Relational Database Support for Complex Objects Defined by Grammars

by R. Cochrane and L. Mark
Relational Database Support for Complex Objects Defined by Grammars *

Robert Cochrane    Leo Mark

Department of Computer Science
and Systems Research Center
University of Maryland
College Park, Maryland 20742
leo@cs.umd.edu, bobbie@cs.umd.edu

Abstract

Context-free grammars provide the basis for many useful tools such as parser-generators, compiler-compilers and syntax-directed editors. This paper demonstrates the potential benefits obtained when context-free grammars are used to define complex objects in the relational model. The grammar formalism facilitates relational queries on the hierarchical structure of these objects and promotes the use of grammar-based tools as front ends to relational database systems.

Keywords: Advanced Applications, Data Models, Query Languages, Complex Objects

1 Introduction

Several research projects ([BK85,CNR90,Lin84,LKM+85,RY85,Row89], among others) provide relational database support for complex objects such as engineering part descriptions, text, and software programs. The schema for these projects is defined using the relational schema definition language and, due to the nature of complex objects, the data of a given object is often distributed over several relations. These relational schema definitions do not indicate how the relational data is derived from the original objects or how it can be combined to reconstruct the original objects. This information must be reflected in data extraction tools and queries that are written by the user. Furthermore, the resulting relations only store aspects about the objects that are of interest to the current application. This poses a problem for query evolution: As the application evolves and new queries arise, additional information about the objects (which was most likely part of the original specification) must be gathered and stored in the database. The data extraction tools must be constantly rewritten to reflect these changes.

*This research was sponsored partially by the National Science Foundation under Grant IRI-8719458, by the Air Force Office for Scientific Research under Grant AFOSR-89-0303, and by the University of Maryland Institute of Advanced Computer Studies.
Grammars have been described as a useful representation for data structures [GT83] and hierarchical structures in information [GT87]. A formal description of a grammatical database model is described in [GPV89]. However, they do not discuss how this model is realized in a relational database system.

This paper proposes one translation of grammatically defined complex objects into relational schema and demonstrates how the grammar definitions form a basis for tools that support the population and manipulation of the database. Section 2 formally describes a grammatical schema definition language and gives an algorithm, GeneRel, that translates the grammatical schema to relational schema under which these objects are stored. Section 3 describes a tool-generator, GeneParse, that generates parsers that populate the database, illustrating the potential for building tool-generators whose products support database operations for these objects. Support for a grammar catalog, described in section 4, is obtained by applying GeneParse to a grammar that defines the structure of the input grammars (the meta-grammar). GeneRel and GeneParse are implemented as semantic actions of the meta-grammar, and the implementation of the grammar catalog is generated by applying these tools to the meta-grammar itself. Section 5 discusses the issue of querying these objects, introducing three graph operators that facilitate the manipulation of hierarchical structures. The computation of each operator is contained in the fixed point of a set of queries that are derivable from the grammar specifications. Finally, Section 6 discusses issues of query evolution that are related to efficiency.

2 Grammatical Schema Definition Language

The grammatical schema definition language we derive, tagged context-free grammars (TCFGs), is a variant of context-free grammars that incorporates the concept of tokens, adapts the closure notations from regular expressions, and includes tag-names for uniquely identifying symbols in the grammar.

Tokens facilitate the grouping of terminal characters into single entities [HU79]. TCFGs have two classes of tokens: delimiters — tokens whose domain consists of a single value, and lexicons — tokens whose domain consists of more than one value. Furthermore, lexicons and nonterminals are referred to collectively as nondelimiter symbols. A lexical analyzer that returns tokens and their values must accompany each grammar.

Closure notation, used in regular expressions, is convenient for representing repeating structures. Kleene closure applied to a symbol (i.e. \( x^* \)) represents all strings that are a concatenation of zero or more occurrences of strings derivable from the symbol (i.e. \( \emptyset, x, xx, x^n \)); positive closure applied to a symbol (i.e. \( x^+ \)) represents the same set of strings as kleene clo-
sure minus the null string \( \emptyset \). We incorporate this notation into TCFGs because it encourages the utilization of the set retrieval aspects of the relational query languages.

