Semantic Interoperability of Heterogeneous Systems

Phani Chilukuri[*] Toni Kazic[*,†] [*]

February 10, 2004

[*] Depts. of Computer Science and [†]Health Management Informatics
University of Missouri, Columbia

Abstract

With the diffusion of information among different systems across the World-Wide Web, many systems need information from others. Building such interoperable systems is not just a matter of making them speak the same language syntactically, but also enabling them to understand the semantics of the other systems. We previously defined a formal language, *Glossa*, to computably specify the semantics of those ideas, data, and algorithms that databases and other web resources wish to automatically exchange [14].

This paper reports a proof of concept demonstration of semantic interoperability using *Glossa*. We have developed a prototype implementation of the machinery needed for a database using SQL to interoperate with *Glossa*. Translation of information requests between a system and *Glossa* proceeds in two steps: mapping of notions indigenous to the database or to *Glossa* to an intermediate hybrid form, and conversion of that hybrid to the output query language using a common grammar (a metagrammar) and SQL → *Glossa* and *Glossa* → SQL parsers. The mapping preprocessors were developed in Java and the parsers and their metagrammar were developed in Turing Extender Language (TXL). We chose MaizeDB as a testbed, and successfully tested the mappers and parsers for their ability to interconvert a suite of test queries over MaizeDB. The parsers are valid for any ANSI-92-compliant flavor of SQL and the mappers run on Java versions 1.3.X and above. Though the mappings are necessarily specific to MaizeDB, they can be easily adapted for any database or other web resource.

[*]University of Missouri Bioinformatics Technical Report 2004-1. **Keywords:** database interoperability, semantics, *Glossa*, databases, ontologies.

1 Introduction

Designing interoperable systems has been a challenging task, especially with the rapid growth of the Web in the past decade. A multitude of biological databases and repositories, digital libraries with terabytes of data and images, and servers of specialized computations have spread information globally. Making this wealth interoperable is complicated by a lack of consensus on everything from schema (*e. g.*, see the notions of "gene" implemented in references 9, 19, 21, 22, 25, 28) to data models[1] [5, 6, 10, 16, 20], query languages [7, 10], and schemes for addressing resources distributed over the Internet [4, 12, 29].

Many researchers have worked on the problem of interoperability, especially of databases, for the past two decades and have proposed a multitude of approaches. One approach is to federate different relational databases by several different methods [7, 24, 26, 27]. In schema integration, a global level schema is built as an upper layer that reconciles individual databases' schemata. In schema translation, each individual schema maps itself to a translating schema which interprets differences in the ways the different schemata represent the same idea (for example, as an attribute in one schema and as an entity in another). This is also known as loose coupling of the schemata. But just manipulating schemata works only if the systems are structured systems and use the same notions in the same way. It is not robust to changes in the participating databases' syntax, schema, or semantics.

Another approach is to simply subsume all participating databases into a single database with a unified schema (*e. g.*, InterPro: see reference 8). Integrated databases are even less robust to changes in their subsumed databases. (In this paper we ignore the problem of system level heterogeneity, which occurs when databases use different operating environments and programming languages. This can be eliminated by using platform independent intermediary systems such as Java, JDBC/ODBC, XML, or CORBA types of technologies [17].) Both federated and integrated approaches have always assumed the data models of the participating databases are the same (usually relational). For unstructured data resources such as those proliferating on the World-Wide Web, the problem is much worse because they can differ radically in their data models.

But the most fundamental problem of interoperability is the lack of consensus on the semantics of the ideas and data used by the different resources. This semantic heterogeneity means that requests for information from different databases using the same term will retrieve different and not necessarily consistent data, independent of how the databases arrange and query their data. Despite many heroic efforts to define the semantics of terms, ranging from controlled vocabularies and ontologies [2, 18], to international nomenclature committees (*e. g.*, reference 11), each resource remains completely free to map the words to radically different ideas in its domain model and schema. The semantic heterogeneity of data resources, whether databases or computations, remains the greatest barrier to achieving their reliable interoperability. It is particularly difficult

[1] We use **data model** to denote the method used to structure data for computation and **domain model** as the abstract representation of a universe of discourse [20, 23]. The size and complexity of a model is independent of whether it models the domain or the domain's data. For example, domain models can represent very restricted universes of discourse (*e. g.*, DNA sequence; see reference 21) and data models can be extremely complex (*e. g.*, the declarative database used for biochemistry in *Moirai*, see reference 13). This distinction is crucial because the different solutions proposed for interoperability differ in their assumptions about the domain and data models.

in biology, which as a subset of natural languages has no language enforcement mechanism.

We have previously suggested an approach to semantic interoperability which circumvents the need for common data models and agreement upon semantics [14]. It employs a formal language to define the semantics of complex ideas by individual data resources, independent of their data model, query/interface language, and implementation language. *Glossa* provides tools for computably declaring the semantics of ideas, data, and computations so that they can be shared among different computational entities. Its axiomatic notions, called **semiotes**, can be combined to define more complex ideas: the semantics of the construct denoting the complex idea is specified by the semantics of its axioms and of *Glossa*'s grammar. Any system that wishes to exchange information with *Glossa* needs only a public set of mappings between its local notions and *Glossa*and a parser to convert between its indigenious query language and *Glossa*.

This paper demonstrates database interoperability using *Glossa*. We chose MaizeDB as a test case because it is built using the Sybase relational database management system; uses SQL as its query language; and expresses the semantics of genetically and physically mapping markers to chromosomes in eukaryotes [22]. We defined new semiotes in *Glossa* to express these ideas, implemented mappings between the notions in MaizeDB and *Glossa*, and developed code that transforms SQL queries to *Glossa* requests and *vice versa*. Our results illustrate the fundamental machinery needed for diverse, heterogeneous systems to reliably and automatically interoperate with accurate semantics.

