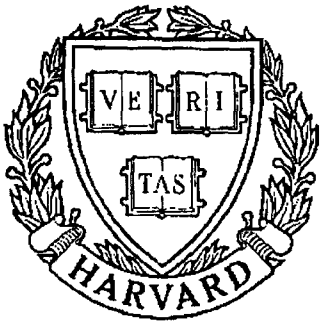
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Automating Relational Database Support for Objects Defined by Context-Free Grammars - the Intension-Extension Framework

by R. Cochrane and L. Mark

TR 89-105

Automating Relational Database Support for Objects Defined by Context–Free Grammars – *the Intension–Extension Framework*

Roberta Cochrane† and Leo Mark†

Abstract

We are designing a framework which provides a common foundation for the integration of databases with other areas of computer science and engineering. This framework is based on the fundamental concepts: context–free grammars and database relations.

Our goal is to provide *automatic* database support for complex objects which can be described by context–free grammars. Such support should include *Data Definition, Data Update, Grammar Catalog Generation, Data Retrieval,* and *Database Restructuring.* This paper addresses the first three areas:

Data Definition: GeneRel automatically generates a set of normalized relational schemes under which objects derived from a given grammar can be stored.

Data Update: GeneParse automatically generates parser specifications with insertion statements for storing sentences acceptable by a given grammar.

Grammar Catalog Generation: GeneRel, when applied to a meta–grammar, generates relations in which grammars derived from the meta–grammar can be stored.

Furthermore, *GeneRel* and *GeneParse* can be implemented through the specification of semantic actions in a compiler–compiler specification of the meta–grammar.

We believe that *GeneRel* and *GeneParse,* together with our related efforts towards providing support for data retrieval and database restructuring in this environment, provide a tool which eliminates the need for manual relational database design, enhances data storage and querying, aids in the process of database restructuring, and provides a common foundation for the integration of databases with other areas of computer science and engineering.

1. Introduction

The development of new research areas, such as *Software Engineering Databases, Knowledge Base Systems, Engineering Information Systems, Extensible Databases,* and *Persistent Data Objects,* indicates a growing interest in integrating database management technology with other established research areas.

- Database support for software engineering has been developed in many projects, including GENESIS [Ramamoorthy, Usuda, Tsai & Prakash] and CACTIS [Hudson & King 86].
- Deductive rule systems, such as PROBE [Dayal & Smith] and POSTGRES [Stonebraker, Hanson, & Hong], indicate efforts to merge databases and expert systems.
- The use of Relational DBMSs in the support of Computer Aided Design (CAD) [Batory & Kim], [Dadam *et al.*] suggest the need to extend existing database management systems to

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

accommodate scientific engineering applications.

- Extensible databases (which are being studied in the context of object–oriented database management systems such as EXODUS [Carey, Dewitt & Vandenberg] and GENESIS [Batory et al 1988]) and persistent data objects [Atkinson, Buneman, & Morrison] are newly developed areas which attempt to integrate databases and programming languages.

Databases supporting the above applications require the ability to define, store and retrieve large amounts of data about objects which have complex structures, such as dependencies between products and processes of software life–cycles, rules, complex objects, text, and programs. Furthermore, the structure of these objects evolves over time, so supporting databases must have the ability to easily adapt to changes.

Each of the aforementioned projects approach database integration from an application specific perspective. The work described in this paper provides a common foundation on which an integration of databases with all of the above areas can evolve. This common foundation is important, not only because it solves a more general problem, but because the above areas are not isolated from one another and are often integrated into the same system. We base our work on one of the fundamentals of computer science, context–free grammars [Hopcroft & Ullman], and the foundation of databases: relations [Codd].

Our goal is to provide *automatic* relational database support for objects whose structure can be described by context–free grammars. Context–free grammars are an appropriate data definition language for these structures because many complex structures can be described by context–free grammars and many experts in these areas are skillful users of context–free grammars. The relational database system is chosen as the target database system because of its theoretical foundations and general acceptance in the research as well as commercial communities.

To achieve automatic database support for objects, we want to develop tools which eliminate the need for manual relational database design and enhance data storage and querying. We have developed two algorithms, *GeneRel* and *GeneParse*, which support data definition and data update, and we used the algorithms to generate grammar catalog support. *GeneRel* (section 4) automatically generates a set of normalized relation schemes R from a given grammar G. Objects derived from grammar G can be stored in the set of relations R. *GeneParse* (section 5) generates parser specifications with insertion statements for storing sentences of a given grammar under the relations generated by *GeneRel*. Such a parser generator facilitates data insertion in environments like programming languages and software engineering. A meta–grammar can be written which defines the language of context–free grammars. Support for a grammar catalog (section 6) can be generated by applying *GeneRel* and *GeneParse* to this meta–grammar. *GeneRel* and *GeneParse* can be implemented in the same system *GeneSys* (section 7) as semantic actions in a compiler–compiler specification of the meta–grammar.

Our work has implications on the previously mentioned research areas because the objects of interest in these areas can be defined in terms of context–free grammars:

Software engineering: Software engineering processes, programs, and even grammars themselves (as demonstrated by our use of a meta–grammar) can be defined in terms of grammars.

Expert systems: Horn clauses, deductive rules, and production rules can all be defined by grammars.

Engineering systems: Engineering process plans and complex objects can be defined in terms of grammars.

Extensible databases: New data types which require database to support can be defined by grammars.

Persistent objects: The definition of types in a program can be given in terms of a grammar.

Database support for all of these objects can be generated by applying *GeneSys* on the grammar definitions of these objects.

Our approach contributes to many established areas of computer science and engineering. However, it concentrates on providing database support for these areas. It *does not* support completely the functionalities of each of the areas; rather, it concentrates on the data structure requirement. It *does not* provide solutions to engineering problems; rather, it provides useful database support. It *does not* intend to replace existing support, such as editing environments, with a database system; rather, it is intended to compliment these tools.

We are not proposing a *new* database system; rather, we are extending current DBMSs to provide support for applications requiring complex object support. We are interested in providing a tool which allows users to develop their *own environment.* We do not want to provide database support specific or tailored to any one of these areas. We want to find general, useful tools that contribute to the research in all of these areas.

The remaining of the paper begins with a review of other research efforts which use grammars and relations as a basis for database integration. Section 3 presents some fundamental definitions of context–free grammars and relations. Sections 4,5,6 and 7 describe the proposed concepts, and Section 8 gives a summary of our progress to date and our plans for future research.

2. Related Efforts

As described in the introduction, there have been a several efforts which provide application specific database support in many areas which would benefit from the integration with databases. However, we are familiar with only a few closely related efforts that consider a combination of the fundamentals – grammars and relations:

The Grammatical Model [Gyssens, Paredaens, & Van Gucht 89] supports data structures based on grammars and provides an algebra and calculus for manipulating grammars. The grammatical model is an ideal basis for a database supporting the application areas described in the introduction. We also recognize the importance of grammars as a tool for describing information. The major difference is that we provide algorithms for generating relational database support from grammar definitions.

