrderDate(dateTime)

Table 4. Correspondence Table: Manually matched attributes in the target star schema and the relational source schema, Product Orders.

Table Name	**Star Schema (target)**	**Product Orders Schema (source)**
Product	Products.ProductID(id)	Products.ProductID(id)
		OrderDetails.ProductID(idref)
	Products.ProductName(string)	Products.ProductName(sting)
	Products.BrandID(string)	Products.BrandID(idref)
		Brands.BrandID
	Products.BrandDescription(string)	Brands.BrandDescription(string)

Sales	Sales.OrderID (id)	Orders.OrderID (id)
		OrderDetails.OrderID(idref)
		Payment.OrderID(idref)
	Sales.ClientID(idref)	Orders.CustomerID(idref)
		Customers. CustomerID(id)
	Sales.PostalCode(idref)	Customers.PostalCode(string)
	Sales.OrderDate(idref)	Orders.OrderDate (dateTime.tz)
	Sales.Quantity(int)	OrderDetails.Quantity(int)
	Sales.UnitPrice(float)	OrderDetails.UnitPrice(number)
	Sales.Discount(float)	OrderDetails.Discount(number)
	Sales.OrderDetailsID (string)	OrderDetails. OrderDetailID(id)
	Sales.ProductID(idref)	OrderDetails.ProductID(idref)
		Products.ProductID(id)
Clients	Clients.ClientID(id)	Customers.CustomerID(id)
		Orders.CustomerID(idref)
	Clients.ClientName(string)	Customers.CompanyName(string)
		Customers.ContactFirstName(string)
		Customers.ContactLastName(string)
	Clients.PostCode(string)	Customers.PostCode(string)
	Clients.State(string)	Customers.StateOrProvince(string)
Geography	Geography.PostalCode(id)	Customers.PostalCode(string)
	Geography.TerritoryID (string)	Territories. TerritoryID(id)
		TerritoryRegion. TerritoryID(idref)
		EmployeeeTerritory. TerritoryID(idref)
	Geography.TerritoryDescription (string)	Territories. TerritoryDescription(string)
	Geography .RegionID(string)	Region.RegionID(id)
		TerritoryRegion. RegionID(idref)
	Geography.RegionDescription (string)	Region.RegionDescription(string)
Time	Time.Date(id)	Orders.OrderDate(dateTime)

Note: The table shows tableName.attributeName(datatype). In some cases, “id” and “idref” are shown rather than data type to indicate the attribute is a table key.