Supporting Queries on Source Code: A Formal Framework

Santanu Paul Atul Prakash

Software Systems Research Laboratory
Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI-48105
santanu@eecs.umich.edu, aprakash@eecs.umich.edu

Abstract

Querying source code interactively for information is a critical task in reverse engineering of software. However, current source code query systems succeed in handling only small subsets of the wide range of queries possible on code, trading generality and expressive power for ease of implementation and practicality. We attribute this to the absence of clean formalisms for modeling and querying source code. In this paper, we present an algebraic framework (*Source Code Algebra* or SCA) for modeling and querying source code. The framework forms the basis of our query system for C source code. An analogy can be drawn with relational algebra, which forms the basis for relational databases. The benefits of using SCA include the integration of structural and flow information into a single source code data model, the ability to process high-level source code queries (command-line, graphical, relational, or pattern-based) by translating them into SCA expressions which can be evaluated using the SCA evaluator, the use of SCA itself as a powerful low-level source code query language, and opportunities for query optimization. We present the SCA's data model and operators and show that a variety of source code queries can be easily expressed using them. An algebraic model of source code addresses the issues of conceptual integrity, expressive power, and performance of a source code query system within a unified framework.

Keywords: Reverse engineering, source code query, query languages, algebra, generalized order-sorted algebra.

1 Introduction

Programmers have become part historian, part detective,
and part clairvoyant.

Tom Corbi, in *Program Understanding: Challenge for the 1990s* [12].

In the last few years, software reverse engineering, code re-engineering, and program understanding have emerged as the latest challenges in the field of software engineering. Interest in these areas has been triggered by the presence of extremely large, difficult-to-maintain software systems, better known as *legacy systems*, which for reasons of economics cannot be thrown away and rewritten.

One of the early conclusions in reverse engineering research is that a complete automation of the design recovery process is not feasible [12]. Given the current state-of-art in reverse engineering technology, it is felt that reverse engineering of real systems can at best be automated 50 percent, and the rest must be by human participation [44]. This acceptance of the critical role that must be played by a human reverse engineer has led to research in software tools that can *assist* or support the human in this task.

Of the many tools that will be required to support reverse engineering, we are concerned with the design of one: a language-based tool for querying source code to support the task of software understanding and design recovery. Support for extracting relevant information from source code has so far been left either to rudimentary, string searching tools like `grep`, `awk`, etc. (which are capable of handling only trivial queries), or to general-purpose database approaches that have limited querying power for the source code domain [8, 10, 28, 30]. The need for sophisticated querying tools for reverse engineering has been articulated by Biggerstaff in terms of a "conceptual `grep`" [3], and also by Chikofsky [9]. The purpose of a source code querying tool is to help a human reverse engineer indulge in *plausible reasoning* [3] or *domain bridging* [5] — an iterative process of guesswork and verification that leads him or her to a better understanding of what the source code is doing.

Reverse engineers may need to make several types of queries. Queries may be based on *global structural information* in the source code, e.g., relations between program entities such as files, functions, variables, types, etc. Queries can also be based on *statement-level structural information* in the source code, e.g., looking for *patterns* (e.g., loops) that fit a programming plan or a *cliche* [33, 34, 36]. Queries may also be based on flow information derived by static analyses such as *data-flow* and *control-flow* analyses, e.g., to locate program slices [43], to find the variables whose values are affected by a particular statement, etc. Finally, a reverse engineer may need to make queries that use both structural information as well as program flow information.

Unfortunately, one of the fundamental problems designers of source code querying systems face is the lack of good underlying models to represent source code information and to express queries. For example, in our previous work on building source code querying tools SCAN [1] and SCRUPLE [34], and earlier in our work on the Evolution Support Environment System (ESE) [35], we found that no satisfactory choice for the underlying model to represent program information was available. One option for us was to use the relational model, as used in several systems such as OMEGA [28], CIA [8], and CIA++ [18]. The advantage of that would have been the availability of a formal query language (based on relational algebra) — our work in developing a query language and a query processor would have been reduced. Unfortunately, it is difficult, if not impossible, to use the relational model to make queries for locating patterns in source code and to make queries based on data-flow and control-flow. Another option would have been to use some other representation model such as graphs or abstract syntax trees, as used Rigi [30] and Microscope [2] or an object-based representation as used in REFINE [26] and in [27, 20]. However, the problem with those models would have been the lack of a query language with well-defined operators. Either option was somewhat unsatisfactory. Current versions of SCRUPLE and SCAN ended up using an attributed syntax-

tree representation whereas the ESE system used a relational representation. In both cases, we definitely felt the lack of either a proper query language or adequate modeling power.