A model for language–based editing environments that includes a relational database is described in [Horwitz & Teitelbaum]. Programs are represented as attributed abstract syntax trees with an associated relational database which aggregates information that would otherwise be scattered throughout the program tree. We also consider the combination of relations and grammars. However, we look at how grammars can be used as a specification of a database; they consider how a

relational database can be used to support specifications and operations in attribute grammars.

Inspiration for this work comes from our previous efforts in *Self-Describing Databases, Engineering Information Systems, Operative Expert Systems,* and *Software Engineering Databases.* In [Mark] we introduced the notion of *Self-Describing Databases* with multiple levels of intensions and extensions of data explicitly represented. This notion has inspired the framework used in the present work (see section 6). In [Mark & Roussopoulos] we used *Update Dependencies,* a powerful operational database specification language, to specify the intension–extension dimension of a relational database system. In [Cochrane 89] we described Update Dependencies through the use of attribute grammars. The semantic actions were specified in terms of Update Dependencies against a set of relations storing Update Dependencies. These relations were manually generated from a context–free grammar specification of Update Dependencies. This process was a key inspiration for the *GeneRel* algorithm presented in this paper. Finally, in [Mark & Rombach] we investigated the automatic generation of customized *Software Engineering Databases* from grammatical specifications of software processes.

3. Fundamentals

This section gives definitions for the basic terminology concerning grammars and relations which is required to understand the remainder of the paper. It also formally defines our grammar formalism, *tagged context–free grammars,* which is a variant of context–free grammars on which *GeneRel* operates.

3.1. Grammars

Context–free grammars have played an important role in every aspect of computer science.

DEFINITION

A *context–free grammar* is a 4 tuple $\mathbf{G=(V,T,P,S)}$ where:

- **V** is a finite set of *nonterminals*;
- **T** is a finite set of *terminals*;
- **V** and **T** are disjoint;
- **S** is the special symbol called the start symbol;
- **P** is a finite set of *productions* of the form $\mathbf{A} \rightarrow \alpha$ where $\mathbf{A} \in \mathbf{V}$ and α is a string generated from the *regular expression* $(\mathbf{V} \cup \mathbf{T})^*$.

□

We have experimented with a number of formalisms for expressing context–free grammars to find one for which an algorithm can produce a nice set of normalized relations. However, this augmentation does not increase the power of the languages expressible by context–free grammars. It simply structures the definition of the context–free grammars into a more usable format. It incorporates the concept of *tokens,* the adaptation of *closure* and *positive closure* notations, and the inclusion of *tag–names.*

The concept of *tokens* is included because, as in other applications of context–free grammars, it is often convenient to group together terminal characters into single entities. These single entities are referred to as *tokens*. However, we would like to further distinguish between those tokens that have associated data and those that do not. We define *delimiters* as those tokens that do not have associated data values and *lexicons* as those tokens that have associated data values. This distinction is made because the data values for lexicons must be explicitly represented for any string in the defined language. We will assume the existence of a lexical analyzer that returns tokens and values.

The *closure* (*) and *positive closure* ($^+$) notations used in the representations of regular expressions are convenient ways of indicating repeating structures. Although we can develop an algorithm to generate relations without these notations, a grammar expressed using these notations allows the generation of relations that utilizes the set retrieval aspects of the relational query languages. This will be discussed in further detail in section 4.

The idea to include *tag–names* into our tagged context–free grammar was inspired by [Madsen & Nϕrgaard]. This notation allows the user to specify meaningful names for the generated relations and attributes. Note that we currently force all nonterminal and lexicon occurrences to be *tagged*; however, this restriction can be lifted by introducing default naming conventions.

DEFINITION

A *tagged context–free grammar* is a 6 tuple **E=(S, V, L, D, T, P)** where:

- **S** is the special symbol called the start symbol;
- **V** is a finite set of *nonterminals*;
- **L** is a finite set of *lexicons*;
- **D** is a finite set of *delimiters*;
- **V**, **L**, and **D** are disjoint;
- **T** is a set of *tag–names*;
- **P** is a set of *productions* of the form $\mathbf{<t:A>} \rightarrow \alpha$ where $\mathbf{A} \in \mathbf{V}$, $\mathbf{t} \in \mathbf{T}$, **t** uniquely specifies a production, and α has one of the following forms:
  - a string generated by the *regular expression* $(\mathbf{<T:V>} \cup \mathbf{<T:L>} \cup \mathbf{D})^*$ where each element from **T** is unique within the string,
  - $\mathbf{K}^*$ where **K** is a symbol in $(\mathbf{<T:V>} \cup \mathbf{<T:L>} \cup \mathbf{D})$
  - $\mathbf{K}^+$ where **K** is a symbol in $(\mathbf{<T:V>} \cup \mathbf{<T:L>} \cup \mathbf{D})$

□

A production consists of a *left–side* and a *right–side*. The *left–side* is the string (i.e. <t:A> from the definition) that precedes the "→", and the *right–side* is the string that follows the "→". The *nonterminals*, *lexicons*, and *delimiters* make up the symbols of the grammar. A *tagged symbol* is of the form <t:A> where t is a *tag–name*, and A is either a *nonterminal* or a *lexicon* (i.e. A is a symbol requiring storage).

There are two applications of tag names that correspond directly to the restrictions of the use of tag–names listed in the definition: to uniquely name productions within a grammar, and to uniquely name non–delimiter symbols within the right–side of a production. The first use allows the algorithm to distinguish between productions defined for the same nonterminal. A given nonterminal can exist on the left–side of more than one production. We therefore tag the nonterminal with a tag–name that must be unique among all tag–names used to tag left–side nonterminals. The second use allows the algorithm to distinguish between the non–delimiter symbols within the right–side of a single production. Since a given symbol can occur several times on the right–side of a single production, each non–delimiter symbol is given a tag–name which must be unique among all tag–names used within the right–side of the given production.

Productions are classified as either *constructor* rules or *list* rules. Productions that have a right–side of the first form given in the definition are *constructor* rules. Productions that have a right–side of the second and third forms given in the definition are *list* rules.

List rules indicate that strings derivable from the production consist of a concatenation of occurrences of the single specified symbol. The closure notation (*) means that there can be zero or more such occurrences; the positive closure notation ($^+$) indicates that there is at least one such occurrence.

3.2. Relations

The Relational Model, [Codd], is a widely accepted database model based on the set–theoretic *relation*. This section briefly overviews the structures and operations of the model, emphasizing those aspects of the model referenced in the subsequent sections of the paper.

