A Source-to-Source Meta-Translation System for Relational Query Languages

D.I.Howells, N.J.Fiddian, W.A.Gray

**Department of Computing Mathematics,
University College Cardiff, Cardiff, Wales.**

Acknowledgement: This work has been undertaken while the first author has been supported by an S.E.R.C. postgraduate studentship.

Abstract

With the expanding use of database management systems and the rapid rate of change in computer technology, particularly the advent of more powerful workstations, together with standard local and wide area networks, there is an increasing need for the ability to access data distributed across different databases. This can mean that users need to learn several different query languages in order to be able to manipulate their data satisfactorily if it is held in several distinct databases. Recent work in heterogeneous distributed databases [1,2] has shown that it is possible to translate automatically queries posed in the query language of one database system into the query language of another database system. At Cardiff we have collaborated with researchers at other U.K. universities in the development of the PROTEUS [2] heterogeneous distributed database system. This work involved developing a number of separate translators, by traditional methods, which translate queries between different source query languages by means of a common intermediate query language (called NQL [2,3] in PROTEUS).

From the experience gained in writing the PROTEUS translators, we concluded that it should be possible to automate the process of query language translator production, at least for the general family of relational query languages. This theory has been tested by creating a meta-translation system, implemented in PROLOG, which accepts specifications of any relational query language as a new input and output language for the system, respectively, and from these specifications derives translators between the new language and every other relational query language already known to the system. The source-to-source translation between query languages is carried out via a common intermediate tree representation, which is able to represent any possible query from any relational query language. This paper outlines the design and operation of this translation system, assesses its worth and briefly discusses some of its potential applications.

1 Introduction

The increasing power of micro/mini computers has meant that a greater number of dispersed database user sites can afford local computers which have sufficient power and storage to meet many of their local requirements without recourse to a central mainframe installation. Parallel developments in computer networks have laid a sound foundation for the reliable transfer of information between computers: either locally, over local area networks, or remotely, over wide area networks. These advances have provided a firm platform for the development of 'distributed' databases and encouraged research into ways of establishing them. At present the distribution of data may occur in four basic types of model: homogeneous or heterogeneous variants of distributed or federated databases.

In heterogeneous systems [4] there is a need to translate queries posed in one database's native query language into other query languages. It is clear from the growing popularity of relational systems, particularly on microcomputers, that in any future heterogeneous system there will be a potential mix of constituent micro, mini and mainframe computers running a variety of relational systems. Such a composite system will have several different groups of users, each group being familiar with and preferring to use a different query language. These preferences can conveniently be accomodated by developing source-to-source relational query language translators.

Relational databases built on micro computers often expand to the point where they can no longer efficiently exist in their entirety on such a machine. At this point the database can be transferred *in toto* to a relational system on a larger, more powerful computer, thereby losing the advantages of local user control. Alternatively, to minimise this drawback, the most frequently used data may be retained locally and just the residue be exported. In either case the change should ideally be as transparent as possible to the user. However, the larger machine, crucially, may not have available the particular database system used on the micro computer. A suitable relational query language inter-translator can be used to make good this deficiency.

A large organisation may have established a database under an existing relational system, but want to access the data through a new relational system. Here again, an appropriate translator can be used to give transparent access to the data.

Thus it can be seen that query language translators are in increasing demand. As an example, the recent PROTEUS research project [2] to develop a heterogeneous distributed database system involved the production of some dozen translators by a number of research groups, including ourselves, working in parallel over a period of two to three years. The translators concerned were developed separately by traditional methods to translate between different source query languages via a common, intermediate, internal Network Query Language (NQL [2,3]). Successful as this collaborative effort was, it was clear that a substantially more productive way of developing such translators needed to be found for the future. Fortunately the PROTEUS project itself provided valuable pointers to the way forward : from our experience of the translators produced we concluded that it should be possible to automate the production process, at least for the family of relational query languages. Thus the idea of a meta-translation system for this purpose was conceived. The following sections consider in turn the architecture, operation, value and application of the resultant system that we have created, beginning with a discussion of productive methods and means of translator construction which have influenced our choice of software technology for the meta-translation system.