2 Materials And Methods

2.1 Languages and Systems

Java, TXL, *Glossa*, and Quintus Prolog 3.4 have been used in our implementation [14, 30–32]. The MaizeDB notions are expressed in *Glossa* as semiotes and constructs of semiotes ("bundles"). The mappings are defined in a tab-delimited ASCII text file and can be modified without affecting the rest of the system. We chose TXL to implement the parsers and their metagrammar because it provides an excellent environment for language translation, with straightforward expression of grammars and well-characterized backtracking. TXL is freely available to academic users and runs on Solaris, Linux, and Windows platforms. Prolog was used for the query language of the testing database (see Section 2.4). To increase platform independence, we chose Java as the programming language to implement mappings between MaizeDB notions and *Glossa* and mapping transformation routines ("mappers").

2.2 SQL `select` Semantics

We briefly review the syntax of SQL `select`, shown in Figure 1. *select_list*, *table_list*, *conditional_expression*, and *sort_order* are respectively: the list of desired variables used as relational table column names; the list of the relational tables containing those variables; conditional expressions bounding the instantiated values of the variables; and the criteria for sorting the data. The `where` and `order by` clauses are optional. The literal meaning of the query is "fetch all the data in columns *select_list* from tables *table_list*, such that if there is a `where` clause they satisfy its conditions expressed in *conditional_expression*, and if there is a `order by` clause sort them

```
select select_list
from table_list
{where conditional_expression}
{order by sort_order}
```

Figure 1: The syntax of the SQL `select` command. $\{X\}$ denote an optional pattern X.

```
select     LG.Name + T2.Name, L.ID#, L.Name, L.Fullname
from       Locus L, LinkageGroup LG, Term T2

where      L.Species = 12808
           and L.Type != 131
           and L.ID# in (select distinct Locus from Locus#Coordinates
                         where Map not in (select ID# from Map
                                           where Name like 'bins%')
                         and and Bin=NULL)
           and LG.ID# =* LinkageGroup
           and T2.ID# =* Arm
           and LOWER(L.Name) like '[0-9a-pr-z]%'
           and LinkageGroup in (select ID# from LinkageGroup
                                where Type = 104)
order by L.Name
```

Figure 2: A complex MaizeDB query showing nesting of two `select`s.

according its conditions *sort_order*". Notice that successfully designing an SQL query requires a clear understanding of the database's schema and semantics.

2.3 Selection of Prototype Queries for System Development

To develop our code we chose a set of SQL queries currently used by MaizeDB (the "test suite"). These queries use the `select` operator to retrieve data, the fundamental task for interoperating databases. The queries chosen use complicated MaizeDB notions and complex SQL conditions. The latter include nested SQL statements, `right joins`, formatting patterns such as `convert`, `substring`, and string concatenation, and ordering operations based on different column names and having an output formatting statement such as `convert`. An example is shown in Figure 2. It has a nested SQL statement, a `right join` operation denoted by =* , string matching to a regular expression using `like`, and string concatenation denoted by a + (*LG.Name* + *Term.Name*). The complete test suite is shown in Appendix 5.

```
start
Locus                       locus(locus_id(Locusid),
                                  locus_name(Locusname),
                                  locus_fullname(LocusFullname),
                                  locus_species(Locusspecies),
                                  locus_type(Locustype),
                                  locus_linkagegroup(LocusLinkagegroup),
                                  locus_arm(Locusarm))
Locus.ID#                   locus_id(Locusid)
Locus.Name                  locus_name(Locusname)
Locus.Fullname              locus_fullname(LocusFullname)
Locus.Species               locus_species(Locusspecies)
Locus.Type                  locus_type(Locustype)
Locus.LinkageGroup          locus_linkagegroup(LocusLinkagegroup)
Locus.Arm                   locus_arm(Locusarm)
end
```

Figure 3: Representing MaizeDB's notion of *Locus* as a *Glossa* bundle in the testing database. Apart from line feeds and indentation added to the `locus/7` relation for legibility, this figure shows the standard format for mappings files. In *Glossa* variables are capitalized and all variables are wrapped with a functor denoting the name of the semiote or bundle. These data are needed for the select query of Figure 2.

2.4 A Testing Database from MaizeDB

To test the semantics of the *Glossa* translation of the MaizeDB queries, we implemented a small portion of MaizeDB as a declarative database in *Glossa*. This testing database contains just the data needed by the test suite and uses Prolog for its query language. Representing the table and column notions of MaizeDB in *Glossa* was a rather straight forward process as most of them fit as semiotes, the basic constructs in *Glossa*. An example is representing *Locus.Name* as `locus_name(`*LocusName*`)`. In contrast *Locus*, a much more complex notion, is a bundle of seven semiotes (abbreviated `locus/7`). Sample MaizeDB notions and the constructs denoting those notions are shown in Figure 3. `start` and `end` delimit each table's mappings. The notion on the left side of each line is denoted in MaizeDB by a table or column name and is separated by a tab from the representation in the testing database. The testing database relations were designed to faithfully reproduce the original MaizeDB data schema relations, though not necessarily the variable names. For example, *Locus* in MaizeDB is represented as `locus` with seven arguments in the testing database; but since different MaizeDB tables have columns with the same name *id*, in the testing database the representation is preceeded by the table name to make it unique: for example `locus_id` instead of `id`. The *Locus* table in MaizeDB has more columns in its definition, but since those columns are not required in any of the model queries, they were not represented in the testing database; new notions can be added as they are needed.

Once the *Glossa* mappings were constructed for all the tables, the testing database was manu-

ally populated with actual MaizeDB data. An example of a set of facts that implement the *Locus* table is shown in Figure 4.

3 Results

3.1 An Overview of the Transformation Process

Figure 5 shows the process of tranforming SQL queries to *Glossa* requests and *vice versa*. We modeled our code on commonly used cross-compiler procedures. As one would expect, the sequence of operations for an SQL $\rightarrow$ *Glossa* transformation is exactly the reverse of that for the *Glossa* $\rightarrow$ SQL transformation. Both involve two stages: a mapping stage that produces a restructured hybrid intermediate between the two languages in form ("semi-*Glossa* statement") and a parsing stage that transforms the semi-*Glossa* statement to the output language. The mapping stage uses the request and mappings described in Section 3.2 as input to the Java mapper routines. The semi-*Glossa* statement is sent to the appropriate TXL parser. This operates by rearranging the parse tree of the semi-*Glossa* statement according to the metagrammar to obtain the correct patterns in the output language. A wrapper Java class controls the total computation, taking the input and calling each step in the transformation procedure.