Tag-names, inspired by [MN88], allow the user to specify meaningful names for the generated relations and attributes. All occurrences of nondelimiter symbols (i.e. nonterminals and lexicons) are tagged.\(^1\)

**Definition 1** A tagged context-free grammar (TCFG) is a 7-tuple \( E = (S, V, L, D, R, A, P) \) where \( V \) is a finite set of nonterminals; \( L \) is a finite set of lexicons; \( D \) is a finite set of delimiters; \( R \) is a set of production tag-names; \( A \) is a set of non-delimiter tag-names; \( V \), \( L \), and \( D \) are disjoint; \( S \in V \) is the special start symbol; and \( P \) is a set of productions. Productions are classified as either constructors or lists. A constructor production has the form:

\[
\langle r : N \rangle \rightarrow w_1 \langle a_1 : N_1 \rangle \ldots w_k \langle a_k : N_k \rangle w_{k+1},
\]

and a list production has the form:

\[
\langle r : N \rangle \rightarrow \langle a_i : N_i \rangle^\square
\]

where \( r \) is a production tag-name, \( N \) is a nonterminal, \( w_i \) is a possibly empty string of delimiters, \( a_i \) is a non-delimiter tag-name, \( N_i \) is a nonterminal or lexicon, and \( \square \) is either * or +.

**Example 1** Mechanical engineers wish to provide database support for information about parts in a manufacturing resource planning (MRP) system [HY88]. The MRP system keeps a part master record for each part which contains the part number, textual specification, and other auxiliary data such as unit of measure and leadtime. Parts are either purchased or manufactured. Information about the supplying vendor and cost is recorded for each part that is purchased. A bill of material is kept for each manufactured part, which contains the quantity of each subpart that is required to manufacture the part. The following TCFG captures this information:

\[
\begin{align*}
\langle \text{pmr:part} \rangle & \rightarrow \langle \text{p#:int} \rangle \langle \text{descr: str} \rangle \langle \text{uom:int} \rangle \\
& \quad \langle \text{leadtime:int} \rangle \langle \text{partType:type} \rangle \\
\langle \text{vendor:type} \rangle & \rightarrow \langle \text{vendorName:str} \rangle \langle \text{cost:int} \rangle \\
\langle \text{bom:type} \rangle & \rightarrow \langle \text{subpartQty:subpQty} \rangle^+ \\
\langle \text{subparts:subpQty} \rangle & \rightarrow \langle \text{subpart:part} \rangle \langle \text{qty:int} \rangle
\end{align*}
\]

This example contains four productions with production tag-names \( \text{pmr}, \text{vendor}, \text{bom}, \) and \( \text{subparts} \) for the three nonterminals \( \text{part}, \text{type}, \) and \( \text{subpartQt}\). Notice that the production tag-names \( \text{vendor} \) and \( \text{bom} \) differentiate between \( \text{type} \) information for purchased and

\(^1\)Note that this restriction need not be inflicted on the user; we have built an automatic tagger that generates TCFGs from YACC grammar specifications.
manufactured parts respectively. There is a non-delimiter tag-name for each occurrence of a nonterminal or lexicon on the right-side of the productions. There must be a lexical analyzer that returns tokens for lexicons `int` and `str` and their corresponding values, and would also be responsible for returning tokens for delimiters if there were any. This TCFG is recursive, since a manufactured part's bill of material must contain at least one `subpart` (indicated by positive closure) which is, in turn, a part.

*GeneRel* is an algorithm that translates a TCFG into relational schema definitions.

**Definition 2** A *relational schema definition* has the form:

```
create r(a_1 : d_1 [not null],...,a_k : d_k [not null])
```

where `r` is the *relation name*, the `a_i`'s are *attribute names* (which are unique within the definition), the `d_i`'s define the *domains* of their corresponding attributes, and `not null` is a string which is attached to each attribute that participates in the key of the relation.

Each nonterminal and lexicon symbol in the TCFG has a corresponding domain. The domain `N` contains a surrogate\(^2\) for each derivation of the nonterminal `N` stored in the database. The domain `L` represents the syntactic category defined by the lexicon `L`.

One relation scheme is generated for each production in the TCFG. The form of the relation schemes generated for the two types of productions is similar and is summarized in Figure 1. Each generated relation inherits its name from the production tag-name of the corresponding production. It has one attribute for each non-delimiter symbol on the right-side of the production; the attribute's name is the same as the non-delimiter's tag-name and the attribute's domain is a domain that corresponds to the non-delimiter symbol. Each relation has an attribute named `occur` which is defined over the domain that corresponds to the left-side nonterminal of the production. Relations representing list productions have an additional position attribute that indicates the order between elements in the same list. The *key* for relations generated from constructor productions consists of the single attribute `occur`; the *key* for relations generated from list productions consists of the attribute pair (`occur, position`).