2 Productive Methods and Means of Translator Construction

A translation system consists of three interconnected main parts: a 'scanner' which does lexical analysis, a parser which does syntax analysis and a code-generator to perform semantic and synthetic actions. The approaches traditionally used to write translation systems are:

(i) to write a hand-coded 'scanner' and 'parser' in a suitable high level language. This is not difficult - for example, the parser can normally be produced in a straightforward way by modelling its structure directly on the grammar of the subject language concerned - but it is nevertheless a relatively lengthy and time-consuming exercise, typically measured in man-months.

(ii) to use a scanner generator (e.g. LEX [5]) and an associated parser generator (e.g. YACC [6]) program to produce the scanner and parser translator components automatically. The time to carry out this exercise can be measured typically in man-weeks.

It should be noted that in each of the above cases the parser per se has to be augmented with program fragments written in a high level language (e.g. C [7] in the case of YACC) defining the translator action that is to be taken upon recognition of each syntactic construct. This is a non-trivial development task which involves programming expertise and takes an appreciable amount of time in its own right. This time is included in the above estimates.

Construction of translators by method (ii) involves substantially less coding effort and therefore takes a much shorter time than by method (i). It is also far easier to modify a translator using the second approach. This approach is therefore the best and the favoured traditional method from the programmer productivity point of view.

As a result of conversations with P.M.D.Gray [8] about his production of a simulator [9] written in PROLOG [10] for the ASTRID database system [11], we decided to investigate the use of PROLOG for the development of relational query language translators (this application of Prolog has also been advocated by other researchers; see for example the book "A Prolog Database System" by D. Li). Although new, this approach is essentially equivalent to an integrated form of method (ii). In an initial investigation of PROLOG in this role [12] we developed a translator between DBASEII [13] and QUEL [14]. This was a relatively rapid process with the translator being encoded in a matter of man-days. This demonstrated that PROLOG is appreciably higher level in its support for source-to-source translator generation than the common LEX-YACC combination and we feel that it is currently one of the most appropriate vehicles for this purpose. On the basis of our experience with this translator we could see that PROLOG would be a most productive and flexible language in which to implement the design of our proposed meta- translation system. This has indeed proved to be the case, as the following sections demonstrate.

3 The Architecture and Operation of the Meta-Translation System

3.1 Overview of the System

The system, illustrated in main Figure 3.1 and supporting Figure 3.2, has at its heart the meta-translator module (see Figure 3.1). This has access to a database schema and a number of translation schemes applicable between relational query languages and a common internal relational algebra tree representation (c.f. Figure 3.2). The meta-translator is linked to only one input query language and one output query language translation scheme at a time. E.g. If RQLi --> Tree and Tree --> RQLj are the current input and output query language translation schemes, respectively, an RQLi to RQLj translation is implied. Relational query language to internal tree translation schemes are specified via the system's meta-language interface (a complementary help module is also available to give information on meta-language constructs). Internal tree to relational query language translation schemes are specified via an interactive forms-based interface.

In operation, the system has two distinct stages, as Figure 3.1 indicates. The first, optional, stage accepts specifications of new relational query languages as either input (RQL --> Tree) or output (Tree --> RQL) query languages or both, as required. The second stage is application of a selected pair of input and output translation schemes (e.g. RQLi --> Tree, Tree --> RQLj) to produce a corresponding source-to-source translation between the two query languages concerned (RQLi --> RQLj).