3.2 Mappings and Mappers

The first step in the transformation process is mapping the input request to a semi-*Glossa* statement. In this implementation, two types of substitution are made: the recursive unwinding of any nested SQL commands and the replacement of local notions with their manually defined equivalent in *Glossa*.

The input request and mappings are sent to a recursive Java class that breaks down a nested SQL statement into a simple statement and maps between *Glossa* and the local notions. Using the mappings defined in an ASCII mappings file (see Section 2.4 for sample mappings), a data structure of mappings is built on the fly and every local notion is replaced by semiotes or bundles as appropriate. An example of input to the Java SQL $\rightarrow$ *Glossa* mapper and the semi-*Glossa* output is shown in Figure 6. Each MaizeDB notion is replaced by its *Glossa* equivalent. Since the comma is used as a delimiter throughout the SQL pattern, the special table delimiters `START_OF_NEW_TABLE`, `END_OF_TABLE`, and `LAST_TABLE_END` are added to separate different tables for easier parsing of the `from` clause. The semi-*Glossa* statement is now ready to be transformed into a *Glossa* form by the SQL $\rightarrow$ *Glossa* parser.

In this particular case mapping the MaizeDB notions to *Glossa* is relatively straightforward because MaizeDB implements very elementary notions as table attributes (see Sections 2.4 and 3.2), but representing complex SQL operations is much more challenging because of the profound syntactic differences between SQL and *Glossa* and SQL's use of implicit variables. As we expected, we found we could carry out the entity transformations at the mapping stage, and used the parsing stage to transform the computational ideas of SQL.

```
locus(locus_id(12330),locus_name('hs1'),
        locus_full_name('hairy sheath1'),
        locus_species('12808'),locus_type(101),
        locus_linkage_group(13597),locus_arm(32022)).
locus(locus_id(12365),locus_name('l3'),
        locus_full_name('luteus3'),
        locus_species(12808),locus_type(101),
        locus_linkage_group(13594),locus_arm(32021)).
locus(locus_id(12366),locus_name('l4'),
        locus_full_name('luteus4'),
        locus_species(12808),locus_type(101),
        locus_linkage_group(13597),locus_arm(32022)).
locus(locus_id(12391),locus_name('lls1'),
        locus_full_name('lethal leaf spot1'),
        locus_species(12808),locus_type(101),
        locus_linkage_group(13579),locus_arm(32022)).
locus(locus_id(20057),locus_name('T2-8(003-5)(2)'),
        locus_full_name('NULL'),
        locus_species(12808),locus_type(131),
        locus_linkage_group(13582),locus_arm(32021)).
locus(locus_id(20096),locus_name('T3-10(5892)(3)'),
        locus_full_name('NULL'),
        locus_species(12808),locus_type(131),
        locus_linkage_group(13586),locus_arm(32022)).
locus(locus_id(72377),locus_name('o*-N1009'),
        locus_full_name('opaqueN1009'),
        locus_species(12808),locus_type(101),
        locus_linkage_group(13579),locus_arm(32022)).
locus(locus_id(74222),locus_name('spt*-N1620B'),
        locus_full_name('spottedN1620B'),
        locus_species(12808),locus_type(101),
        locus_linkage_group(13588),locus_arm('NULL')).
```

Figure 4: The relation `locus/7` that implements the parts of the *Locus* table of MaizeDB relevant to the query of Figure 2. Line feeds and indentation have been added for legibility.

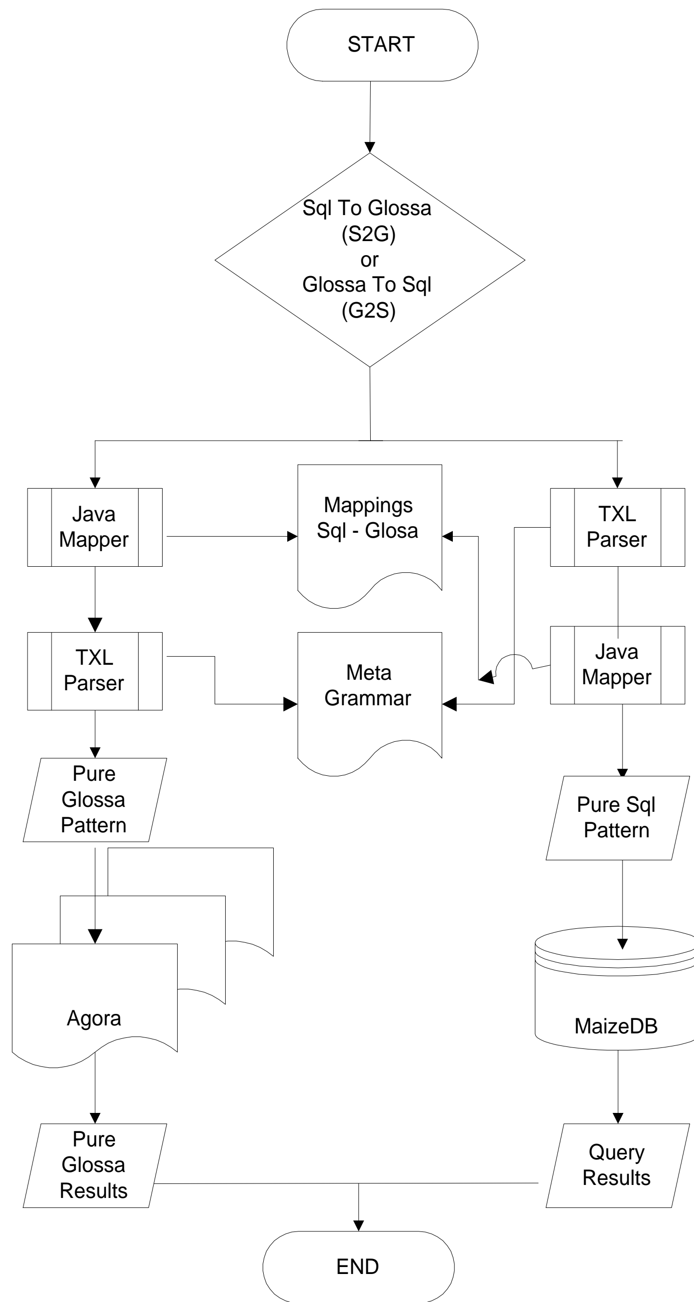

Figure 5: A flow chart of request translation from SQL → *Glossa* and *vice versa*. The mappers and parsers shown for each direction are separate modules of code. The metagrammar that is the union of the grammars of the two parsers is not shown for clarity.