In order to alleviate the above dilemma faced by designers of reverse engineering tools, this paper proposes a *source code algebra* as the foundation for building source code querying systems. An algebra defines a model for representing source code information and gives a well-defined set of operators that can be used to make queries on the information. The analogy is the use of *relational algebra* [11] as the foundation for relational database systems. Algebras have also been used in the design of general-purpose query languages for the relational data model [11], the nested relational model [21, 22, 23], the extended relational model [38], the object model [29, 31, 40, 41], and also in the design of a domain-specific query language for structured office documents [19]. The benefits of using an algebra as the basis for a query language include the ability to provide formal specifications for query language constructs, the ability to use the algebra itself as a low-level query language, and opportunities for query optimization. The need for a special-purpose algebra for source code stems from the modeling limitations of above-mentioned data models for representing source code information and the absence of appropriate operators for expressing queries of interest to reverse engineers.

The proposed source code algebra (SCA) effectively models source code information and contains the necessary operators for making a variety of queries of interest to reverse engineers on source code. The model views source code as a domain of *typed* objects with attributes that store component information, relations with other objects, computation methods, and any other relevant information. The model supports the notion of a *collection* of objects. Collections can be viewed as either *sets* (e.g., a set of `variable` objects) or as *sequences* (e.g., a sequence of `statement` objects). Operators are then provided to operate on individual objects and their collections. As in relational algebra, queries are expressed by writing expressions using the given operators.

The paper is organized as follows. Section 2 discusses the type of queries on source code that we would like to be able to handle in the source code algebra. Section 3 discusses our approach of using an algebra to support querying on source code. In order to specify the algebra, we first define a data representation model that is rich enough to capture relevant information about the source code and then give a well-defined set of operators for the model that can be used to express a variety of queries on the source code. Section 4 illustrates the expressive power of the operators — it shows how different kinds of queries on source code are expressed using the given operators. Section 5 outlines design and performance issues in using the algebra as the basis of a system to support source code querying. Section 6 compares our algebra to other algebras that have been proposed for querying in other domains. Finally, Section 7 presents our conclusions and future work.

2 Requirements of a Source Code Query System

While a well-researched survey of commonly-used source code queries continues to be unavailable, a comparative study of systems currently used to query code offers valuable clues regarding the functionality that needs to be supported. In this section, we will present sample source code queries and specify the requirements of a source code query system.

2.1 Examples of Source Code Queries

- **Queries based on Global Structural Information**:

  The first category consists of queries that pertain to global structural information, relating to files, modules, functions, global definitions, etc.

  1. *What are the functions defined in the file* `analyzer.c`*?*

2. *Find all global variable definitions of type* `matrix`.
3. *Find the file that has the maximum number of functions.*

Query 1 pertains to the *organization* or high-level design of the program, specifically, it concerns itself with the distribution of functions in files. Query 2 detects the use of a certain type definition. Query 3 is a numerical query based on program structure, and is representative of a large class of source code queries that are based on software metrics.

- **Queries based on Syntactic Structure**:

  These are queries that deal with fine-grain syntactic and structural information, such as code patterns, structures of constructs, etc.

  1. *Show the body of the function* `sort()`.
  2. *Find patterns consisting of sequences of three* `if` *statements, possibly separated by arbitrary statements.*
  3. *Find all the iterative statements in the program.*

  Query 1 pertains to the *abstract syntax* of a function. Query 2 is essentially a syntactic pattern at the level of statements, based on the implicit concept that a statement list has the semantics of a sequence. Implicit in query 3 is the notion of *generalization*, i.e., `while`,`do`, and `for` statements are specialized forms of iterative statements.

- **Queries based on Program Flow Information**:

  These are queries that probe information flow between source code entities. Typically, maintainers are interested in information that can be obtained by static analyses of source code, such as definition and use of identifiers, data-flow and control-flow information, and so on.

  1. *Find all references to the identifier* `counter`.
  2. *Identify the set of all functions that are directly or indirectly invoked by the function* `sort()`.
  3. *Find the subsequent uses of the variable* `v` *defined in statement* `s`.