3.2 Components of the System

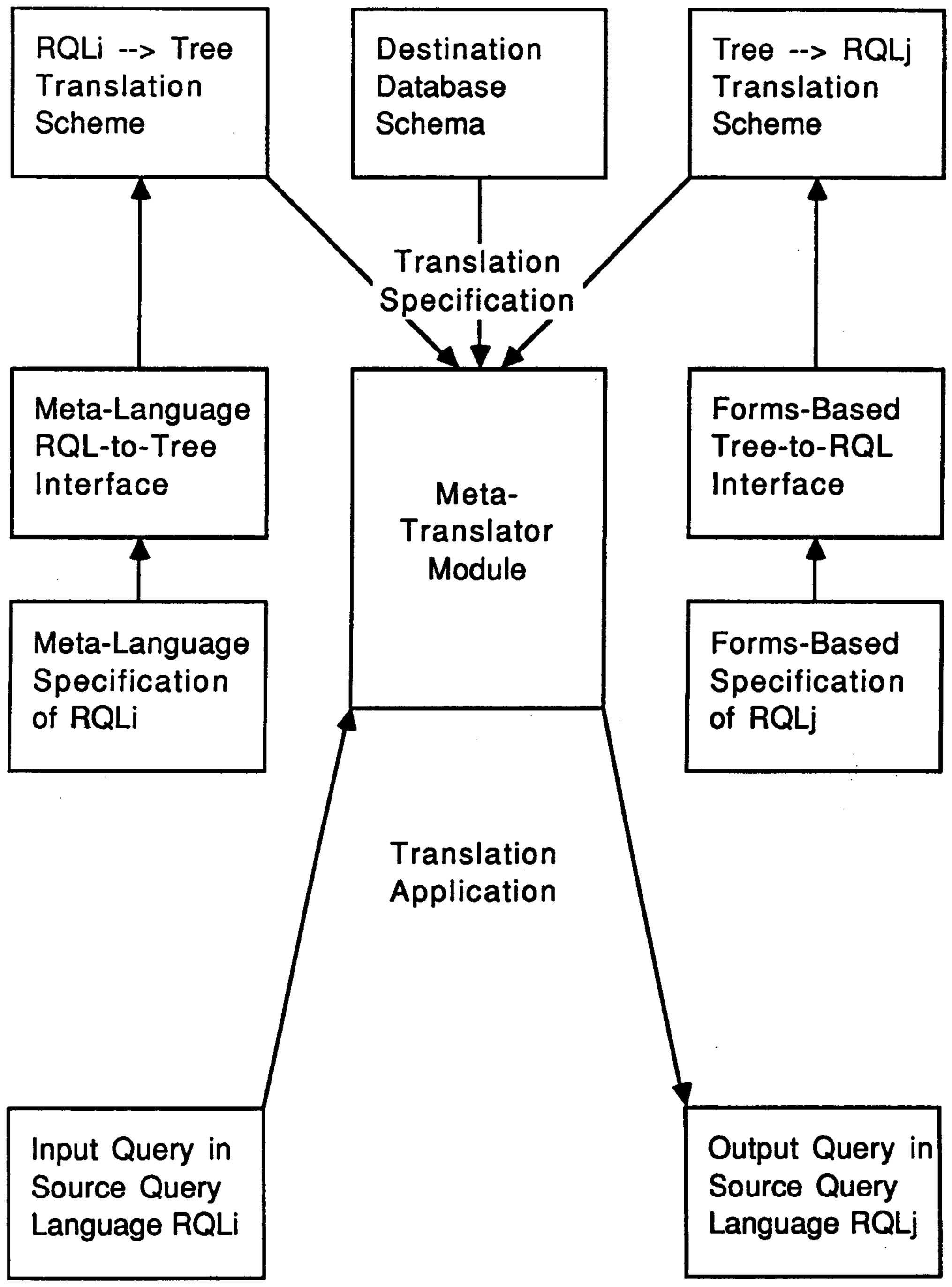
3.2.1 The Internal Database Schema

This is a simple but adequate schema consisting of a PROLOG clause for each relation within either the associated database schema of the destination (i.e. output) query language or the global schema of the system, as appropriate. A typical example is :-

```
schema(parts,[pnum],[pname+str,pnum+int,pweight+int]).
```