It is often convenient to think of a relation as a table in which there is no duplication of rows (tuples), row order is insignificant, column (attribute) order is insignificant, and all table entries are atomic values. The relational scheme defines the table name, column headings and domains.

DEFINITIONS

A *relational scheme* for a relation has the form $R(A_1{:}D_1, A_2{:}D_2, \ldots A_n{:}D_n)$ where

- **R** is the *relation name*
- $\mathbf{A_i}$'s are *attribute names* denoting attributes of the relation; each attribute name is unique within a given relation scheme
- $\mathbf{D_i}$'s are *domain names* denoting domains over which the attributes are defined; domains need not be distinct.

A *relation* R consists of a set of tuples where a *tuple* is a mapping from the set of attributes to a set of values in the associated domains.

A *key* of a relation is a combination of attributes for which the corresponding tuple values are unique.

□

The relational model is extended with domains of *surrogates* [Hall, Owlett & Todd] for representing non–lexical object types. Surrogates are system generated internal identifiers that are ideal for representing unnamed objects such as uses of production rules in a grammar.

4. GeneRel

The first step in providing relational support for objects is facilitating the definition of relational schema under which such objects can be stored. The *GeneRel* algorithm automates the process of constructing the relational schemes for a given tagged context–free grammar.

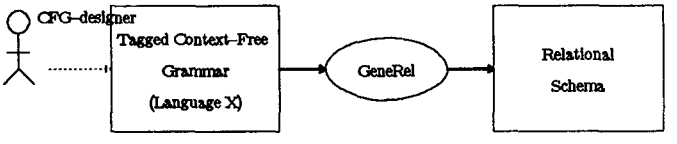


Figure 1: *Automatic Generation of Relation Schemes*

GeneRel defines a mapping from tagged context–free grammar specifications to relational schemes which support the storage of the objects defined by the grammars (Figure 1). Relation names, attributes names and domains are determined from the tag names and symbols in the productions. Tag names are used to derive relation names and attribute names. All nonterminal and lexicon symbols, have corresponding domains. For each nonterminal, *N*, in the grammar, a domain, *N*, of surrogates uniquely represents derivations of the nonterminal. For each lexicon, *L*, in the grammar, a domain, *L*, of lexical objects represents the syntactic category defined by the lexicon.

The *GeneRel* algorithm is summarized in Figure 2. One relation scheme is generated for each production in the grammar. The form of the generated relation schemes depends on the structure of the productions. Productions are classified according to their structure as either *constructor* rules, *list–of–structure* rules, or *list–of–delimiter* rules.

A *constructor* rule has the form:

$$<t_0{:}A> \rightarrow w_1 <t_1{:}A_1> w_2 <t_2{:}A_2> \ldots w_n <t_n{:}A_n> w_{n+1}$$

where w_i is a possibly empty string of delimiters, A_i is a nonterminal or lexicon, A is a nonterminal, and t_i is a tag–name. The relation generated for a constructor rule is named with the tag–name, t_0, of the left–side nonterminal of the rule. The key for this relation is an attribute named *occur* which is defined over the domain, A, which corresponds to the left–side nonterminal. Additionally, there is an attribute corresponding to each nonterminal or lexicon symbol, A_i, on the right–side. The attribute name is the tag–name of the symbol, t_i, and the domain is the domain, A_i, which corresponds to the symbol.

A *list–of–structure* rule has the form:

$$<t_0{:}A> \rightarrow <t_1{:}A_1>^* \text{ or } <t_0{:}A> \rightarrow <t_1{:}A_1>^+$$

where A_i is a nonterminal or lexicon, A is a nonterminal, and t_i is a tag–name. The relation generated for a list–of–structure rule is named with the tag–name, t_0, of the left–side nonterminal of the rule. This relation has three attributes: one named *occur* that is defined over the domain, A, which corresponds to the left–side nonterminal, one corresponding to the nonterminal or lexicon

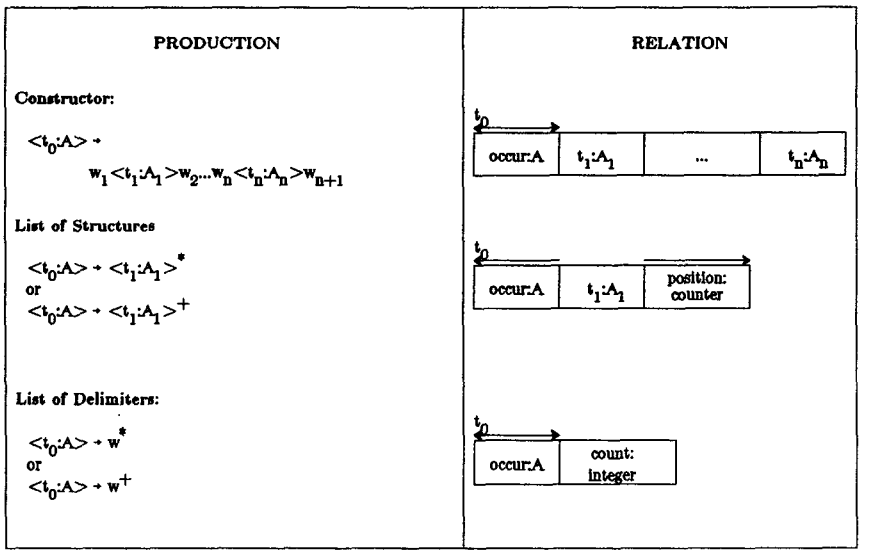


Figure 2: *GeneRel* Algorithm

symbol A_1 named by the tag–name t_1 defined over the domain A_1, and an additional attribute named *position* defined over the domain of positive integers. The *position* attribute is introduced to indicate the order of symbols in the stored sentential form. The attributes *occur* and *position* constitute a key for the relation.

A *list–of–delimiter* rule has the form:

$$<t_0:A> \rightarrow w^* \text{ or } <t_0:A> \rightarrow w^+$$

where w is a delimiter, A is a nonterminal, and t_0 is a tag–name. The relation generated for a list–of–delimiter rule is named with the tag–name, t_0, of the left–side nonterminal of the rule. This relation has two attributes: one named *occur* that is defined over the domain, A, which corresponds to the left–side nonterminal, and an additional attribute named *count* defined over the domain of positive integers. The *count* attribute is introduced to indicate the number of occurrences of the delimiter in the stored sentential form.

EXAMPLE

This example shows the production set of a tagged grammar for specifying drawing descriptions which contain complex objects (Figure 3). A drawing is a rectangle which consists of a location and a body which is described by a number of smaller rectangular areas. The location of a drawing is defined by the bottom–left and top–right points of the drawing. There are two types of areas: components and free paths. The lexicon *integer* is assumed to be defined in the lexical analyzer. Figure 4 shows the relation schemes that are generated by applying *GeneRel* to the Drawing Descriptions tagged grammar.