  Query 1 is a common source code query based on the "refers-to" relationship between an identifier reference and its definition. Query 2 can be thought of as a recursive query that computes the closure of the program call graph, starting from a given function. Query 3 is an example of simple data flow analysis.

2.2 Definition of a Source Code Query System

We define a source code query system informally as an environment with the following characteristics. First, it must provide a data model for source code which captures structural as well as program flow information. Second, it must provide a query language, that permits the specification of queries based on structural as well as flow information in a seamless manner.

Ideally, the source code data model should be *complete* and *minimal*. Completeness ensures that "all" information needed to query source code is available in the model. In the absence of a formal notion of source code *query completeness*, we must settle for *approximate* completeness based on the range of queries a model can handle. Minimality eliminates redundant information from the data model. At the same time, the source code query language should be *expressive* and *usable*. Expressiveness implies that any information that exists in the data model or can be computed from it should be accessible using the query language.

Usability measures the ease with which such information can be derived. For example, a declarative or applicative language is easier to use than a procedural language.

An implementation of a source code query system must include 1) a repository that stores source code information according to the data model 2) tools that populate the repository with structural and/or program flow information, such as parsers, static analyzers, etc. 3) a interface for the user to specify queries, and 4) a query processor that handles queries by examining the repository.

2.3 Designing a Formal Query Language

To be expressive and usable, a source code query language, in our view, should have two characteristics. First, it should have a formal framework, second, it should be non-procedural.

The arguments in favor of building a formal query language are compelling. The constructs of a formal language have well-defined semantics. It has been observed in the context of query languages that formal frameworks such as relational algebra [11], relational calculus [42], NST-Algebra [19], etc. have yielded powerful and expressive high-level query languages, and have been argued to be functionally complete within their respective data models. Well-defined semantics has led to clean implementations for query processors. In algebraic frameworks such as relational algebra (both classical and extended), rules and heuristics of algebraic transformation have been used for query optimization. In NST-Algebra, as in relational algebra, the algebra can serve as an applicative query language.

A non-procedural query language is desirable because it greatly simplifies the task of expressing queries. In applicative languages such as algebras, a query is specified as an algebraic expression that must be evaluated to obtain the result. In declarative languages such as calculi, a query is a logical assertion about the properties of the result. In either case, there is no need for procedural descriptions of queries.

In contrast, the lack of formal frameworks and the absence of non-procedural query languages in many object-oriented data models has led to problems in query processing and optimization [14].

3 Our Approach: An Algebra for Source Code

To facilitate queries on source code, we have developed a source code data model that captures the necessary structural and program flow information and designed a formal framework to query the model for such information.

The key feature of our approach is the modeling of source code as an *algebra*. Informally, algebras are mathematical structures that consist of data types (*sorts*) and operations defined on the data types (*operators*). We are interested in the design of a *source code algebra* (SCA). The objective is to model the data types in the source code domain as sorts of the SCA, and to design source code query primitives as operators of the SCA. A clear analogy can be found in the relational data model, where the *relational algebra* serves as the underlying mathematical model. By modeling source code as an algebra, we hope to address the conflicting issues of conceptual integrity, expressive power, and performance of a source code query system within a single formal framework.

We will begin this section with a brief description of relational algebra. The purpose is to demonstrate how the domain of relations benefits from an algebraic framework, and offer a rationale for the use of algebras to model source code. Next, we will present our source code data model, and show why SCA must belong to a class of algebras (*generalized order-sorted algebras*) more powerful than that of relational algebra (*one-sorted algebras*). Finally, we will outline the operators of SCA.

3.1 Relational Algebra

Operator	Signature	Description
union,difference intersection	RELATION × RELATION ⟶ RELATION	Obvious
select	RELATION ⟶ RELATION	Returns a subset of the tuples based on a boolean condition
project	RELATION ⟶ RELATION	Returns a relation with only the specified fields
cartesian product	RELATION × RELATION ⟶ RELATION	Combines the tuples in two relations exhaustively
join (natural)	RELATION × RELATION ⟶ RELATION	Cartesian product followed by select

Table 1: Relational Algebra Operators

Classical relational algebra is an instance of a *one-sorted algebra*, i.e., it deals with only one data type, namely *relations*. Relations are sets of tuples whose fields have atomic values such as integers, strings, etc. The primitive operators of the algebra are *union*($\cup$), *set difference*($-$), *select* (σ_c), *project* ($\pi_{a_1,a_2,\ldots,}$), and *cartesian product* ($\times$) [11]. *Join* ($\bowtie$) is a derived operator of the algebra (composition of σ and $\times$). Each of these operators take relations as arguments, and produce new relations. For example, the σ_c operator takes a relation R and produces a new relation R' that contains only those tuples of R which satisfy a given boolean condition c. The signatures of the operators are shown in Table 1.