**Example 2** The relational schema definitions generated from the TCFG in Example 1 are:

---

\(^2\)We assume the relational model ([Cod79]) extended with domains of *surrogates* as described in [HOT76]. Surrogates are system generated internal identifiers that are ideal for representing unnamed objects.
<table>
<thead>
<tr>
<th>Production</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td></td>
</tr>
<tr>
<td>$r : N \rightarrow w_1 \langle a_1 : N_1 \rangle \ldots w_k \langle a_k : N_k \rangle w_{k+1}$</td>
<td>$r(occur : N, a_1 : N_1, \ldots, a_k : N_k)$</td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>$r : N \rightarrow \langle a_1 : N_1 \rangle \Box$</td>
<td>$r(occur : N, a_1 : N_1, position : counter)$</td>
</tr>
</tbody>
</table>

Figure 1: GeneRel

create pmr(occur:part not null, p#:int, descr:str, uom:int, leadtime:int, partType:type)
create vendor(occur:type not null, vendorName:str, cost:int)
create bom(occur:type not null, subpartQty:subpQty, position:counter not null)
create subparts(occur:subpQty not null, subpart:part, qty:int)

There are four productions in Example 1, so four relations are generated. There is a domain of surrogates (part, type, subpQty) corresponding to each of the three nonterminals, and the lexical domains int and str contain all values that are in the syntactic category returned by the lexical analyzer for the corresponding lexicons. The relation bom has a position attribute since it is generated from a list production.

3 Database Population

Context-free grammars provide the basis for many extremely useful tools such as parser-generators, compiler-compilers and editing environments. For instance, we have designed and implemented a tool, GeneParse, that generates parsers that populate the database. This tool is appropriate in applications, such as programming languages, in which objects are defined by context-free grammars and are written as sentences of the grammar.

GeneParse generates one parser, parser+, for each TCFG. Sentences accepted by the TCFG can then be parsed and stored under the TCFG’s corresponding relational schema. Each production in the TCFG is translated into an equivalent production in YACC[Joh78]. The translation for constructor productions is straightforward – tags are removed and the proper delimiters are used. The list productions generate two YACC productions: one is left recursive and the other is either a single symbol (for +) or the empty string (for *). In addition, GeneParse generates semantic actions that insert the sentences into the database when the corresponding productions are recognized.
begin
X := 3
if X == 4 then
  X := 5
else
  if X == 3 then
    X := 4
  else
    X := 3
  endif
endif
end

Figure 2: GeneParse: Stores Data into Relations

Example 3 Software engineers are interested in providing database support for programming languages, whose structures are often formally specified by grammars. The following TCFG defines a simplified structured programming language.

\[
\begin{align*}
\langle \text{prog:block} \rangle & \rightarrow \text{begin} \langle \text{body:stmtlist} \rangle \text{ end} \\
\langle \text{stmts:stmtlist} \rangle & \rightarrow \langle \text{stmt:stmt} \rangle^* \\
\langle \text{ifstmt:stmt} \rangle & \rightarrow \text{if} \langle \text{bool:cond} \rangle \\
& \quad \text{then} \langle \text{trueact:stmtlist} \rangle \text{ endif} \\
& \quad \text{else} \langle \text{falseact:stmtlist} \rangle \text{ endif} \\
\langle \text{assign:stmt} \rangle & \rightarrow \langle \text{var:id} \rangle := \langle \text{value:int} \rangle \\
\langle \text{equal:cond} \rangle & \rightarrow \langle \text{var:id} \rangle == \langle \text{value:int} \rangle
\end{align*}
\]

Figure 2 shows a program that is in the language of this TCFG and how it is stored under the corresponding relations by the parser generated by GeneParse. Note that for this example we show surrogates that would not normally be exposed to the user.

4 The Grammar Catalog

The TCFGs and their sentences correspond to the schema and data levels in the intension-extension framework for DBMSs presented in [Mar85] and [MR87]. In this framework, a

\[^3\text{[O'C90] depicts the tasks that are associated with the usage of context-free grammars in a similar framework.}\]
relational schema is the intension of the (extensional) data. TCFGs and their sentences are coupled to this framework using GeneRel and GeneParse (figure 3).

The middle level of this framework is the level of database definition where the user specifies TCFGs. GeneRel generates the corresponding relational schemes, and GeneParse generates the corresponding parser+ which is employed at the lowest level to store the sentences in the database. Additionally, the specified TCFGs are parsed and stored in a grammar catalog. This supports, as is often necessary, access to structural information about the data stored in the database.