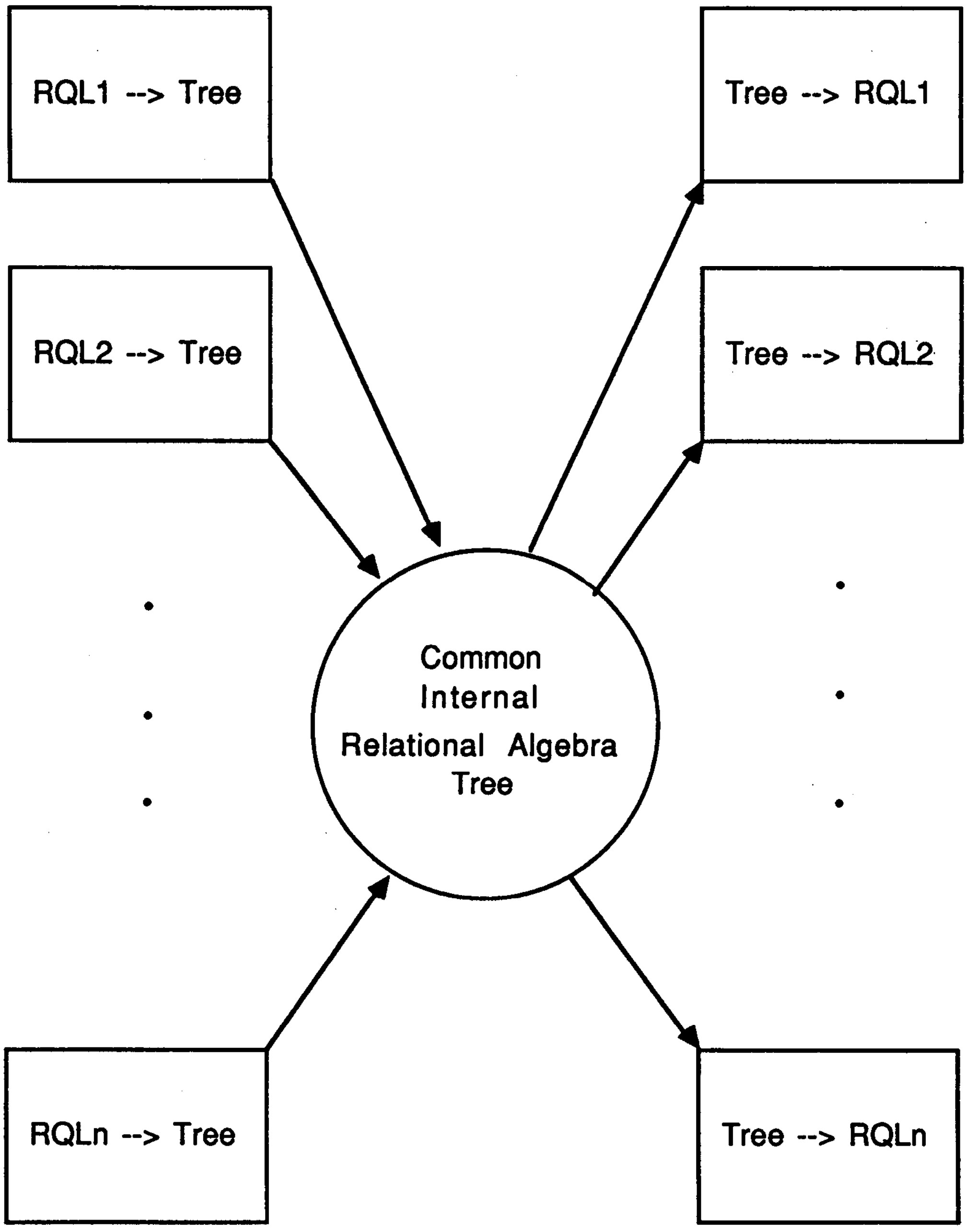
This shows the presence of the relation parts in the schema. This relation has the key pnum and attributes pname, pnum and pweight of respective types str(ing), int(eger) and int(eger).

Temporary relations are also held within the overall schema. A typical example is :-


RQLi --> Tree Translation Scheme
Destination Database Schema
Tree --> RQLj Translation Scheme
Translation Specification
Meta-Language RQL-to-Tree Interface
Meta-Translator Module
Forms-Based Tree-to-RQL Interface
Meta-Language Specification of RQLi
Forms-Based Specification of RQLj
Translation Application
Input Query in Source Query Language RQLi
Output Query in Source Query Language RQLj

An Overview of the System

Figure 3.1



Multiple Translation Schemes
Figure 3.2

```
temp_schema(res1,Tree).
```

Here Tree stands for the internal tree representation of the query which generates the intermediate relation called res1 (see next subsection for a detailed description of the internal tree form).

The system can be directed to work either from a specific schema for a particular destination database nominated at translation time or from a permanently resident global schema.