Input MaizeDB query:

```
SELECT DISTINCT LC.Bin, L.ID#, L.Name, L.Fullname FrOM Locus L,
Locus#Coordinates LC WHERE LC.Bin != NULL AND
L.ID# = LC.Locus AND L.Species = 12808 AND L.Type IN
(101,120,121,122,666,40414,24621,24978,40071,61091,84832,131633)
AND LC.Bin BETWEEN 10 AND 100 AND lower(L.Name) LIKE '10' ORDER BY
lOWER(L.Name)
```

semi-*Glossa* statement:

```
select distinct locus_coordinates_bin(Locuscoordbin),
                locus_id(Locusid),
                locus_name(Locusname),
                locus_fullname(LocusFullname)

from  START_OF_NEW_TABLE locus(locus_id(Locusid),
                               locus_name(Locusname),
                               locus_fullname(LocusFullname),
                               locus_species(Locusspecies),
                               locus_type(Locustype),
                               locus_linkagegroup(LocusLinkagegroup),
                               locus_arm(Locusarm))
      END_OF_TABLE,
      START_OF_NEW_TABLE
                         locus_coordinates(
                               locus_coordinates_locus(Locuscoordlocus),
                               locus_coordinates_bin(Locuscoordbin),
                               locus_coordinates_map(Locuscoordmap))
      LAST_TABLE_END

where locus_coordinates_bin(Locuscoordbin) != null
      and locus_id(Locusid) = locus_coordinates_locus(Locuscoordlocus)
      and locus_species(Locusspecies) = 12808
      and locus_type(Locustype)
          in (101,120,121,122,666,40414,24621,24978,40071,61091,84832,131633)
      and locus_coordinates_bin(Locuscoordbin)
          between 10 and 100
      and lower(locus_name(Locusname))
          like '10'

order by lower(locus_name(Locusname))
```

Figure 6: Transformation of an SQL query to semi-*Glossa* by the SQL → *Glossa* mapper. MaizeDB notions are replaced by semiotes, variables are consistently substituted, and the query is reorganized preparatory to parsing. Notice the transformations are case-insensitive (*e. g.* the first `FrOM`). Line feeds and indentation have been added for legibility.

```
define select_query_specification
     [select_clause] [table_expression][opt sepop] [opt
     order_by_clause]
end define

define select_clause
    [select_terminal] [select_list]
end define
```

Figure 7: Grammar definition for `select_query_specification`.

3.3 Parsing

3.3.1 The Parsers's Requirements

Cross language transformation with TXL needs the following machinery:

1. a working grammar of both languages;
2. a metagrammar combining the definitions of both languages; and
3. transformation rules to gradually change a statement from one language to the other.

3.3.2 Grammar Definition and Metagrammar Development

A grammar definition in TXL is a set of `define` statements starting at the root of the grammar's tree and proceeding all the way to the leaves (the language's terminals). A typical `define` rule for the ANSI-92 SQL grammar is shown in Figure 7. The optional clause is marked by the keyword `opt`. In the above example `select_query_specification` is the root of any `select` query and it parses as [select_clause] [table_expression] [opt sepop] [opt order_by_clause]. The next statement defines the *table_expression.* These definitions are successively expanded further until the leaf nodes are completely defined. If you observe the above statement closely, in the `[opt sepop] opt` is optional, which means it may or may not be there. The same applies for the optional `order_by` clause.

An example of a TXL grammar rule for *Glossa* is shown in Figure 8. It defines the `set_of_statement` corresponding to the SQL `select` command, expanding the previously described Backus-Naur Form left-recursive context-free grammar of *Glossa* to this new operation [14, 15].

A metagrammar is produced by combining the different nodes of the individual grammars. For example, the roots of the grammars' definitions for `select/set_of_statement` are combined as in Figure 9. Now both a *Glossa* pattern and a SQL pattern can be parsed using the `select_query_specification` definition, since it has root of the *Glossa* and SQL grammars as its child nodes.

The rule of thumb when adding new rules to the metagrammar is to look at the similarity of the input and output patterns and add them under a common parent. For example, the *Locus.Name* pattern of MaizeDB is transformed to `locus_name(`*LocusName*`)` in *Glossa*. The

```
define set_of_statement
    [setop]
       [ltupdelim][select_clause][sepop][table_expression]
       [sepop][argument][rtupdelim]
       [opt sepop][opt order_by_clause]
end define

define setop
    'setof
end define
```

Figure 8: Grammar definition for set_of_statement. A right quote preceeding a term marks the term as a terminal in the language.

```
define select_query_specification
    [select_clause] [table_expression][opt sepop]
       [opt order_by_clause]
|
    [setop]
       [ltupdelim][select_clause][sepop][table_expression]
       [sepop][argument][rtupdelim]
       [opt sepop][opt order_by_clause]
end define

define table_expression
    [from_clause] [opt where_clause]
|
    [ltupdelim] [from_clause] [opt where_clause][rtupdelim]
end define
```

Figure 9: Metagrammar obtained by combining the root nodes of the SQL and *Glossa* grammars. The select_query_specification of Figure 7 and the set_of_statement of Figure 8 are combined by the alternation operator (|).

```
 redefine column_reference
   ...
 |
   [simple_statement]
end define
```

Figure 10: A redefinition of `column_reference` to include a new *Glossa* pattern.

```
rule stripOffTableTags
    replace $ [table_name]
        TabName[table_name]

        deconstruct TabName
           TabStartTag[table_start_tag] EmbdStmt[embedded_statement]
           TabEndTag[table_end_tag]
    by
        EmbdStmt
end rule
```

Figure 11: A TXL rule to remove table starting and ending tags shown in Figure 6.