In our implementation, we support the middle level of the framework having only implemented the top level. Support for the grammar catalog was generated by applying GeneRel and GeneParse to a TCFG (the meta-grammar) that describes the class of TCFGs. GeneRel was applied to the meta-grammar to produce a set of relation schemes under which any
TCFG - including the meta-grammar itself - can be stored. *GeneParse* was applied to the meta-grammar to produce the parser\(^*\) in the middle level for storing the TCFGs in the database.

5 Queries

The following examples demonstrate that several queries involving complex objects that are described by TCFGs can be expressed in current relational languages.\(^4\)

Example 4 What is the level 1 explosion (i.e. the immediate sub-parts) of the manufactured part p1?

\[
\begin{align*}
\text{select} & \text{ sub.p}\# \\
\text{from} & \text{ super as pmr, bom, subparts, sub as pmr} \\
\text{where} & \text{ super.p}\# = p1 \\
\text{and} & \text{ super.partType = bom.occur} \\
\text{and} & \text{ bom.subpartQty = subparts.occur} \\
\text{and} & \text{ subparts.subpart = sub.occur};
\end{align*}
\]

Example 5 Which variables have a value of 4 assigned to them?

\[
\begin{align*}
\text{select} & \text{ var} \\
\text{from} & \text{ assign} \\
\text{where} & \text{ value = 4};
\end{align*}
\]

Example 6 What are all the occurrences of statements?\(^5\)

\[
\begin{align*}
(\text{select} & \text{ occur from ifstmt}) \\
\text{union} & \\
(\text{select} & \text{ occur from assign});
\end{align*}
\]

However, queries on complex objects frequently involve some form of recursion and, therefore, cannot be described by standard relational queries. Several research efforts extend the expressiveness of relational languages with limited forms of recursion such as transitive closure \([Agr88]\), linear recursion \([JAN87]\), path algebra \([DS86, Car78]\), and traversal recursion \([RHDM86]\). A survey of extensions to query languages to support graph traversal appears in \([MS90]\). However, none of these extensions allow the user to express recursion that involves multiple relations. This type of recursion is important for complex objects since the data of a given object is often distributed over several relations.

\(^4\)The examples in this section are based on the example grammars in Section 2 and Section 3.

\(^5\)Although surrogates cannot be printed, they may be the intermediate results of queries.
The operators introduced in this section help the user express queries about sentences that correspond to TCFGs. They can be combined with relational expressions to form queries that associate information from multiple sentences even when these sentences are derived from different TCFGs. The operators can be expressed as a system of simultaneous relational queries (e.g. a set of queries whose fixed point contains the desired result) that are generated from the TCFG at the same time the relations are defined.

5.1 Graph Operators - Semantics

The semantics of the graph operators presented in this section is described in terms of a conceptual directed acyclic graph (DAG) that corresponds to the set of complex objects stored in the relations. The operator \textit{derives}, based on the notion of $\Rightarrow$ from language theory [HU79], reconstructs the lexical information (as defined by the TCFG) for surrogates that represent the nodes in the DAG. The operators \textit{reach} and \textit{contain} facilitate the extrapolation of parent-sibling relationships from graphs that span several relations. They are based on the concepts of reach and inverse reach from graph theory: Suppose $G$ is a directed graph and there is a path from node $j$ to node $k$ in $G$; then $k$ is in the \textit{reach} of $j$ and $j$ is in the \textit{inverse reach} of $k$.

The graph operators are defined as follows:

Let $S$ be a unary relation with one attribute of surrogate values denoting nodes; let $N$ be a set of production tag-names representing node types.

\textit{derives}($S$) is a binary relation with attributes \textit{node} and \textit{sentence} representing the mapping between nodes in $S$ and their derived sentences.

\textit{reach}($N$, $S$) is a unary relation with attribute \textit{occur} that consists of the surrogates for all nodes that are in the reach of at least one of the nodes represented by the surrogates in $S$ and have one of the node types in $N$. If $N$ is not specified, then all node types are considered.

\textit{contain}($N$, $S$) is a unary relation with attribute \textit{occur} consisting of the surrogates for all nodes that are in the inverse reach of at least one of the nodes represented by the surrogates in $S$ and have one of the node types in $N$. If $N$ is not specified, then all node types are considered.

The queries in this paper are written in the Starburst query language [HCL+90] which, among other features, enhances SQL with table expressions [Dat84] and table functions.