3.2.2 The Internal Relational Algebra Tree

When translating between source query languages it is normal to use a common intermediate internal query language (e.g. NQL in PROTEUS [2,3]). In developing a meta-translation system for generalised source-to-source translation between any pair of relational query languages, we felt that an approach based on an intermediate language was essential and chose a tree-structured language based on a relational algebra tree. A typical example of an SQL [15] query and its tree representation is shown in Figure 3.3. The tree is represented in a linear internal intermediate language form as follows :-

```
proj(sel(rel(parts,[[pnum],[pname+str,pnum+int,pweight+int]]),
exp(gt(pnum,5)),[[pnum],[pname+str,pnum+int,pweight+int]]),
attrs(pname,pnum),[[pnum],[pname+str,pnum+int]])
```

This corresponds to a preorder traversal of the tree in which each tree node has as its first argument an input relation and as its last argument a schema-like specification of the relation resulting from the application of the relational algebra function of the node. Thus internal trees are totally self- contained and can be translated from without reference to the associated schema. Hence in specifying an output language translation scheme (see 3.2.4 below) there is no need to reference the schema.

3.2.3 The Meta-Language for Specifying Relational Query Language to Tree Translations

To facilitate the translation of a relational query language to the internal tree form, a semantic meta-language was developed. Input language translation schemes are specified by entering the syntax of the input query language in a PROLOG-compatible BNF form. This is augmented by meta-language constructs to specify the semantics of the language. The complete language specification is entered in a specification file. If necessary, extra PROLOG clauses particular to an input translation scheme can be added to the associated specification file and used as required. A typical extract for INGRES's query language QUEL [14] is :-

```
line(R2) --> [retrieve] , resrel(R) , ['('] , proj_list(Pl) , [')'] ,
join_list(S) ,
EC { ( inp_rels(L) ,
EC get_inp_rels(L,Rels) ,
ML mk_join_node(Rels,S,N1) ,
ML mk_proj_extn_grp_node(N1,Pl,[],inc,inc,N2) ,
ML mk_rel(N2,R) ,
PR abolish(inp_rels,1) ,
PR asserta(inprels([])) ) } .
```

Here, for expositional clarity, in the semantic actions section extra clause invocations are labelled by the prefix EC, meta- language construct applications by ML and PROLOG clause invocations by PR. Inp_rels and get_inp_rels jointly produce a list of the names (Rels) of the relations used in the QUEL query. These are passed, along with the join and selection expressions (S), to mk_join_node, which produces the corresponding relational algebra tree N1. Mk_proj_extn_grp_node takes as its arguments N1, the projection list Pl (which may include extension and grouping expressions) and the attributes that are being grouped by (in this case []). It produces from these the corresponding relational algebra tree N2 consisting of a collection of projection, extension and group nodes. The two inc arguments state that the extension and grouping expressions must be included as projection attributes in the projection node of this tree. Mk_rel takes as its arguments a tree (N2) and a result relation name (R). If the result relation name is blank then the tree is stored as a result tree clause. Otherwise the named intermediate result relation is stored in the temporary schema.

Only one tree is produced. This may consist of a number of subtrees that are intermediate relations (which have been stored in the temporary schema).

Within the system environment, help about the semantic meta-language constructs can be requested by typing in a construct name. An explanation of the function of the cited construct, the input its expects and the output it produces is then provided.

3.2.4 The Forms-Based Interface for Specifying Tree to Relational Query Language Translations

A scheme for translating from the internal tree form to a relational query language is specified by following a set of prompts which enumerate all possible single relational algebra nodes that can appear in a tree. The user is asked if each node exists in the output query language in question. If it does, the system calls an editor so that the equivalent statement in the output query language can be specified. A typical example for the INGRES query language QUEL is the following relational algebra selection node template :-

```
sel(rel(Rname,[Hr|Tr]),exp(Sexp),[Hs|Ts]) into New_relname
```

which has the equivalent QUEL specification :-

```
nl,
write('retrieve into '),
write(New_relname),
write(' ('),
write(Rname),
write('.all)'),
nl, write('where '),
gen_sel_out_monadic(Rname,Sexp,Sexp2),
write_list(Sexp2).
```

In the tree form template, upper case words are variable names. The corresponding output is specified by write statements (outputting text, or the values associated with variables) interspersed with newlines (nl). If values associated with variables are not in the correct format for output a number of reformatting functions may be called. For example, in the above output specification for QUEL, the clause gen_sel_out_monadic, given the relation operand of the selection function and the associated selection expression from the tree, generates a corresponding output form selection list with each attribute preceded by the relation name Rname and '.'. The selection expression is held internally in prefix form without reference to relations, e.g. and(eq(pname,cog),gt(pnum,1)). Gen_sel_out_monadic automatically translates this into standard infix form and replaces operator mnemonics by standard symbols, e.g. = for eq. If this is not exactly what is required, the construct gen_sel_out_monadic_var allows the user to enter alternative specific relation names, separators and expression operator symbols.