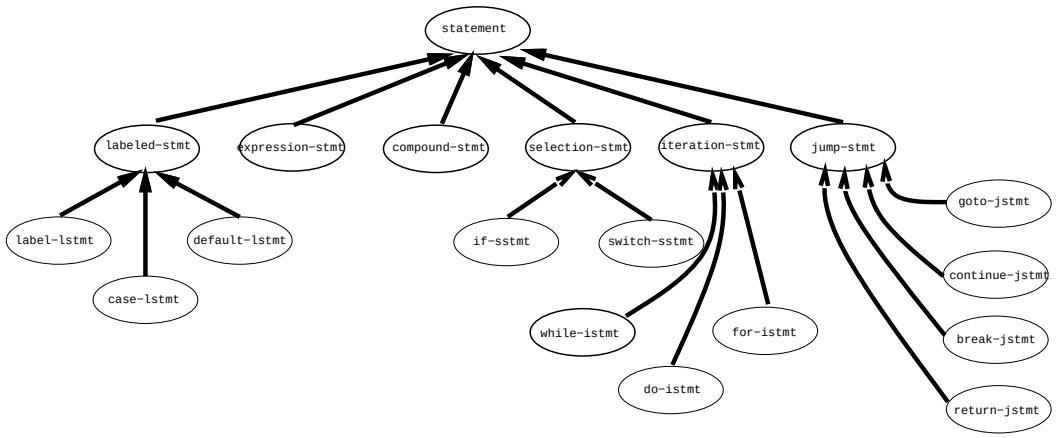
Codd has shown that all information stored using relations can be accessed using the five primitive operators of relational algebra. In that sense, the relational algebra is *query-complete* [11]. Relational algebra has also been shown to be equivalent to *relational calculus* [42]. Relational algebra (or its equivalent relational calculus) forms the basis of a wide variety of relational database query languages such as SQL, QUEL, ISBL, and QBE [42]. However, a major weakness of relational algebra is that it fails to include basic data types such as integers, strings, etc. as elements of the algebra itself. Consequently, many operations permitted in SQL (aggregate, sort, etc.) do not have well-defined semantics in terms of relational algebra [19].

Relational algebra also helps in query optimization by algebraic transformations. Consider the relational algebra expression $\sigma_{c_1}(\sigma_{c_2}(R))$. It so happens that σ commutes with itself, and we have the following identity:
$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
Now, if R were to contain a large number of tuples, and condition c_2 was significantly harder to compute than c_1, we could optimize an algebra expression which contained the subexpression $\sigma_{c_1}(\sigma_{c_2}(R))$ by replacing the subexpression with $\sigma_{c_2}(\sigma_{c_1}(R))$. Many such identities that arise in relational algebra are used in practice to optimize queries [42].

3.2 The Domain of Source Code

3.2.1 Many Data Types

An obvious difference between relational algebra and an algebra for source code is that the latter must handle many different kinds of data types. We will concern ourselves with source code written in C. The data types that arise in source code modeling can be classified into two broad groups:



TYPE HIERARCHY FOR C STATEMENTS

Figure 1:

- **Atomic data types**: These are the basic data types such as INTEGER, FLOAT, BOOLEAN, CHAR, STRING, etc. Unlike relational algebra, SCA treats these basic data types as elements of the algebra. This permits the introduction of operators such as $+$, $-$, **and**, **or**, etc. as valid algebra operators.
- **Composite data types (Objects)**: Some examples of composite data types in C are the `while-statement` type, the `relational-expression` type, and so on. Two different kinds of source code objects are modeled in SCA:
  - **Singular objects** such as a `while-statement`, an `identifier`, etc. Typically, these are constructs of the programming language which have a *syntactic structure* given by the abstract syntax of the language. For example, a `while-statement` object has two structural components, the *condition* (of type `expression`) and the *body* (of type `statement`). Singular objects are analogous to *nested relations* in the nested relational model [21, 22, 23].
  - **Collective objects**: These are collections of other objects. For example, the type `statement-list` represents a *sequence* of objects of type `statement`. Similarly, the type `declaration-list` represents a *set* of objects of type `declaration`.