\footnote{A given object corresponds to a parse tree, but if sharing of sub-objects is supported, the set of objects corresponds to a DAG.}
Table functions are used to express the graph operators, and table expressions are used to bind subqueries as input to the operators. A table expression, \texttt{var as query}, binds the variable \texttt{var} to the subquery \texttt{query}. A table function, \texttt{var as tf(p1,...,pn)}, binds the variable \texttt{var} to the table produced by the function \texttt{tf} which takes zero or more input parameters. Any of these parameters can be tables. Table expressions and table functions are listed in the from clause of a query, and the variables are treated as table names of the referenced tables for the duration of the query. An example of an implementation that uses table functions to extend the query language is described in [WCL91].

\textbf{Example 7} Find the part numbers of all purchased parts that are needed in the manufactured part p1.

\begin{verbatim}
select r.occurs
from i as
  (select occurs
   from pmr
   where p#=p1)
  r as reach({VENDOR}, i);
\end{verbatim}

\textbf{Example 8} Find the part numbers of all the parts that will be delayed if supplies of the purchased part p7 are delayed.

\begin{verbatim}
select pmr.p#
from pmr
where pmr.occurs in
  (select c.occurs
   from i as
     (select pmr.occurs
      from pmr
      where pmr.p#=p7)
     c as contain({pmr}, i));
\end{verbatim}

\textbf{Example 9} All the sentences derived from the nonterminal STMT are obtained with the following query:

\begin{verbatim}
select d.sentence
from i as ( (select occurs from ifstmt)
    union
    (select occurs from assign)
  ), d as derives(i);
\end{verbatim}

\textbf{Example 10} The surrogates of all statements that contain an assignment of the value 3 to a variable can be expressed using the \texttt{contain} operator.
select c.occu
from i as (select occur from assign where value = 3),
  c as contain({stmt}, i);

Example 11 The results of contain and reach operations can be combined with other
operators to further enhance the query. Let the result of the above query be the view P.
All conditions of if statements that contain the assignment of the value 3 to a variable can
be retrieved with the following query:

select ifstmt.cond
from ifstmt, P
where ifstmt.occu = P.occu;

Example 12 The surrogates of all assignment statements that are embedded in if-statements
that have a condition on the variable X can be retrieved with the following query:

select r.occu
from i as (select ifstmt.occu
  from equal
    where var = "X"),
  r as reach({assign}, i);

5.2 Graph Operators - Computation

A computation for each operator is derived from the TCFGs. During schema definition, a
set of SQL queries is constructed from the TCFG for each operator. An expression involving
a graph operator is evaluated by computing the fixed point of the set of queries (for that
operator) applied to the given input parameter. The construction of the set of queries for
computing reach and contain is similar, so only the computation of reach is given here.
The reach set of queries has one query for each production. The query, \( K_r \), generated for production with tag-name \( K \) has the following form:

```sql
select * from K
where occur in
  (select I.occurs from I
   union
   select \{rhstag_i\} from \{prodtag_i\}_r
   ...
   union
   select \{rhstag_n\} from \{prodtag_n\}_r )
```

where \( I \) is the relation to which the operation is applied, and for each right-side occurrence \( i \) of the nonterminal for production \( K \), \( prodtag_i \) is the tag-name of the production containing this occurrence and \( rhstag_i \) is the non-delimiter tag-name of \( i \). To evaluate \( \text{reach}(\{K_1, \ldots, K_n\}, E) \) where \( K_i \) is production tag-name from the TCFG, the fixed point of this set of queries is computed with \( E \) substituted for \( I \), and the relation \( \{K_1\}_r \) union \( \{K_2\}_r \), \ldots, union \( \{K_n\}_r \) is returned.

**Example 13** Suppose we have the following grammar:

```plaintext
<A1:A>  →  <b1:B>  <c1:C>
<A2:A>  →  <d1:D>*
<E1:E>  →  <b2:B>  <a1:A>
<F1:F>  →  <c2:C>  <a2:A>
```

The following queries are generated for the reach operator:

\( A_1_r = \text{select * from } A_1 \)

where occur in

\( (\text{select occur from } I \)

union

\( \text{select a1 from } E_1_r \)

union

\( \text{select a2 from } F_1_r ) \)

\( A_2_r = \text{select * from } A_2 \)

where occur in

\( (\text{select I.occurs from } I \)

union

\( \text{select a1 from } E_1_r \)

union

\( \text{select a2 from } F_1_r ) \)
\[ E_{1r} = (\text{select } \ast \text{ from } E_1 \text{ where occur in select I.\text{occur from I}}) \]

\[ F_{1r} = (\text{select } \ast \text{ from } F_1 \text{ where occur in select I.\text{occur from I}}) \]

To evaluate \( \text{reach}(\{A1, A2\}, E) \), compute the fixed point of the \( \text{reach} \) system of queries with \( E \) substituted for \( I \), and return \( A1_r \cup A2_r \).  