Many relational database query languages allow users to perform more than one relational algebra operation in a single statement. A common example of this is the use of projection and selection together. So, after all the information on single nodes has been entered, the system invites the user to enter information about composite nodes. These are entered explicitly by the user in preorder sequence, viz :-

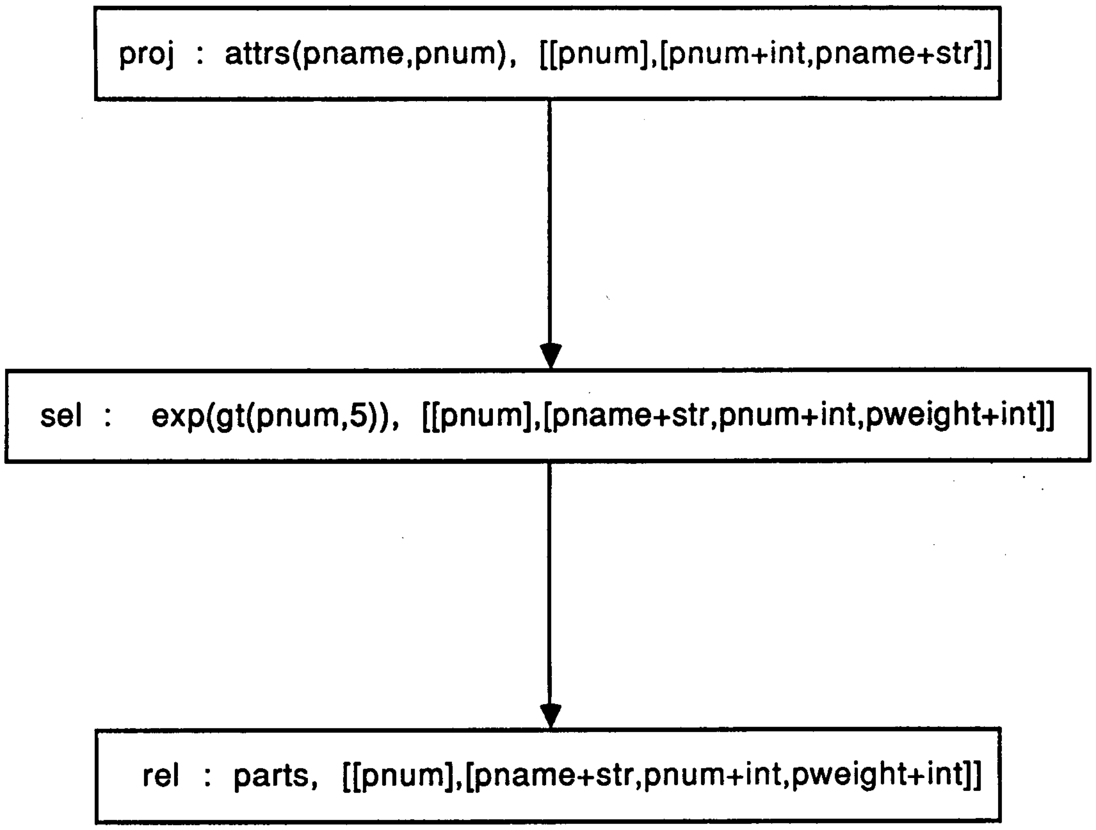
```
proj sel
```

The system responds as before, by placing the user in an editor with a node template, this time for the given composite node, at the top of the screen :-

```
proj(sel(rel(Rname,[Hr|Tr]),exp(Sexp),[Hs|Ts]), attrs(Pattrs),[Hp|Tp])
into New_relname
```

The user is then able to specify the equivalent output query language statement in the way shown previously.

Composite nodes of arbitrary complexity may be entered in the same manner, for example :-


proj : attrs(pname,pnum), [[pnum],[pnum+int,pname+str]]
sel : exp(gt(pnum,5)), [[pnum],[pname+str,pnum+int,pweight+int]]
rel : parts, [[pnum],[pname+str,pnum+int,pweight+int]]

A Relational Algebra Tree

```
Select pname pnum
From parts
Where pnum > 5
```

The Equivalent SQL Query

```
schema(parts, [pnum], [pname+str, pnum+int, pweight+int])
```

The Associated Target Schema

Figure 3.3

proj sel thjoin sel rel sel rel

is a useful combination of operations which can be specified as a composite entity in translation schemes for many output query languages.