A new notation is introduced for expressing relation schemes. The domains are represented by circles: a solid circle represents a domain of surrogates (non–lexical objects), a dashed circle represents a domain of lexical objects. The rectangles denote the relation schema with the attribute

<DRAWING:DRAWING>	→	<location:RECTANGLE> <body:AREAS>
<AREAS:AREAS>	→	<area:AREA>*
<PATH:AREA>	→	<free:RECTANGLE>
<COMPONENT:AREA>	→	<occupied:DRAWING>
<RECTANGLE:RECTANGLE>	→	<bot–left:POINT> <top–right:POINT>
<POINT:POINT>	→	<x:integer> <y:integer>

Figure 3: Tagged Grammar for Drawing Descriptions

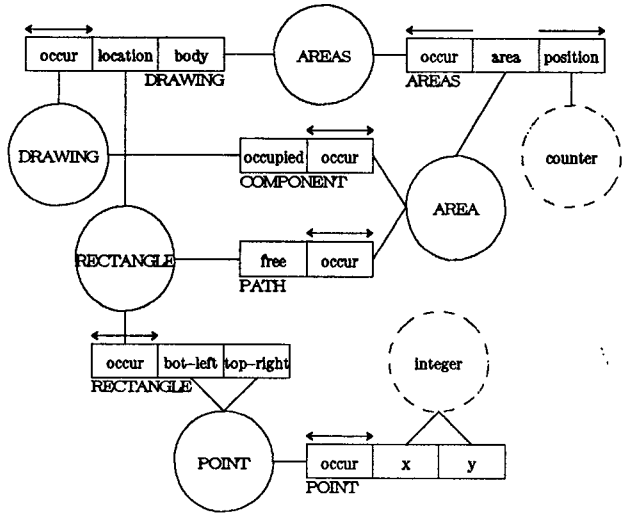


Figure 4: Schema for Storing Drawing Descriptions

names occurring within the rectangle and the relation name occurring somewhere outside but near the rectangle. The arrowed lines indicate attributes that constitute a key for the relation. This notation allows shared domains, which indicate joinable attributes, to be highlighted.

EXAMPLE

This example shows the production set of a tagged grammar that defines a simple structured programming language (Figure 5). A block is defined as a possibly empty list of statements. The lexicons *id* and *int* are assumed to be defined in the lexical analyzer. Figure 6 shows the relation schemes that are generated by applying *GeneRel* to the grammar in Figure 5.

<BLOCK:BLOCK>	→	begin <body:STMTLIST> end
<STMTLIST:STMTLIST>	→	<stmt:STMT>*
<IFSTMT:STMT>	→	if <cond:COND>
		then <trueact:STMTLIST>
		else <falseact:STMTLIST> endif
<ASSIGNSTMT:STMT>	→	<var:id> := <value:int>
<EQUAL:COND>	→	<var:id> == <value:int>

Figure 5: Tagged Grammar for Simple Language

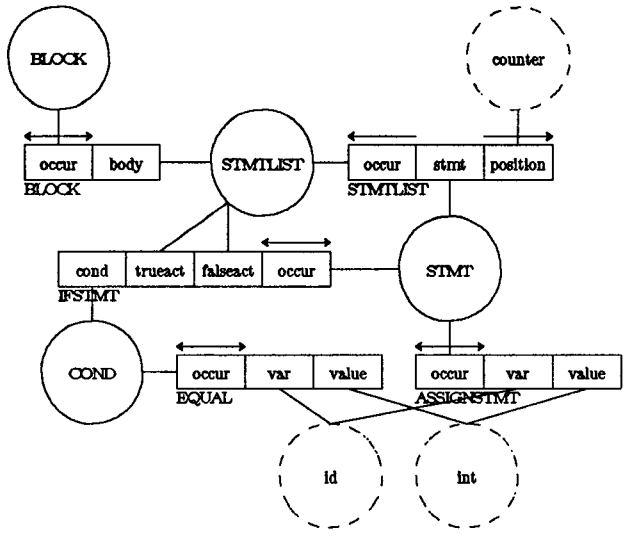


Figure 6: Schema for Simple Language Example

An instance of a block stored under this schema would have a tuple, K, in the relation BLOCK, where $K.occur$ is a surrogate which uniquely identifies the block and $K.body$ is a surrogate which identifies the statement list comprising the block. The tuples in the relation STMTLIST which have the value $K.body$ in the *occur* field are all of the statements which comprise the statement list of the block. Figure 7 shows a program which is a sentence of the blocks grammar, and a possible representation for this program under the schema generated by GeneRel.

```
begin
  X := 3
  if X == 4 then
    X := 5
  else
    if X == 3 then
      X := 4
    else
      X := 3
    endif
  endif
end
```

BLOCK

occur	body
B_1	STS_1

STMTLIST

occur	stmt	position
STS_1	ST_1	1
STS_1	ST_2	2
STS_2	ST_3	1
STS_3	ST_4	1
STS_4	ST_5	1
STS_5	ST_6	1

ASSIGNSTMT

occur	var	value
ST_1	"X"	"3"
ST_3	"X"	"5"
ST_5	"X"	"4"
ST_6	"X"	"3"

IFSTMT

occur	cond	trueact	falseact
ST_2	C_1	STS_2	STS_3
ST_4	C_2	STS_4	STS_5

EQUAL

occur	var	value
C_1	"X"	"4"
C_2	"X"	"3"

Figure 7: Program

The representation stored under the GeneRel schemes can be conceptually pictured as directed acyclic graphs (DAGS) with typed nodes. The value of a node is the attribute value it represents. If the attribute value is a surrogate, then the type of the node is the domain (which corresponds to a nonterminal) over which the surrogate is defined. If the attribute value is lexical, then the type of the node is lexical. Surrogate nodes have at least one child, and lexical nodes do not have any children. Node N is a child of surrogate node X if the value of N is an attribute value in the same tuple for which node X is the *occur* value. If there is no object sharing, the stored DAGS are trees which are similar to parse trees for the stored sentences. In either case, a parse tree for a given derivation can be constructed from the stored database. Figure 8 is the conceptual DAG which is stored for the example in Figure 7.