The \( \text{derives} \) set of queries is not as natural to the relational model as that of the other operators because of its dependence on the position information in the list rules. It uses the aggregate operator \( \text{max} \) and the string concatenation operator (\( || \)). The \( \text{reach} \) operator is used to restrict the computation of derivations to only those objects that are components of objects specified in the input relation, \( I \).

The set of queries has one \( \text{nonterminal} \) query for each nonterminal and one \( \text{list} \) query for each list production. A nonterminal query for nonterminal \( N \) generates the temporary relation \( N_{\text{str}}(\text{occur, str}) \), where \( \text{str} \) is the derivation of the object represented by \( \text{occur} \). \( N_{\text{str}} \) is a union of subqueries, one for each production for \( N \). The form of the subqueries is dependent on the type of the production. A list query for a list production with production tag name \( r \) generates the temporary relation \( r_{\text{lst}}(\text{occur, position, str}) \) where \( \text{str} \) is the concatenation of the derivations for all surrogates including and following the \( \text{position}^t \) element in the list.

Recall the production forms from section 2. Let

- \( \{< v_1 : V_1 >, \ldots, < v_n : V_n >\} \) for \( n < k \) represent all the \( < a_x : N_x > \) that are non-terminal symbols.
- \( \{< l_1 : L_1 >, \ldots, < l_m : L_m >\} \) for \( m < k \) represent all the \( < a_x : N_x > \) that are lexicon symbols.
- \( a_x\text{str} \) represent the string value for \( < a_x : N_x > \), where
  - \( a_x\text{str} = N_x\text{str}.\text{str} \) if \( N_x \) is a nonterminal, and
  - \( a_x\text{str} = r.a_x \) if \( N_x \) is a lexicon.
- \( \text{reach0f} = \text{select } r.\text{occur from } i \text{ as } I, r \text{ as } \text{reach}(i) \).

Subqueries for constructor rules where \( n > 0 \) have the form:

\[
\text{select } r.\text{occur}, v_1 \ || \ a_1\text{str} \ || \ \ldots \ || \ a_k\text{str} \ || \ v_{k+1}
\text{ from } r, r_1 \text{ as } V_{1\text{str}}, \ldots, r_n \text{ as } V_{n\text{str}}
\text{ where } r.v_1 = r.1.\text{occur} \text{ and } \ldots \text{ and } r.v_n = r_n.\text{occur};
\]
Subqueries for constructor rules where \( n = 0 \) have the form:

\[
\text{select } r.\text{occur}, w_1 || a_1.\text{str} || ... || w_k || a_k.\text{str} || w_{k+1} \\
\text{from } r, r_1 \text{ as } V_{1,\text{str}}, ..., r_n \text{ as } V_{n,\text{str}}, \text{reachofI} \\
\text{where } r.v_1 = r_1.\text{occur} \text{ and } ... \text{ and } r.v_n = r_n.\text{occur} \text{ and } r.\text{occur} = \text{reachofI.occur} ;
\]

The subquery that is generated as part of the nonterminal query for list queries has the form:

\[
\text{select } \text{occur}, \text{str} \\
\text{from } r_{L,\text{str}} \\
\text{where } \text{position} = 1;
\]

The list query \( r_{L,\text{str}} \) for a list production where the repeating symbol is a non-terminal has the form:

\[
\begin{align*}
\text{r}_{L,\text{str}} &= \text{select } r_{L,\text{str}}.\text{occur}, r_{L,\text{str}}.\text{position}, \text{str} = \{N_1\}_{\text{Str}}.\text{str} \parallel r_{L,\text{str}}.\text{str} \\
&\text{from } r_{L,\text{str}}, \{N_1\}_{\text{Str}}, r \\
&\text{where } r.\text{position} = r_{L,\text{str}}.\text{position} - 1 \text{ and } r.a = \{N_1\}_{\text{Str}}.\text{occur} \\
&\text{union} \\
&\text{select } r.\text{occur}, \text{max}(r.\text{position}), \{N_1\}_{\text{Str}}.\text{str} \\
&\text{from } r, \{N_1\}_{\text{Str}} \text{ group by } r.\text{occur} \\
&\text{where } r.a = \{N_1\}_{\text{Str}}.\text{occur};
\end{align*}
\]