SQL pattern parses as the child of a definition `column_reference` in SQL and the corresponding *Glossa* pattern parses as a `simple_statement` in *Glossa*. So the new rule added to the grammar would be that shown in Figure 10. Its `redefine` statement appends a new rule to the existing grammar definition. The ellipsis (`...`) denotes the old definition and the new definition is added as another alternative. If the old definition were to be replaced, then the `redefine` will have only the new definition. The order of the clauses in the definition is important for the efficiency of parsing. If most constructs in the language parse using definitions that occur later in the metagrammar definition of the parent node, more incorrect parses will be generated unnecessarily. When taken over the entire metagrammar, there is no consistent choice in ordering clauses for one language over the other. In effect, there must be a balance between rules that put *Glossa* clauses first and those that put SQL clauses first. The same methodology is used in defining the new grammar rules to arrive at a functional metagrammar.

3.3.3 Formulating Grammars in TXL

Pattern translation using TXL rules involves parsing the input pattern according to the grammar and rearranging the tree to obtain the target pattern. During the pattern transformation, only sibling nodes of the same parent can be interchanged with each other. A simple TXL rule to replace SQL's `and` pattern by the *Glossa* conjunction operator `,` is shown in Figure 11. The rule parses all occurences of table names in the SQL pattern, checks for the existence of table tags, and removes them. The output of the SQL $\rightarrow$ *Glossa* parser is an executable *Glossa* bundle.

```
% switch bagof/setof

bagof(( linkagegroup_id(Linkagegrpid)),
      ( linkagegroup_name(Linkagegrpname)^
        from(linkagegroup(linkagegroup_id(Linkagegrpid),
                          linkagegroup_name(Linkagegrpname),
                          linkagegroup_type(Linkagegrptype))),
        where(equal_to(linkagegroup_type(Linkagegrptype), 104))),
        ResultsList_12828 ),


bagof(( map_id(Mapid)),
      ( from(map(map_id(Mapid), map_name(Mapname))),
        where(like(map_name(Mapname), 'bins%'))),
        ResultsList_10 ),

setof(( locus_coordinates_locus(Locuscoordlocus)),
      ( from(locus_coordinates(locus_coordinates_locus(Locuscoordlocus),
             locus_coordinates_bin(Locuscoordbin),
             locus_coordinates_map(Locuscoordmap))),
       where( not_in(locus_coordinates_map(Locuscoordmap),
            ( ResultsList_10)),
              equal_to(locus_coordinates_bin(Locuscoordbin), null))),
            ResultsList_12818 )
```

Figure 12: The *Glossa* version of the query of Figure 2, part I.

3.4 Testing the Output Transformations

The transformations of the MaizeDB test queries were tested with different notions and patterns and perfectly executable *Glossa* requests were obtained. We also back-transformed the *Glossa* versions of the SQL queries and obtained semantically identical MaizeDB queries. The *Glossa* version of the query of Figure 2 generated by the SQL $\rightarrow$ *Glossa* mapper and parser is shown in Figures 12 and 13. Its back-transformation by the *Glossa* $\rightarrow$ SQL mapper and parser is shown in Figure 14. Figures 15 and 16 show the correspondence between the SQL patterns and their *Glossa* equivalents generated by the two parsers.

In a nested SQL statement, the inner-most statement is transformed into a *Glossa* pattern first. The order of execution in Prolog, our language of choice for querying the test database, proceeds from left to right and since the inner-most statements act as input to the next upper level SQL statements, their patterns need to be evaluated first. The third argument of each `setof/3` and `bagof/3` is the output result, and given a unique name to avoid name conflicts and thus

```
% existential quantifier

bagof(( locus_name(Locusname),
        linkagegroup_name_term_name_concat(LinkagegrpnameTermnameconcatinated),
        locus_id(Locusid), locus_fullname(LocusFullname) ),
      ( locus_arm(Locusarm)^
        from(locus(locus_id(Locusid),
                   locus_name(Locusname),
                   locus_fullname(LocusFullname),
                   locus_species(Locusspecies),
                   locus_type(Locustype),
                   locus_linkagegroup(LocusLinkagegroup),
                   locus_arm(Locusarm))),

        where( join(join_type(right),left_column(term_id(Termid)),
                   right_column(locus_arm(Locusarm)),
                   table(term(term_id(Termid),term_name(Termname))),
                   output_required(term_name(Termname))),
               linkagegroup_type(Linkagegrptype)^
               join(join_type(right),left_column(linkagegroup_id(Linkagegrpid)),
               right_column(locus_linkagegroup(LocusLinkagegroup)),
               table(linkagegroup(linkagegroup_id(Linkagegrpid),
               linkagegroup_name(Linkagegrpname),
               linkagegroup_type(Linkagegrptype))),
               output_required(linkagegroup_name(Linkagegrpname))),
               equal_to(locus_species(Locusspecies), 12808),
               not_equal_to(locus_type(Locustype), 131),
               in(locus_id(Locusid),(ResultsList_12818)),
               like(lower(locus_name(Locusname)), '[0-9a-pr-z]%'),
               in(locus_linkagegroup(LocusLinkagegroup),(ResultsList_12828)),
               concat(linkagegroup_name(Linkagegrpname),
                     term_name(Termname),
                     linkagegroup_name_term_name_concat(
                      LinkagegrpnameTermnameconcatinated)))),
               OutPutList),

        order_by([( key_element(locus_name(Locusname)),
                    sort_order(default),
                    additional_conditions(none))],
                 OutPutList,SortedList )
```

Figure 13: The *Glossa* version of the query of Figure 2, part II.

```
select Locus.Name , LinkageGroup.Name + Term.Name , Locus.ID# , Locus.Fullname
from Locus , LinkageGroup , Term
where Locus.Species = 12808 and Locus.Type != 131
and Locus.ID# in( select distinct Locus#Coordinates.Locus  from Locus#Coordinates
                  where Locus#Coordinates.Map not in( select Map.ID#  from Map
                                                      where Map.Name like 'bins%' )
                  and Locus#Coordinates.Bin = null )
                  and lower(Locus.Name ) like '[0-9a-pr-z]%'
                  and Locus.LinkageGroup in( select LinkageGroup.ID#
                                             from LinkageGroup
                                             where LinkageGroup.Type = 104 )
and Term.ID# =* Locus.Arm and LinkageGroup.ID# =* Locus.LinkageGroup
order by Locus.Name
```

Figure 14: Sql pattern generated *Glossa* → SQL mapper and parser. The shortened representation of table names in Figure 2 are replaced by full tablenames.

incorrect results in a nested SQL query. In Figures 12 and 13 the output of the first `bagof/3`, ResultsList_10, occurs in immediately following `setof/3` and is an input to the next predicate. A `DISTINCT` clause in the SQL statement, which eliminates any duplicate results in the output results, transforms to a `setof/3` *Glossa* pattern; otherwise they transform to a `bagof/3`.