3.2.2 Hierarchy of Data Types

An interesting feature that characterizes source code data types is the presence of a *type hierarchy* or *class hierarchy*. For example, `while-statements` are a subtype of the type `statements` (by specialization of *behavior*), in turn `statements` form a subtype of the type `program-objects`. Consequently, during query processing, it should be possible to substitute a `while-statement` in place of a `statement`, and a `statement` in place of a `program-object`. A pictorial representation of the C type hierarchy restricted to the type `statement` is shown in Figure 1.

A critical requirement in the design of SCA then must be the ability to incorporate the source code *type hierarchy* as an integral part of the algebraic framework. The algebra must handle the notion of subtyping

and inheritance, and support *substitutability*, a critical feature which lets an instance of a subtype be used in place of a supertype.

3.2.3 Object Attributes

There are four different kinds of *attributes* that may be associated with a source code object.

By *components*, we mean syntactic or structural information. In the case of a `while-statement` object, the components are its *condition* and *body*. Conceptually, a restriction of the source code representation with respect to component attributes would yield the abstract syntax tree of the program. Extracting structural information from source code and storing it in the source code database is a part of the source code parsing process.

By *relationships*, we refer to the associations between objects. In addition to simple cross-referencing information, they offer a way of modeling program flow relationships that occur between objects. One set of important data flow relationships in the source code domain model are the "uses" and "defines" relationships (see STATEMENT in Figure 2). If a statement `s` uses a variable `v`, a "uses" (and symmetrically, "used-by") exists between them. Similarly, if a statement `s` defines a variable `v`, a "defines" (and symmetrically, "defined-by") exists between them. Extracting and storing such information is the responsibility of flow analyzers.

An attribute of an object can also be a *method* or a function that is computed on-the-fly. Methods are a standard feature of object-oriented data models [13], and can be used to introduce complex and specialized algorithms into the data model. For example, efficient algorithms for data flow analysis such as *live variable analysis*, *available expression analysis*, etc. [24] can be used to compute the attributes such as "live" (see STATEMENT in Figure 2), which computes the set of live variables for a given statement, and their respective next statements in the "uses" chain. While the algebra, in principle, should be powerful enough for such computations, methods can be used as hooks to incorporate specialized algorithms on grounds of efficiency.

In addition to components, relationships, and methods, other kinds of information may be relevant to the problem of querying source code. Typical among these are information regarding line numbers, metrics, etc.

In addition to attributes that are precomputed or computed on-the-fly using methods, new attributes can be added to objects *during* a query. Such relationships can be thought of as *derived* attributes, and their computation should be part of a *view generation* process [42]. In section 3.3, we will introduce the **extend** operator, which lets such attributes to be added to objects.

3.2.4 A Suitable Algebra for Source Code

As seen in the previous sections, an algebra for source code marks a major departure from relational algebra because it must 1) support a wide variety of atomic and composite data types, and 2) incorporate the notion of a type hierarchy within the algebra itself.

The first condition can be satisfied if, instead of using the class of *one-sorted algebras*, we use the class of *many-sorted algebras* [4, 17] to model SCA. Unlike one-sorted algebras that model a single data type, many-sorted algebras can model a variety of atomic and composite data types and the operations on those types within a single algebraic framework.