The list query \( r_{L,\text{str}} \) for a list production where the repeating symbol is a lexicon has the form:

\[
\begin{align*}
\text{r}_{L,\text{str}} &= \text{select } r_{L,\text{str}}.\text{occur}, r_{L,\text{str}}.\text{position}, \text{str} = r.a \parallel r_{L,\text{str}}.\text{str} \\
&\text{from } r_{L,\text{str}}, r \\
&\text{where } r.\text{position} = r_{L,\text{str}}.\text{position} - 1 \\
&\text{union} \\
&\text{select } r.\text{occur}, \text{max}(r.\text{position}), r.a \\
&\text{from } r, \text{reachofI} \text{ group by } r.\text{occur} \\
&\text{where } r.\text{occur} = \text{reachofI.occur};
\end{align*}
\]
Example 14 The following is the derives set of queries for the language example given in Section 3.

\[\text{blockStr} = \text{select prog.occur, 'begin' || r1.str || 'end'}\]
\[\text{from prog, r1 as stmtlistStr}\]
\[\text{where prog.body = r1.occur;}\]

\[\text{stmtStr} = \text{select ifstmt.occur, 'if' || r1.str ||}
\[\text{'then' || r2.str ||}
\[\text{'else' || r3.str}\]
\[\text{from ifstmt, r1 as condStr, r2 as stmtlistStr, r3 as stmtlistStr}\]
\[\text{where ifstmt.bool = r1.occur and ifstmt.trueact = r2.occur}
\[\text{and ifstmt.falseact = r3.occur}\]
\[\text{union}\]
\[\text{select assign.occur, assign.var || ':' || assign.value}\]
\[\text{from assign, reach0fI}\]
\[\text{where assign.occur = reach0fI.occur;}\]

\[\text{stmtlistStr} = \text{select occur, str}\]
\[\text{from stmtlistLstr}\]
\[\text{where position = 1;}\]

\[\text{stmtsLstr} = \text{select stmtsLstr.occur, stmtsLstr.position}\]
\[\text{str = stmts.stmt || stmtsLstr.str}\]
\[\text{from stmtsLstr, stmts}\]
\[\text{where stmts.position = stmtsLstr.position - 1}\]
\[\text{union}\]
\[\text{select stmts.occur, max(stmts.position), stmts.stmt}\]
\[\text{from stmts}\]
\[\text{group by stmts.occur;}\]

\[\text{condStr} = \text{select equal.occur, equal.var || '=' || equal.value}\]
\[\text{from equal, reach0fI}\]
\[\text{where equal.occur = reach0fI.occur;}\]

6 Varying the Level of Decomposition

As previously mentioned, other projects that provide database support for complex objects store only aspects about the objects that are of interest to the current application. This poses a problem for query evolution. As the application evolves and new queries arise, additional information about the objects must be gathered and stored in the database.
The premise for our work is that all the information about complex objects, in particular textual objects, is stored in the database. If this information is stored fully decomposed and only accessed as a whole, there is clearly a lot of unnecessary overhead required to reconstruct the information, (i.e. the computation of derives from Section 5).

We suggest that the level of decomposition for any complex object should reflect the level of access needed to support the applications that are querying this information. The combination of GeneRel, GeneParse, and derives provides this flexibility. Information for which component fields are not being accessed can be composed. If, in the future, it is necessary to access the components of these fields, the information needed to parse the composed fields is contained in the stored TCFG. Furthermore, if the component fields are being accessed frequently, relations for storing these fields can be generated and the data in these fields can be decomposed into the new relations.

The level of decomposition is specified by a set of composite nonterminals. The decomposition of a composite nonterminal, N, can be automated. The procedure that decomposes N must perform the following:

1. apply derives to the meta-grammar to regenerate the TCFG from the Catalog
2. let $G_N$ represent a TCFG with start symbol N that contains all productions reachable from N in the original TCFG:
   - apply GeneRel to $G_N$ to define the new relations needed to support the decomposition of N
   - apply GeneParse to $G_N$ to generate a sub-parser\(^+\) for parsing and storing the composed lexical fields of N
3. for every production that has N on the right-side, generate a new relation with the same structure as the old relation replacing the the lexical domain N with a domain of surrogates,
4. update the parser\(^+\) with insertion statements for storing of future sentences, and
5. define views to support queries that previously accessed composed fields for N.

Clearly, there is an inverse procedure for building composite nonterminals from decomposed information.