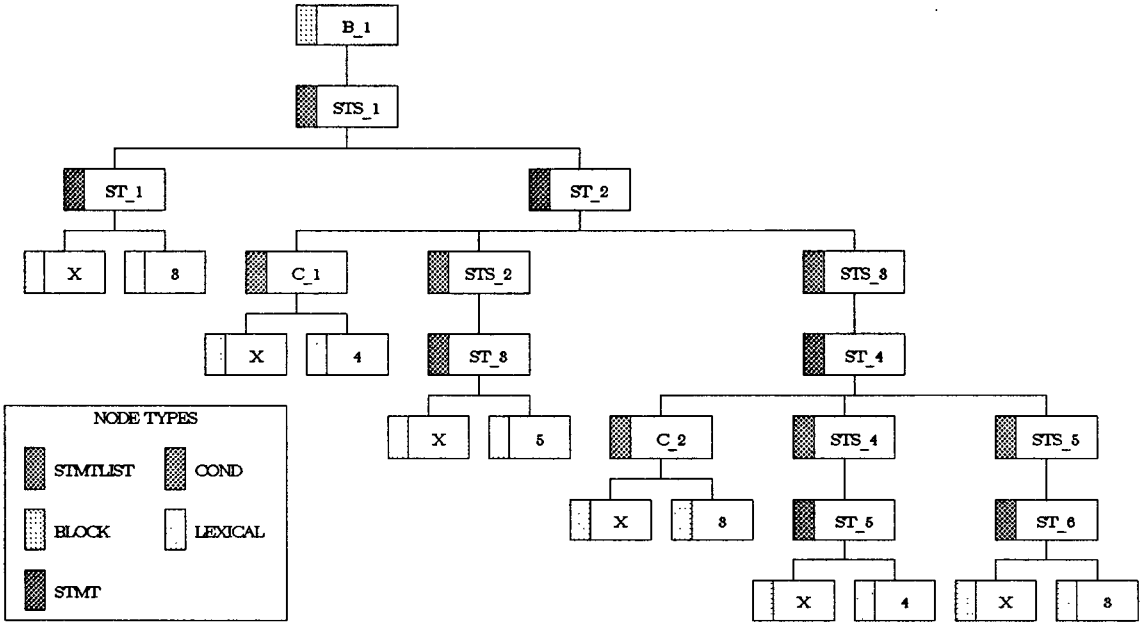


Figure 8: Conceptual DAG

5. GeneParse

Database update, often overlooked as being trivial, is an important issue in providing relational database support. However, in systems which support the storage of objects with complex structures, data insertion, deletion, and modification are nontrivial problems.

In a system which provides general support for many different kinds of objects, different tools will be needed to support database update. In a CAD/CAM environment, a tool may interface with the graphics display to determine the hierarchy of objects and object sharing and to automatically store these objects and relationships under the relations generated by *GeneRel*. In an interactive system, a tool may guide the user through data update, allowing the specification of data describing new objects, as well as data identifying previously stored data. In the simplest case, a tool may parse sentences accepted by a tagged grammar and store them under the schema generated by *GeneRel* for the tagged grammar. This section describes an algorithm, *GeneParse*, which provides

the basis for a tool which supports this simplest case.

In many applications, objects which are described by context–free grammars are also written as sentences of the grammar. These sentences are subsequently parsed for further processing, such as object code generation (when the sentences are programs) or compiler generation (when the sentences are grammars describing programming languages). In such an environment, a useful tool for data insertion is one which parses the sentences and stores them under the schema generated by *GeneRel* (Figure 9). This tool would be useful in a programming environment in which it is desirable to store programs for further analysis and querying.

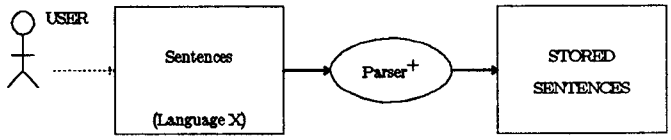


Figure 9: Parsing and Storage of Sentences

GeneParse supports the creation of such a tool. The algorithm takes as input a tagged grammar and generates a parser specification which contains insert statements for storing the parsed sentences in the database (Figure 10)

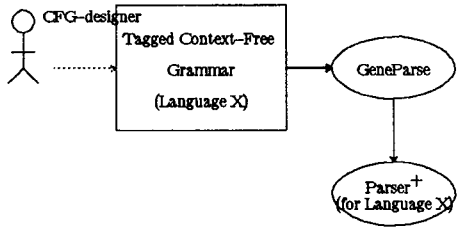


Figure 10: Automatic Generation of Parsers

Sentences accepted by the tagged grammar can then be parsed and stored under the relations generated by *GeneRel* for the tagged grammar. Figure 11 depicts *GeneRel* and *GeneParse* in the intension–extension framework. The intension–extension framework relates the intensional tagged context–free Grammars to their extensional sentences and the intensional generated relational schema to their extensional stored data.

Instead of actually creating a parser, *GeneParse* creates a parser specification for a compiler–compiler. Each production in the tagged grammar is translated into an equivalent context–free grammar production acceptable to the compiler–compiler. The translation for constructor rules is straightforward — tags are removed and the proper delimiters are used. The list rules involve producing two extra rules to express the list as left recursive compiler–compiler rules. The database insert statements are added as semantic actions which are invoked when the corresponding productions are recognized. These statements are based on the knowledge of the relations generated by *GeneRel*.

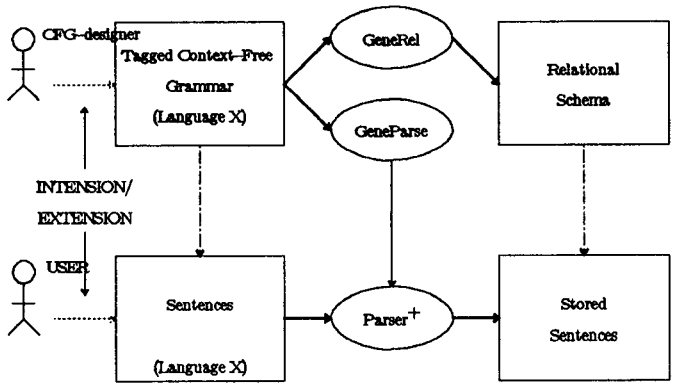


Figure 11: *GeneRel & GeneParse* in the Intension–Extension Framework

6. The Grammar Catalog

It is often desirable to obtain structural information about data stored in the database. In the case of data stored under the schema generated by *GeneRel*, this structural information is contained in the tagged grammar that generated the schema. For example, one may want to know which components comprise an *if-statement*, or if *nonterminal N* is used in the definition of *nonterminal S*. The structural information is also necessary for the execution of the extended algebra operators, described in [Cochrane & Mark 90A]. It is, therefore, necessary to store the tagged grammars in the database. The relations which support the storage of the tagged grammars will be referred to as the *grammar catalog*.

Support for the catalog can be generated by applying *GeneRel* and *GeneParse* to a tagged context–free grammar (called the *meta–grammar*) which describes the class of tagged context–free grammars. Figure 12 is a meta–grammar defining the class of tagged context–free grammars.

GeneRel applied to the given meta–grammar produces a set of relation schemes under which any tagged grammar – including the meta–grammar itself – can be stored (Figure 13).