3.3 Using the System for Translations

The meta-translator is table-driven, so initially the user is prompted to indicate the tables needed for the particular translation required. The following information is requested and entered :-

(i) Input query language name.
(ii) Name of the file containing the input source query.
(iii) Output query language name.
(iv) Destination database name.
(v) Name of file to receive the internal tree representation of the source query (not essential, but useful for debugging purposes).
(vi) Name of file to receive the output source query.

The following tables are then read in by the system (c.f. Figure 3.1) :-

(i) Input query language to tree translation scheme.
(ii) Destination database schema.
(iii) Tree to output query language translation scheme.

The current query is then translated and the current tables are kept resident within the system environment. More queries can then be presented for translation in the same way, different translations can be selected or the system can be exited.

The user interface has been designed in this way primarily for presentation purposes. Obviously, manual entry of the input query language name, etc. could be obviated. Also, fixed translators could be built using pre-selected input and output translation schemes.

In the process of translation, an input query may generate a tree that cannot be represented completely in the output query language (i.e. the input query language has some operator(s) not supported by the output language). In this case the meta- translator translates as much of the tree as possible to the output query language and the remaining part of the tree is either translated to a default query language (whose database system can complete the remaining part of the query) or, if desired, to the original input query language. In the latter case it is foreseen that, in a distributed or federated database context, as much of the query concerned as possible would be carried out at the output language database site. The input language site would receive the resulting intermediate relations and the residual part of the original query (in its own language) and complete the processing of the query against the returned intermediate data.

4 Evaluation of the Meta-Translation System

4.1 System Performance

Input and output translator pairs for DBASEII, QUEL, a line-oriented QBE and SQL have been successfully developed using the system. These represent all the available types of relational query language, namely relational algebra, tuple-based relational calculus, domain-based relational calculus and relational algebra-calculus hybrids, respectively. The translators were very straightforward to produce, taking a short time and little effort. All were successfully tested on a comprehensive set of 34 sample queries.

4.1.1 Performance Measurements

To develop a translator from a relational query language to the internal tree form took no more than a few days (at most a week). The specification input required varied between 50 and 120 PROLOG clauses.

To develop a translator from the internal tree form to a relational query language, specified via the forms-based interface, took a very short time indeed. This was typically less than a day. The tree to DBASEII translator took just over an hour to produce (including seven composite nodes)!

The translators produced were efficient. A typical complex query took approximately five seconds to be translated between any pair of source query languages using an interpretive version of Prolog (Edinburgh New Implementation of Prolog (NIP) on a VAX 8200).

4.2 Advantages of Using the System

The system is designed to be used by a person with a working knowledge of relational algebra, BNF, the relational query languages for which translators are required and a minimal amount of PROLOG. There is very little difference in the required level of expertise between such a person and a person who would write translators in a traditional manner. The advantages of using the system are summarised as follows :-

(1) Translators can be very easily and quickly developed (see typical productivity figures above).
(2) Relatively little input is needed to specify translators.
(3) The translation schemes produced are very manageable and easily maintainable. This is due to the conciseness of these schemes, their high level of abstraction and their applicative nature.
(4) The system produces space and time-efficient translators. Translation schemes are compact and typical queries are translated in a few seconds.
(5) Translation input is parsed in a top-down manner. This is a more readily understandable approach than bottom-up methods, such as that used by YACC. This fact greatly facilitates the process of debugging translators during their development.
(6) It is easy to learn to use the system. The relational query language to tree semantic meta-language has only 17 simple constructs and often only a subset of these is needed. The tree to relational query language forms-based interface is extremely easy to use and is quickly learnt.
(7) The meta-translator produces efficient translations since the tree is built in as optimal a fashion as possible and composite nodes within the tree can be recognised. These composite nodes can be of arbitrary size.