**Example 15** Imagine an application that maintains a database of names and addresses, described by the TCFG that follows.

\[
\begin{align*}
<pinfo:info> & \rightarrow \ <pname:name> \ <paddress:address> \\
<usa:address> & \rightarrow \ <street:street> \ <state:state> \ <zip:zipCode>
\end{align*}
\]
In the early stages of the application, the database was only required to print the addresses, so `address` was added to the list of composite nonterminals, and the information above was stored in the relation:

```
pinfo(occur:info, pname:name, paddress:address)
```

where `info` is a set of surrogates, `name` is a lexical domain for storing names, and `address` is a domain that is defined by the above production for `address`.

It then became necessary to form queries that require access to the `state` and `zipcode` fields of the addresses. The new relations generated to handle this scenario were:

```
pinfo'(occur:info, pname:name, paddress:address');
usa(usa:address', street:street, state:state, zip:zipcode');
```

where `address'` is now a set of surrogates, and `street`, `state`, and `zipcode` are lexical domains.

The following steps must be taken to decompose the data in the database and to provide support for queries that accessed the old relations.

```
for each tuple P in pinfo
{
    parse P.address and store in usa;
    let S be the surrogate of the stored tuple;
    insert into pinfo'
    values (occur = P.occur, name = P.name, address = S);
}
delete relation pinfo;
create view pinfo =
    select occur, name, address=d.sentence
    from pinfo', a as (select occur from usa), d as derives(a)
    where pinfo'.occur = a.occur;
```

Future applications can now form queries that access the states and zipcodes of the addresses. Meanwhile, the existing applications that access `pinfo` will not be invalid, but supported by the view `pinfo`.

7 Future Work

Several other projects have developed grammatical models for describing complex objects [GT83,GT87,GPV89,CCRZ+90], built tool-generators [MKN89], employed grammar specifications to support database operations [Loh87,RB82], incorporated relational concepts
to enhance language specifications [HT86, Hor90, CCRML88], and used relational databases to store complex objects [BK85, CNR90, Lin84, LKM85, RUTP85, RY85, Row89]. To our knowledge, no one else has attempted to map grammatical descriptions of objects into the relational model and to use the structural information contained in these descriptions to facilitate the manipulation of the stored data.

There are still several issues that must be addressed to understand the full potential and practicality of our approach.

- **GeneRel** is one possible mapping from TCFGs to relational schema. Is there a better representation for the generated relations? One which collapses recursive queries that span several relations into transitive closure queries within a single relation? This would allow us to take advantage of the existing formalisms for expressing transitive closure [Agr88, JAN87, Car78, RHDM86, KB88] and the known techniques for the efficient management of transitive closure [ABJ89, VB86].

- **GeneParse** is a tool that facilitates the population of the database for textual objects. Other tools must be developed that support the update of objects in the database and the specification of shared subobjects (a requirement for any system that supports complex objects [BKKG88]). The Exodus data model [CDV88] provides support for distinguishing between shared and non-shared fields that can be adapted to our environment. Can shared subobjects be specified through a data editor generated by an editor-generator?

- The three graph operators demonstrate that the database can utilize the structural information from the grammatical descriptions. There must be additional facilities for expressing queries that relate information within a complex object. We are very interested in investigating the utilization of a modified attribute grammar formalism [Hor90] for expressing queries that relate information within a complex object and combining these results with relational languages to relate information between several complex objects.

- **GeneRel** and **GeneParse** facilitate varying levels of decomposition. How should the level of decomposition be specified? Can it be adjusted during database operation by analyzing query usage patterns [Rou82]?  

- Grammars are the basis for several software tools such as syntax-directed editors [RT85] and data translation [MKN89, RC99]. Can these tools be employed in a database environment? For example, how useful are syntax-directed editors for editing database objects? In the scenario described in section 6, can they be used to
ensure that the data conforms to the relational schema and to enforce the structure of the domains of composed nonterminals?

This paper has described one realization of a grammatical schema definition language in the relational model that has been implemented using meta-description, parser generators, and attribute grammars. The generation of the graph-operator computations demonstrated that the grammar descriptions contain more information about the objects stored in the database than previous flat relational descriptions. Furthermore, efficiency issues concerning fully decomposed objects can be alleviated using GeneParse and GeneRel to vary the level of decomposition.

Acknowledgements

We are grateful to Steve Rosen who implemented GeneRel and GeneParse, and to Lois Delcambre, Jennifer Widom and Stefano Ceri for their discussions on various issues related to this work. We would also like to thank the anonymous referees for their encouragement and useful comments on an earlier version of this paper.

References