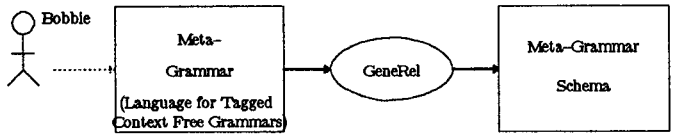


Figure 13: *GeneRel* Produces Meta–grammar Schema

Figure 14 depicts the relation schemes generated by applying *GeneRel* to the highlighted meta–grammar productions from Figure 13.

```
{ GRAMMAR,
  { GRAMMAR START NONTERMS LEXICONS DELIMITERS TAG_NMS PRODS PRODUCTION
    NONTERM_TAG RHS SYMBOL LEXICON_TAG }
  { nonterminal delimiter lexicon tag_name }
  { { } → * + ' < > : ; }
  { GRAMMAR, start, nonterms, lexicons, delimiters, tag_nms, prods, START, value, NONTERMS, LEXICONS, DEL-
    IMITERS, TAG_NMS, PRODS, PRODUCTIONS, left-side, right-side, CONSTRUCTOR, LIST, POSITIVE_LIST,
    DELIMITER_SYM, NONTERM_SYM, LEXICON_SYM, NONTERM_TAG, name, domain, LEXICON_TAG }
  {

    <GRAMMAR:GRAMMAR>          →  '{'<start:START>  '{'<nonterms:NONTERMS>'}'
                                  '{'<lexicons:LEXICONS>'}'
                                  '{'<delimiters:DELIMITERS>'}'
                                  '{'<tag_nms:TAG_NMS>'}'  '{'<prods:PRODS>'}'
                                  '}'

    <START:START>              →  <value:nonterminal>


    <NONTERMS:NONTERMS>        →  <value:nonterminal>+
    <LEXICONS:LEXICONS>        →  <value:lexicon>*
    <DELIMITERS:DELIMITERS>    →  <value:delimiter>*
    <TAG_NMS:TAG_NMS>          →  <value:tag_name>*


    <PRODS:PRODS>              →  <value:PRODUCTION>*
    <PRODUCTIONS:PRODUCTION>→  <left-side:NONTERM_TAG> '→' <right-side:RHS> ;



    <CONSTRUCTOR:RHS>          →  <value:SYMBOL>*
    <LIST:RHS>                 →  <value:SYMBOL>'*'
    <POSITIVE_LIST:RHS>        →  <value:SYMBOL>'+'
    <DELIMITER_SYM:SYMBOL>     →  '''<value:delimiter>'''
    <NONTERM_SYM:SYMBOL>       →  <value:NONTERM_TAG>
    <LEXICON_SYM:SYMBOL>       →  <value:LEXICON_TAG>
    <NONTERM_TAG:NONTERM_TAG>  →  '<'<name:tag_name>':'<domain:nonterminal>'>'
    <LEXICON_TAG:LEXICON_TAG>  →  '<'<name:tag_name>':'<domain:lexicon>'>'
  }
}
```

Figure 12: Meta–Grammar

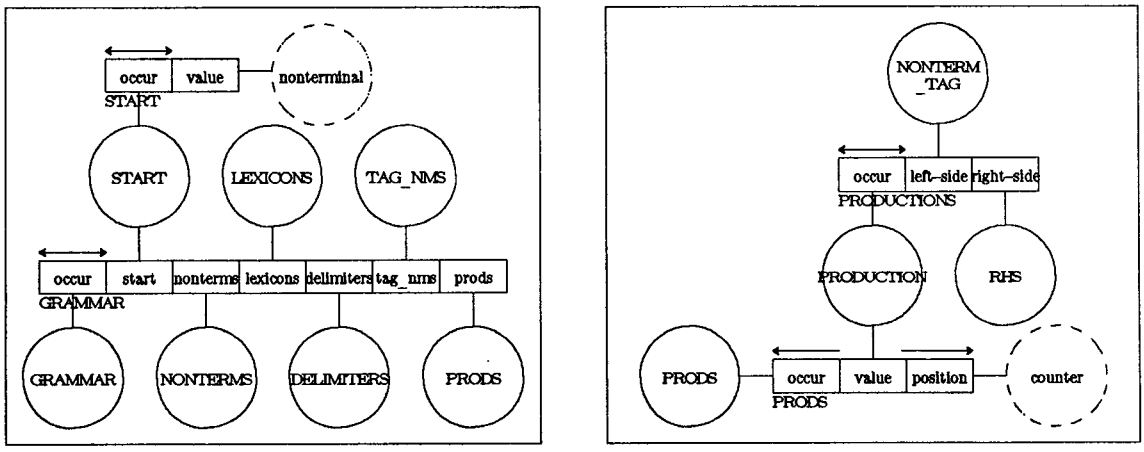


Figure 14: Meta–Grammar Schema

GeneParse applied to the meta–grammar produces the specification of a parser which reads tagged grammars, and stores them under the meta–grammar relations produced by *GeneRel* (Figure 15).

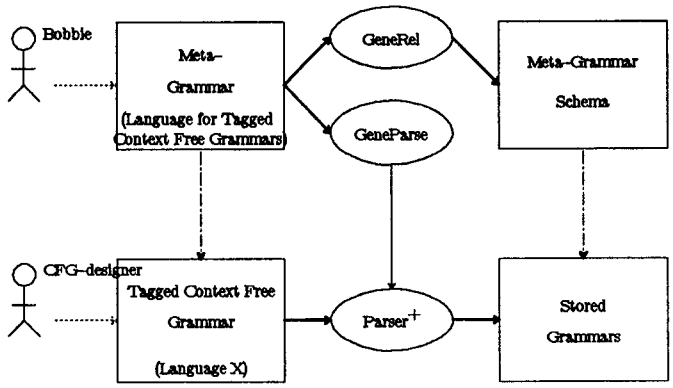


Figure 15: *GeneParse* Produces Tagged grammar Parser

GeneRel and *GeneParse* applied to the meta–grammar add an extra level to the intension–extension framework described in the previous section. The resulting framework is illustrated in Figure 16. It is quite similar to the intension–extension framework for DBMSs presented in [Mark] and in [Mark & Roussopoulos].