A `right join` pattern such as *Term.ID#* `=*` *Arm*, fetches the required results from the *Term* table if there is a *Term ID* equivalent to the *Locus Arm*. If no such *ID* is found, then all the *Term* results required in the output for this particular tuple are set to null. The *Glossa* `right join` pattern is built to represent exactly the same idea.

How difficult is it to make the mapper and parser work for a new set of mappings? New mappings can be added and existing mappings can be modified simply by editing the ASCII text file containing mappings in a text editor. Suppose the new set of mappings identified for the SQL pattern `foo.bar` is `foo_bar(FooBar)` in *Glossa*; then the new addition to the mappings in order for the mapper to accept them as legal input are shown in the Figure 17. The substitution shown is correctly parsed by both parsers (data not shown).

4 Discussion

This paper implements a general methodology to establish interoperability among different participating systems using the formal language *Glossa*. The method does not depend on a rigid data model or on schema(ta) coupling, but on basic axiomatic notions and a context-free grammar that defines the semantics of more complex ideas. A change of schema or semantics of a participating system can be easily acounted for by defining new semiotes and/or changing the mappings.

The need for general transformations that efficiently ground their nonterminals militates against using TXL to transform requests in a single step. Because TXL backtracks upon failure

```
1.  select ID# from Map where Name like 'bins%'  ===

bagof((map_id(Mapid)),
      (from(map(map_id(Mapid), map_name(Mapname))),
       where(like(map_name(Mapname), 'bins%'))),
      ResultsList_10)                              ===

 select Map.ID#  from Map  where Map.Name like 'bins%'



2.  select distinct Locus from Locus#Coordinates
                   where Map
                   not in (select ID# from Map where Name like 'bins%')
                   and Bin = NULL   ===

setof((locus_coordinates_locus(Locuscoordlocus)),
      (from(locus_coordinates(locus_coordinates_locus(Locuscoordlocus),
            locus_coordinates_bin(Locuscoordbin),
    locus_coordinates_map(Locuscoordmap))),
       where(not_in(locus_coordinates_map(
       Locuscoordmap),(ResultsList_10)),
             equal_to(locus_coordinates_bin(Locuscoordbin), null))),
             ResultsList_12818
     )                            ====

select distinct Locus#Coordinates.Locus  from Locus#Coordinates
 where Locus#Coordinates.Map not in (select Map.ID#  from Map
            where Map.Name like 'bins%')
 and Locus#Coordinates.Bin = null
```

Figure 15: Equivalences between the SQL and *Glossa* statements, Part I. For each of four SQL statements, the corresponding *Glossa* statements follow after the ===.

```
3.  LG.ID# =* LinkageGroup ===

join(join_type(right),right_column(locus_arm(Locusarm)),
     bagof(term_id(Termid),term_name(TermName)^term(term_id(Termid),term_name(TermName)),
           Termidlist),
     output_required(term_name(Termname))
    )                       ===
LinkageGroup.ID# =* Locus.LinkageGroup


4.  Term.ID# =* Arm ===

join(join_type(right),right_column(locus_linkagegroup(LocusLinkagegroup)),
     bagof(linkagegroup_id(Linkagegrpid),
           linkagegroup_name(Linkgrpname)^linkagegroup_type(Linkgrptype)^
           linkagegroup(linkagegroup_id(Linkagegrpid),linkagegroup_name(Linkgrpname),
                        linkagegroup_type(LinkagegrpType)),Linkagegrpidlist),
     output_required(linkagegroup_name(Linkagegrpname),
                     linkagegroup_type(Linkagegrptype))
     )              ===


Term.ID# =* Locus.Arm
```

Figure 16: Equivalences between the SQL and *Glossa* statements, part II.

```
start
foo.bar                    foo_bar(FooBar)
end
```

Figure 17: An example of adding new mappings.

```
select distinct convert(numeric(8,2),LC.Bin), L.ID#, L.Name, L.Fullname
From Locus L, Locus#Coordinates LC
where LC.Bin != NULL  and L.ID# = LC.Locus
      and L.Species = 12808
      and L.Type
          in (101,120,122,666,40414,24621,24978,40071,61091,84832,131633)
      and LC.Bin
          between 2 and 8
      and lower(L.Name) like '$PS'
order by lower(L.Name)
```

```
locus_map(species(maize),
      locus_name(c734),
      locus_type_list([101,120,122,666,40414,24621,24978,40071,61091,84832,131633]),
      bin_lower_limit(2),
      bin_upper_limit(8),
      format_bin_value(required(yes),precision(2)),
      sort_in(AscOrDesc),ResultSet),
      order_by(lower(locus_name(LocusName),ResultSet,SortedSet).
```

Figure 18: Naïve representation of a MaizeDB query. The upper panel shows a sample query and the lower panel a less efficient representation of that query in *Glossa*.

of a rule, parsing of complex SQL queries requires many nearly identical TXL rules and can produce inconveniently large parse trees before the required transformation is obtained. So the transformation is split into two stages, first replacing mappings using the Java mappers, and second formulating the output pattern using a TXL parser. Since the *Glossa* version of `right join` and nested `select`s require extensive rearrangement of the query, they are implemented in mappings too. This lets us exploit the efficient string handling functions of Java. To build the parser stage using TXL, as we have done here, one must identify the metagrammar and define new rules for transforming the patterns. We did this by separately defining TXL grammars for both SQL `select` commands and *Glossa* and then combining these to form the metagrammar.