Consequently, if a query language is translated directly into the same language, the resulting queries will be at least as efficient as their original equivalents. Many database systems only optimise subqueries, not entire queries. The meta-translator produces a single tree to represent an entire query. This tree may be made up of the subtrees (stored in the temporary schema) that represent subqueries. So, for example, if a naive user of a query language believed that he could not project and select in one query, he might formulate his query as two subqueries, the first to select and the second to project. The meta-translator would produce a single tree consisting of a project and a select node. This tree could then be recognised as a composite entity and an output query be generated which combines the two operations. This principle extends to composite nodes of arbitrary complexity.

5 Applications of the Meta-Translation System

In this section we briefly survey some of the applications we foresee or have already prototyped for this translation system. It can be used in heterogeneous distributed and federated database systems as a tool for translating queries between different relational query languages; to produce front-ends for existing relational database systems to enable users wanting the facilities of non-native relational query languages to access those systems; as a tool to optimise queries in any relational database system; and as a teaching tool for helping a person familiar with one relational query language to learn another.

5.1 A Query Translation Tool in Heterogeneous Distributed and Federated Databases

The system, as a tool, can provide translators between query languages in a more efficient and cost-effective manner than traditional approaches. It will make heterogeneous database systems more flexible as enhancement with new query languages will be made significantly simpler. Using the system we have already developed translators for all of the major types of relational query language. These were developed in a short time and tested thoroughly on a large sample set of queries covering all types of relational query operation, considered both individually and in combination.

5.2 Front-Ending an Existing Database System

When users want to have the facilities of non-native query languages as alternative front-ends to an existing database, the system could be used to produce appropriate translators. This approach could be employed to save users having to learn a new query language when they change jobs and have to access a new database managed by an unfamiliar system, or to ease the introduction of a new, more powerful database system into an existing database context.

5.3 Query Optimisation Applications

The system could also be useful as an aid to optimising queries by recognition of composite nodes - see 4.2 (7) above. This requirement could occur in a centralised, distributed or federated database environment. In distributed or federated environments the system could be linked to a global query optimiser and transaction manager so that the internal tree may be broken into subqueries for each destination site before subquery translation into the relevant query language for each site concerned takes place.

5.4 To Teach a New Query Language (Learning by Example)

The system could be used in teaching a new query language to a user who already knows one but wants to learn another. The user could type in a query in the language he knows about, and the equivalent query would be output in the language being learnt. This process could be repeated for a series of queries of increasing complexity until the new language is mastered.

6 References

[1] Smith, J.M. et al, 'Multibase - integrating heterogeneous distributed database systems', Proc. AFIPS Conference, 50, pp. 487-499, 1981.
[2] Stocker, P.M. et al, 'PROTEUS: A Heterogeneous Distributed Database Project', In: 'Databases - Role and Structure', pp. 125-151, Cambridge, 1984.
[3] Fiddian N.J. et al, 'First PROTEUS queries successfully transmitted', University Computing, 6, pp. 177-182, 1984.
[4] Ceri, S. and Pelagetti, P., 'Distributed Databases, Principles and Systems', McGraw Hill, 1984.
[5] Lesk, M.E., 'LEX - A Lexical Analyser Generator', Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.
[6] Johnson, S.C., 'YACC - Yet Another Compiler Compiler', Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
[7] Kernighan, B.W. and Ritchie, D.M., 'The C Programming Language', Prentice Hall, 1978.
[8] Gray, P.M.D. (Department of Computer Science, University of Aberdeen), Private communication, 1986.
[9] Gray, P.M.D. (Department of Computer Science, University of Aberdeen), Program Communication, 1986.
[10] Clocksin, W.F. and Mellish, C.S., 'Programming in Prolog', Springer-Verlag, 1981.
[11] Gray, P.M.D., 'The ASTRID System for Access to Codasyl Databases', IUCC Bulletin, 4, pp. 70-76, 1982.
[12] Howells, D.I., Fiddian, N.J. and Gray, W.A., 'A Comparison of Old and New Technologies for Translating Between Relational Query Languages', Proceedings of the Third International Workshop on Statistical and Scientific Database Management, Luxembourg, pp. 179-183, July 1986.
[13] dBASEII User Manual, Ashton-Tate, 1984.
[14] Stonebraker, M., Wong, E. and Kreps, P., 'The Design and Implementation of INGRES', ACM Transactions on Database Systems, 1, pp. 189-222, 1976.
[15] Chamberlin, D.D. et al, 'SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control', IBM Journal of Research and Development, 20, pp. 560-575, 1976.