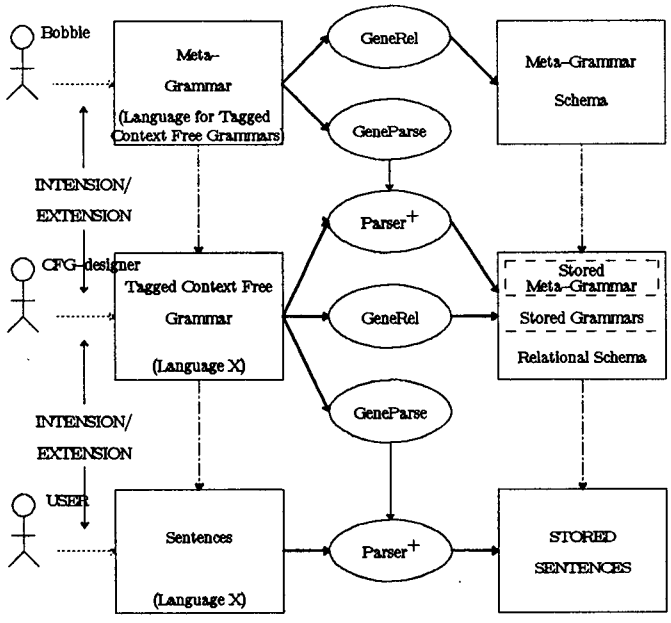


Figure 16: Intension–Extension Framework

7. GeneSys

Both the *GeneRel* algorithm and the *GeneParse* algorithm are driven by the structure of the tagged grammar rules. Therefore, they can be implemented as semantic actions in a compiler–compiler input specification of the meta–grammar.

This section describes a system, *GeneSys*, which supports the middle level of the intension–extension framework. This is the level which supports the data definition and automatic storage for objects whose structures are defined by tagged grammars. *GeneSys* must perform three functions:

- *store*
  store the input tagged grammar specification under the meta–grammar schema,
- *GeneRel*
  apply *GeneRel* to the input tagged grammar, generating schema in the database, and
- *GeneParse*
  apply *GeneParse* to the input tagged grammar, generating a compiler–compiler input specification for the tagged grammar which contains database insert statements.

Since each of these three functions are implemented by semantic actions of the same tagged grammar, they can be merged as semantic actions in the same compiler–compiler specification.

For flexibility, *GeneSys* can perform any combination of its three functions each time it runs. The top level of the intension–extension framework is implemented by *GeneSys* run with the *GeneRel* and *GeneParse* options applied to the meta–grammar. This invocation of *GeneSys* is represented by the notation $GeneSys_{GeneRel,GeneParse}$(meta–grammar). The compiler–compiler specification for the tagged grammar parser$^+$ generated by this invocation of *GeneSys* is exactly the one needed to support the *store* function of *GeneSys*.

Implementation of the system then consists of the following steps:

(1) Integrate the *GeneRel* and *GeneParse* algorithms into the compiler–compiler specification of the meta–grammar. This compiler–compiler supports the top level of the intension–extension framework.

(2) $GeneSys_{(GeneRel,\ GeneParse)}$(meta–grammar) which generates the meta–grammar schema and a parser$^+$ which is a compiler–compiler specification of the meta–grammar containing database insert statements for tagged grammars.

(3) Integrate, by hand, the insertion statements from the parser$^+$ generated in the previous step into the compiler–compiler specification of the first step. This is done to support all three functions of *GeneSys* in one parse of a grammar. The final specification supports the middle level of the intension–extension framework.

(4) $GeneSys_{store}$(meta–grammar) will now store the meta–grammar under the meta–grammar schema.

The default mode, $GeneSys_{store,GeneRel,GeneParse}$ applied to any tagged grammar implements the middle level of the intension–extension framework. The compiler–compiler specification generated is the specification of the parser required to implement the third level of the framework for the given

tagged grammars.

Figure 17 depicts the construction of the different phases of *GeneSys* including the invocations of the compiler–compiler (CC).

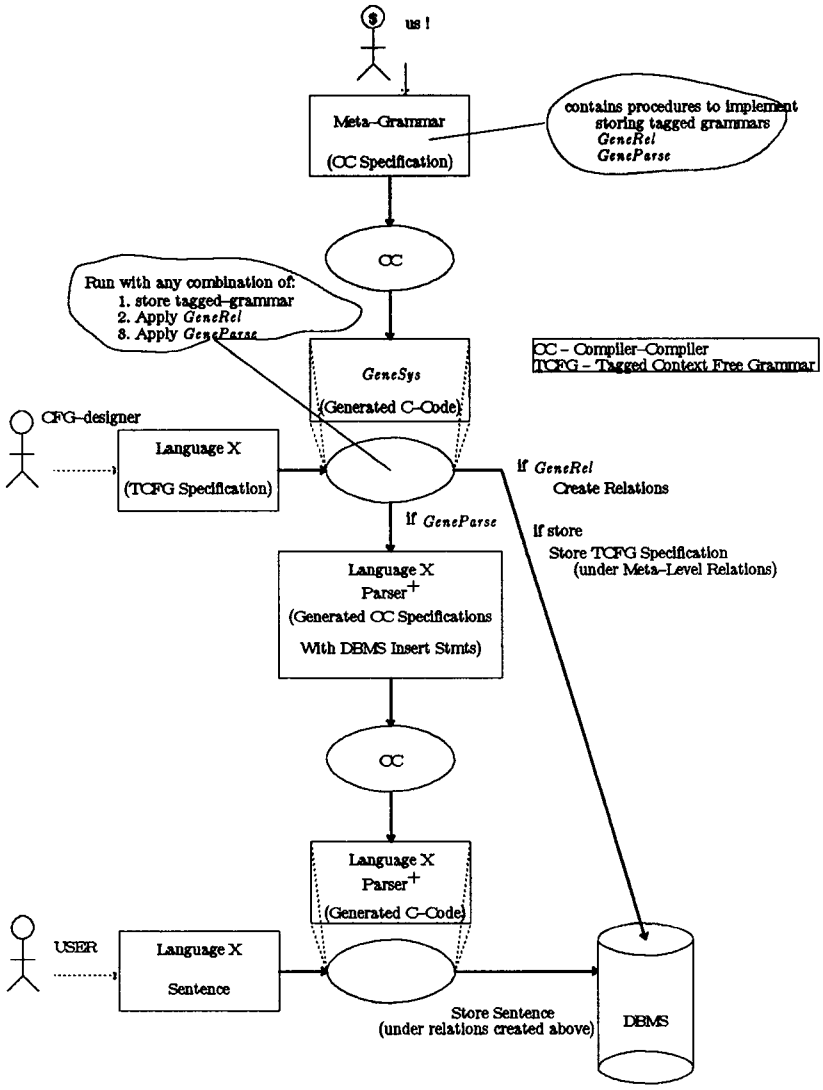


Figure 17: *GeneSys*

8. Summary and Future Research

We have presented two algorithms, *GeneRel* and *GeneParse*, which provide a common foundation for the integration of databases with other areas of computer science and engineering. These algorithms provide automatic data definition and one form of data insertion, respectively, from grammar specifications. We have shown how these algorithms can be implemented in the same system, *GeneSys*, through the specification of semantic actions in a compiler–compiler specification of the meta–grammar. We are currently implementing *GeneSys* as described in section 7.