However, to handle type hierarchies within the overall framework of many-sorted algebras, it is first necessary to define a *partial order* on the different types (sorts) of the algebra based on the *subtype of* or *subsort of* relationship. The issue of ordering the sorts of a many-sorted algebra was first addressed as a theoretical problem by Goguen and Meseguer [16] who proposed an *order-sorted algebra* based on the interpretation of subsorts (subtypes) as subsets. The interpretation of subsorts was later relaxed in the work of Bruce and Wegner on *generalized order-sorted algebras* to a weaker form of *behavioral compatibility*

```
type   IDENTIFIER-REF   subtype of EXPRESSION
                        ......
                        id:IDENTIFIER inverse id-references
                        ......
endtype
type   IDENTIFIER       subtype of PROGRAM-OBJECT
                        ......
                        name:STRING
                        id-references:SET(IDENTIFIER-REF) inverse id
                        ......
endtype
type   FUNC-CALL        subtype of EXPRESSION
                        ......
                        func:FUNCTION
                        arguments:EXPR-LIST
                        ......
endtype

type   FUNCTION         subtype of PROGRAM-OBJECT
                        ......
                        type-spec:TYPENAME
                        name:STRING
                        parameters:PARAM-LIST
                        body:COMPOUND-STMT
                        calls:SET(FUNCTION)
                        in-file:FILE inverse funcs
                        ......
endtype

type   FILE             subtype of PROGRAM-OBJECT
                        ......
                        name:STRING
                        funcs:SET(FUNCTION) inverse in-file
                        decls:SET(DECLARATION)
                        ......
endtype

type   STATEMENT        subtype of PROGRAM-OBJECT
                        ......
                        line-no:SET(FUNCTION)
                        uses:SET(VARIABLE) inverse used-by
                        defines:SET(VARIABLE) inverse defined-by
                        ......
endtype
.....
```

Figure 2: Type definitions

[6]. Essentially, a sort is a subsort of another if the former is behaviorally compatible with (i.e., can be substituted for) the latter.

A generalized order-sorted algebra is thus a many-sorted algebra with a partial order defined on its sorts. Intuitively, it is apparent that SCA can be modeled as a generalized order-sorted algebra where the sorts are the various source code data types (atomic and composite) ordered by the *subtype of* relationship. The concept of behavioral compatibility is particularly suitable because a `while-statement` is indeed a *behavioral subtype* of a `statement` (as opposed to being a subset or a restriction) since it contains additional attributes.

We now use a formal definition of generalized order-sorted algebras to characterize SCA:

Definition 1: Let S be a set of sorts. In SCA, S contains all the atomic data types and composite data types discussed in section 3.2.1. Thus,

ATOM = {INTEGER,BOOLEAN,FLOAT,...}
COMP = {`while-statement`,...,`statement`,...,`statement-list`,...}
S = ATOM $\cup$ COMP

Definition 2: A generalized order-sorted algebra A is a 3-tuple $< S, \leq, OP >$, where S is a set of sorts, $\leq$ a partial ordering defined on the sorts, and OP a set of functions (called the operator set) such that:

1. a set A_s, called the *carrier set* or *domain* of s, is defined for each $s \in S$
2. the signature of a function $\sigma \in OP$ is given by
   $\sigma : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n} \longrightarrow A_s$
   where $s_1, s_2, \ldots, s_n, s$ are elements of S.
3. if $t \leq s_i$, then elements of A_t can *substitute* as elements of A_{s_i}.

In SCA, the partial order $\leq$ is given by the *subtype of* relationship. The set OP contains operators for atomic data, objects, and collections of object such as sets and sequences. The details of the SCA operators are presented in the next section.

3.3 Source Code Algebra Operators

Given the source code data model in SCA, the next task is to define the algebra operators that are relevant to the task of querying source code. SCA is essentially an algebra of objects. We have used and extended operators from pre-existing object algebras for set operations, generalizing them to operate on sequences wherever possible, and proposed appropriate operators for sequences. Operators for sequences have only recently begun to be proposed in literature [15, 37]. We have introduced **seq_extract**, a powerful new operator for sequences which uses regular expressions as the basis for extracting subsequences. SCA offers a unified approach to querying *collections*, whether they be sets or sequences. This is a departure from earlier approaches where the data model is either essentially set-oriented or sequence-oriented. Using the SCA operators, source code queries can be expressed as algebraic expressions. An evaluation of an algebraic expression on the source code representation yields the result of the query.

The operators of SCA can be classified into two broad categories. Table 2 shows SCA operators defined on atomic data types. The second category of SCA operators are defined on objects (composite data types). Operators shown in Table 3 are defined on all objects. Table 4 shows operators defined on collections, i.e., both sets and sequences. Operators specific to sets and sequences are shown in Tables 5 and 6 respectively.