Defining mappings between a system and *Glossa* requires attention to both the notions of the system and the general pattern of the system's query or interface language. For example, we started out by trying to represent a MaizeDB `select` query in *Glossa* by mapping the query's notions in a very finely grained manner. Naïvely, one might represent the query shown in the upper panel of Figure 18 as shown in the lower panel. The difficulty is that this *Glossa* pattern is valid only for this query. Any variation of at least one condition in the query's `select` statement would need a completely different *Glossa* pattern. More coarsely grained bundles that have a more generalized structure usually give a better solution. Changing or augmenting the mappings is trivial: the mappings text file can be revised without affecting any other part of the system.

The platform independence of Java, use of an ASCII file for mappings, and availability of

TXL for different operating systems, lets our code work successfully on different platforms (Solaris, Linux, and Windows). It would be trivial to substitute XML for ASCII in the mappings file. The next implementation of the parsers will be done by converting a Backus-Naur Form specification of the grammars with standard tools like Flex, Bison, JLex, and CUP to further improve platform independence and simplify installation [1,3]. This will also avoid execution delays that arise when calling an external program and TXL's limitations on the number of characters in the input pattern (1024 on Solaris).

The input queries in this implementation required the output from a single database. However, in a real world scenario where a number of databases communicate with each other, the input information request might need information from different data sources. In this case the input request must go through a central hub that distributes pieces of the request to the appropriate data sources, receives the results as pure *Glossa*, packages them, and returns the result to the requesting entity. We are currently developing the appropriate tools for this scenario.

We demonstrate a general procedure that any participating system can follow to exchange data and computations in *Glossa*. The participating system needs to determine if its ideas can be expressed with the existing set of semiotes, or define new semiotes that have the required formal properties [14, 15]. Once the semiotes are identified, mappings are constructed between the local notions and *Glossa*. Not all notions in a database need mappings to *Glossa*: a database curator actively chooses which information to share. In our test case, the mappings were defined for just the model queries. We did not consider integrity constraints within MaizeDB because these are an exclusive function of the local system. If the system uses SQL, no further effort is required unless its semantics changes. Otherwise, parsers that translate between *Glossa* and the local query language must be built. Once built, they can be used by all other databases that use that query language. The result is a general set of tools and methods for the interoperability of independent, heterogeneous systems over the World-Wide Web.

Acknowledgments

We thank Mary Polacco, Ed Coe, and Denis Hancock for giving us access to MaizeDB, helping with query selection, and patiently explaining the semantics and biology of MaizeDB. We thank Avanthi Mummaneni and Bhavani Akunuri for helping with bibliographic processing. This work is supported by a grant from the U. S. National Institutes of Health (GM-56529) to T. K.

Bibliography

1. Appel, A. W., 1998. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge.

2. Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., Harris, M. A., Hill, D. P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J. C., Richardson, J. E., Ringwald, M., Rubin, G. M., and G. Sherlock, 2000. Gene Ontology: tool for the unification of biology. *Nature Genet.* **25**:25–29.

3. Berk, E. and C. S. Ananian, 2000–present. *JLex: a Lexical Analyzer Generator for Java.* Princeton University, `http://www.cs.princeton.edu/ appel/modern/java/JLex/`.

4. BioMOBY.org, 2003–present. *BioMOBY.org.* BioMOBY.org, `http://www.biomoby.org/`.

5. Chen, I.-M. A., Kosky, A. S., Markowitz, V. M., and E. Szeto, 1997. Constructing and maintaining scientific database views. In *Proceedings of the Ninth Conference on Scientific and Statistical Database Management*, pages 237–248. IEEE Computer Society Press, Rockville MD.

6. Date, C. J., 2000. *An Introduction to Database Systems.* Addison-Wesley Publishing Co., Reading MA, seventh edition.

7. Elmagarmid, A., Rusinkiewicz, M., and A. Sheth, eds., 1999. *Management of Heterogeneous and Autonomous Database Systems.* Morgan Kaufmann, San Francisco.

8. European Bioninformatics Institute, 2001–present. *InterPro.* European Bioninformatics Institute, `http://www.ebi.ac.uk/interpro/`.

9. Genome Data Base Staff, 1993. *Online Mendelian Inheritance in Man.* `http://gdbwww.-gdb.org/omimdoc/omimtop.html`.

10. Harold, E. R. and W. S. Means, 2002. XML *in a Nutshell.* O'Reilly and Associates, Inc., Sebastopol CA, second edition.

11. International Union of Biochemistry and Molecular Biology, 1992. *Enzyme Nomenclature. Recommendations (1992) of the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology.* Academic Press, Inc., London.

12. Kahn, R. and R. Wilensky, 1995. *A Framework for Distributed Digital Object Services.* Corporation for National Research Initiatives, `http://www.cnri.reston.va.us/home/cstr/-arch/k-w.html`.

13. Kazic, T., 1994–present. *Moirai: Modeling Biochemistry.* University of Missouri, Columbia, MO, `http://www.biocheminfo.org/moirai/`.

14. Kazic, T., 2000. Semiotes — a semantics for sharing. *Bioinformatics* **16**:1129–1144. Also at `http://www.biocheminfo.org/repository/semiotes.ps`.

15. Kazic, T., Jiang, J., Kutikkad, G., Yao, G., Bugrim, A., and J. Slomczynski, 2000–present. Glossa *Semiotes.* University of Missouri, Columbia, MO, `http://www.the-agora.org/-glossa/semiote_list.ps`.

16. Kazic, T., Lusk, E., Olson, R., Overbeek, R. A., and S. Tuecke, 1990. Prototyping databases in Prolog. In Sterling, L., ed., *The Practice of Prolog*, pages 1–29. MIT Press, Cambridge MA.

17. Letovsky, S., 1996. *MIMBD 96: Java and CORBA for Bioinformatics.* GDB, `http://info.gdb.org/̃letovsky/jcws.html`.

18. Library of Medicine, N., 1999–present. *National Library of Medicine Fact Sheet: UMLS Semantic Network.* National Library of Medicine, http://www.nlm.noh.gov/pubs/factsheets/umlssemn.html.