Our goal is to provide automatic database support for objects which can be described by context–free grammars. The database support needed and our efforts to provide this support are summarized below.

Data Definition: GeneRel automatically generates a set of normalized relations in which objects derived from the grammar can be stored. *GeneRel* has been implemented as part of *GeneSys*. Future research in this area could relax the requirements of the tagged grammar by introducing default relation and attribute naming.

Data Update: GeneParse automatically generates parser specifications with insertion statements for storing sentences acceptable by a grammar from the grammar specification. *GeneParse* has also been implemented as part of *GeneSys*. We are currently developing a method for inserting objects which share objects. Complex objects are rarely written down as sentences of the grammar that describes their structure. They are normally input through an interactive mechanism which allows the user to specify reuse of data objects as a subparts of other objects. We are developing tools which allow the user to identify existing objects and combine them into more complex structures. Future research could concentrate on developing a general update language to support the modification of the data stored under *GeneSys*.

Grammar Catalog Generation: We have shown how *GeneRel* and *GeneParse* can be applied to a meta–grammar to generate automatic catalog support, allowing structural information to be stored as relational data.

Data Retrieval: We are currently developing an *extended relational algebra* which contains graph operators especially designed to retrieve information from the relations generated by *GeneRel* [Cochrane & Mark 90A]. These operators allow the user to specify traversals of the stored parse trees. They facilitate the expression of complex, recursive queries which are common for data having complex structure. The operators are implemented as a set of algebra equations whose fixpoint contains the result. This system of equations can be generated from the grammar specifications. These operators will eventually be integrated into *GeneSys*.

Database Restructuring: We are currently designing a restructuring framework which uses *GeneSys* and the catalog information to automate the process of database restructuring when a grammar is transformed [Cochrane & Mark 90B].

We feel that our approach has great potential because it contributes to many current efforts to integrate databases into other areas of computer science and engineering, and it is based on fundamental concepts in computer science and databases. This common foundation is important, not only because it solves a more general problem, but because the above areas are not isolated from one another and are often integrated into the same system.

References

[Atkinson, Buneman, & Morrison]

Atkinson, M. P., P. Buneman, and R. Morrison, eds., *Data Types and Persistence,* Springer–Verlag, 1988.

[Batory et al 1988]

Batory, D. S., J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, GENESIS: An Extensible Database Management System, *IEEE Transactions on Software Engineering* **Vol. 14**, No. 11 (November 1988), pages 1711–1730.

[Batory & Kim]

Batory, D. S. and W. Kim, Modeling Concepts for VLSI CAD Objects, in *Supplement to ACM SIGMOD International Conference on Management of Data,* 1985, pages 18–32.

[Carey, Dewitt & Vandenberg]

Carey, Michael J., David J. Dewitt, and Scott L. Vandenberg, A Data Model and Query Language for EXODUS, in *Proc. ACM SIGMOD International Conference on Management of Data,* Chicago, Illinois, June 1–3, 1988, pages 413–423.

[Cochrane 89]

Cochrane, Roberta, *Operational Relational Model - Implementation Through Specification,* Systems Research Center, College Park, Maryland, SRC–TR–89–46, 1989.

[Cochrane & Mark 90A]

Cochrane, Roberta and Leo Mark, *Automatic Relational Database Support for Objects Defined by Context-Free Grammars - An Extended Relational Algebra,* Department of Computer Science and Systems Research Center, University of Maryland, College Park, Maryland, (in progress), 1990.

[Cochrane & Mark 90B]

Cochrane, Roberta and Leo Mark, *Automatic Relational Database Support for Objects Defined by Context-Free Grammars - Database Restructuring,* Department of Computer Science and Systems Research Center, University of Maryland, College Park, Maryland, (in progress), 1990.

[Codd] Codd, E. F., Extending the Database Relational Model to Capture More Meaning, *ACM Transactions on Database Systems* **Vol. 4**, No. 4 (December 1979), pages 397–434.

[Dadam *et al.*]

Dadam, P., A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, *Proc. ACM-SIGMOD International Conference on the Management of Data,* Washington DC, May 1986, pages 356–364.

[Dayal & Smith]

Dayal, U. and J. M. Smith, PROBE: A Knowledge–Oriented Database Management System, *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Gyssens, Paredaens, & Van Gucht 89]

Gyssens, M., J. Paredaens, and D. Van Gucht, A Grammar–Based Approach towards Unifying Hierarchical Data Models (extended abstract), in *Proc. ACM SIGMOD International Conference on Management of Data*, Portland, OR, June 1989, pages 263–272.

[Hall, Owlett & Todd]

Hall, P., J. Owlett, and S. Todd, Relations and Entities, in *Modelling in Database Management Systems*, edited by G. M. Nijssen, North–Holland, 1976.

[Hopcroft & Ullman]

Hopcroft, John E. and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley Publishing Company, 1979.

[Horwitz & Teitelbaum]

Horwitz, Susan and Tim Teitelbaum, Generating Editing Environments Based on Relations and Attributes, *ACM Transactions on Programming Languages and Systems* **Vol. 8**, No. 4 (October 1986), pages 577–608.

[Hudson & King 86]

Hudson, S. and R. King, CACTIS: A Database System for Specifying Functionally–Defined Data, in *Proc. Workshop on Object–Oriented Databases*, Pacific Grove, CA, 1986.

[Madsen & Nϕrgaard]

Madsen, O. L. and C. Nϕgaard, An Object–Oriented Metaprogramming System, in *Hawaii International Conference on System Sciences*, January 1988.

[Mark] Mark, Leo, *Self–Describing Database Systems - Formalization and Realization*, Department of Computer Science, University of Maryland, College Park, MD., TR–1264, 1985.

[Mark & Rombach]

Mark, Leo and H. D. Rombach, Generating Customized Software Engineering Information Bases from Software Process and Product Specifications, in *Proc. Twenty–Second Annual Hawaii International Conference on System Sciences*, edited by Bruce D. Shriver, IEEE Computer Society Press, Washington, D. C., January 1989, pages 587 - 595.

[Mark & Roussopoulos]

Mark, Leo and Nick Roussopoulos, Metadata Management, *IEEE Computer* **Vol. 19**, No. 6 (December 1986).

[Ramamoorthy, Usuda, Tsai & Prakash]

Ramamoorthy, C. V., Y. Usuda, W. Tsai, and A. Prakash, GENESIS: An Integrated Environment for Supporting Development and Evolution of Software, in *Proc. COMPSAC*, 1985.

[Stonebraker, Hanson, & Hong]

Stonebraker, M., E. Hanson, and C. Hong, The Design of the POSTGRES Rules System, in *Proc. Third International Conference on Data Engineering*, Los Angeles, CA, February 1987.