19. Microarray Gene Expression Data Society, 2002 – present. *MGED Home.* Microarray Gene Expression Data Society, `http://www.mged.org/`.

20. Naqvi, S. and S. Tsur, 1989. *$\mathcal{L}$DL: A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville MD.

21. National Center for Biotechnology Information, 1995. *GenBank.* National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov/GenBank/index.html`.

22. Polacco, M. L. and J. Edward E. Coe, 1999. MaizeDB: the Maize Genome Database. In Letovsky, S., ed., *Bioinformatics: Databases and Systems*, pages 151–162. Kluwer Academic Publishers, Boston MA.

23. Previato, E., ed., 2002. *A Dictionary of Applied Math for Scientists and Engineers.* CRC Press, Inc., Boca Raton, FL.

24. Princeton Softech, 2000. Database interoperability – eliminating DBMS barriers. Technical report, Princeton Softech, Princeton NJ.

25. Research Collaboratory for Structural Biology, 1995–present. *Protein Data Bank.* Research Collaboratory for Structural Biology, `http://www.rcsb.org/pdb/`.

26. Rishe, N. D., Athuada, R. I., Yuan, J., and S. ching Chen, 2000. Knowledge Management for Database Interoperability. *ISCA 2nd International Conference On Information Reuse And Integration (IRI-2000), November 1-3, 2000,Honolulu, Hawaii, USA.* pages 23–26. Also at `http://citeseer.nj.nec.com/374595.html`.

27. Sheth, A. P. and J. A. Larson, 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comp. Surv.* **22**:183–236.

28. Sidman, K. E., George, D. G., Barker, W. C., and L. T. Hunt, 1995–present. *PIR: The Protein Identification Resource.* National Biomedical Research Foundation, `http://www.-gdb.org/Dan/proteins/pir.html`.

29. Snell, J., Tidwell, D., and P. Kulchenko, 2002. *Programming Web Services with SOAP.* O'Reilly and Associates, Inc., Sebastopol CA.

30. Sun Microsystems, Inc., 1996. *[Java].* Sun Microsystems, Inc., `http://java.sun.com/`.

31. Swedish Institute of Computer Science, 1999–present. *SICS Quintus Prolog Manual.* Swedish Institute of Computer Science, `http://www.sics.se/isl/quintus/qp/frame.html`.

32. TXL Software Research, Inc., 2002 – present. *The TXL Programming Language.* TXL Software Research, Inc., `http://www.txl.ca/`.

5 Appendix: Suite of Test MaizeDB Queries

1. **locus_qtl Query:**

```
select   distinct convert(numeric(8,2),LC.Bin), L.ID#, L.Name, L.Fullname

From     Locus L, Locus#Coordinates LC

where    LC.Bin != NULL
          and L.ID# = LC.Locus
          and L.Species = 12808
          and L.Type in (25396)
          and LC.Bin between 2 and 8

order by lower(L.Name)
```

2. **locus_probe Query:**

```
select     distinct convert(numeric(8,2),LC.Bin), L.ID#, L.Name,
L.Fullname

From       Locus L, Locus#Coordinates LC

where      LC.Bin != NULL
           and L.ID# = LC.Locus
           and L.Species = 12808
           and L.Type in (113)
           and LC.Bin between 2 and 8
           and lower(L.Name) like 'a%'

order by lower(L.Name)



locus_gene Query:
select     distinct LC.Bin, L.ID#, L.Name, L.Fullname
From       Locus L, Locus#Coordinates LC
where      LC.Bin != NULL
           and L.ID# = LC.Locus
           and L.Species = 12808
           and L.Type in
       (101,120,121,122,666,40414,24621,24978,40071,61091,84832,131633)
           and LC.Bin between 2 and 8
           and lower(L.Name) like 'a%'
```

```
order by lower(L.Name)"
```

3. **locus_other Query:**

```
select    LG.Name + T2.Name, L.ID#, L.Name, L.Fullname
from      Locus L, LinkageGroup LG, Term T2

where     L.Species = 12808
          and L.Type != 131
          and L.ID# in (select distinct Locus from Locus#Coordinates
                     where Map not in (select ID# from Map
                                       where Name like 'bins%')
                      and Bin=NULL)
          and LG.ID# =* LinkageGroup
          and T2.ID# =* Arm
          and LOWER(L.Name) like '[0-9a-pr-z]%'
          and LinkageGroup in (select ID# from LinkageGroup
                           where Type = 104)
order by L.Name
```

4. **Maps_probes_alphabetical Query:**

```
select   PL.Bins,PL.Probe,PL.Name,substring(T.Name,1,30),
         convert(numeric(8,2),PL.IBM), convert(numeric(8,2),PL.UMC),
         convert(numeric(8,2),PL.BNL), convert(numeric(8,2),PL.Pio),
         PL.GenBank,PL.ZmDB,PL.TIGR, convert(numeric(8,2),PL.IBM),
         PL.URL_Suffix

from     Probe#Links PL, Probe P, Term T

where    P.ID#=PL.Probe and P.Type=T.ID#
          and lower(PL.Name) like 'a%'
          and PL.Bin between 5 and 9

order by PL.Name
```

5. **Maps_probe_maporder Query:**

```
select   PL.Bins,PL.Probe,PL.Name, substring(T.Name,1,30),
         convert(numeric(8,2),PL.IBM),convert(numeric(8,2),PL.UMC),
         convert(numeric(8,2),PL.BNL),convert(numeric(8,2),PL.Pio),
         PL.GenBank,PL.ZmDB,PL.TIGR,PL.URL_Suffix,PL.Locus,PL.LName,
         PL.CUGI, convert(numeric(8,2),PL.IBM)
```

```
from     Probe#Links PL, Probe P, Term T

where    P.ID#=Probe and P.Type=T.ID# and PL.Bin between 5 and 9

order by PL.Bin desc, convert(numeric(8,2),PL.IBM) desc,
         convert(numeric(8,2),PL.Pio) desc
```