Operator	Signature
+,−,*	INT × INT ⟶ INT FLOAT × FLOAT ⟶ FLOAT
/	FLOAT× FLOAT ⟶ FLOAT
and,**or**	BOOL × BOOL ⟶ BOOL
not	BOOL ⟶ BOOL
=,<,>,≤,≥	ATOM × ATOM ⟶ BOOL

Table 2: SCA Operators for Atomic Data Types

Operator	Signature
closure	COMP ⟶ SET(COMP)
equal	COMP1 × COMP1 ⟶ BOOL
identical	COMP1 × COMP1 ⟶ BOOL

Table 3: SCA Operators for Composite Data Types

In the remainder of this section, we will discuss the semantics of some of the interesting operators of SCA. A complete description of SCA operators can be found in [32].

3.3.1 closure

This computes the transitive closure, or reachability graph, given a attribute name. For a given object and an attribute name, **closure** produces a set of all objects that are reachable from the original object using the only the named attribute as 'links'.
syntax: $closure_{<attribute>}(< object >)$
For example, **closure** on the attribute "call" would result in all the functions directly and indirectly called by a function.

3.3.2 select

Given a set (sequence) of objects and an algebraic expression that evaluates to a boolean value, **select** returns a subset (subsequence) of the objects for which the expression evaluates to TRUE. This has been extended from **select** in relational algebra. **select** produces a *subclass* of objects of the same type.
syntax: $select_{<boolean\ expression>}(< objectcollection >)$

3.3.3 project

Given a collection of objects and a list of valid attributes of the objects, it returns a collection of new objects which contain only the listed attributes. This is extended from the **project** operator in relational algebra. **project** produces a *superclass* of the input class, since the resulting type is a supertype of input type by generalization (as it has fewer attributes).
syntax: $project_{<attributelist>}(< objectcollection >)$

Operators	Signature
select	COLLECTION(ANY1) $\longrightarrow$ COLLECTION(ANY1)
project	COLLECTION(COMP1) $\longrightarrow$ COLLECTION(COMP2)
extend	COLLECTION(COMP1) $\longrightarrow$ COLLECTION(COMP2)
retrieve	COLLECTION(COMP) $\longrightarrow$ COLLECTION(ANY)
apply	COLLECTION(ANY1) $\longrightarrow$ COLLECTION(ANY2)
product	COLLECTION(COMP1) $\times$ COLLECTION(COMP2) $\longrightarrow$ SET(COMP3)
flatten	COLLECTION(COLLECTION(COMP1)) $\longrightarrow$ COLLECTION(COMP1)
pick	COLLECTION(ANY1) $\longrightarrow$ ANY1
size_of	COLLECTION(ANY) $\longrightarrow$ INT
reduce	COLLECTION(ANY) $\longrightarrow$ ANY
forall,exists	COLLECTION(ANY) $\longrightarrow$ BOOL
is_empty	COLLECTION(ANY) $\longrightarrow$ BOOL
member_of	ANY1 $\times$ COLLECTION(ANY1) $\longrightarrow$ BOOL

Table 4: SCA Operators for Collections (sets and sequences)

3.3.4 extend

Given a collection of objects, a function name, and an algebraic expression, **extend** returns a collection of new objects which have all the attributes of the input objects and also a new attribute whose value is obtained by evaluating the algebraic expression. This is equivalent to the λ operator in NST-Algebra [19], and the **extend** operator in Schek and Scholl's extended relational algebra [39]. **extend** produces a *subclass* of the input class, since the resulting type is a subtype of the input type by specialization (as it has more attributes).
syntax: $extend_{<attribute:=algebraic\ expression>}(< objectcollection >)$

3.3.5 retrieve

Given a collection of objects, this operator retrieves the specified attribute (field) for each object. The attribute may be specified using its index or its name.
syntax: $retrieve_{<attribute>}(< objectcollection >)$

3.3.6 apply

Given a collection and a unary operator, **apply** returns a new collection where the objects are obtained by applying the operator to each of the input objects.
syntax: $apply_{<operator>}(< objectcollection >)$

3.3.7 Cartesian Product

Given two collections of objects, it returns a set where the objects are obtained by systematically combining all possible pairs of objects between the two collections. Extended from **cartesian product** in relational algebra.
syntax: $product(< objectcollection1 >, < objectcollection2 >)$

Operator	Signature
union,difference,intersection	SET(ANY1) × SET(ANY1) ⟶ SET(ANY1)
subset_of	SET(ANY1) × SET(ANY1) ⟶ BOOL
set_to_seq	SET(ANY1) ⟶ SEQ(ANY1)

Table 5: SCA Operators specific to Sets

Operator	Signature
head,tail	SEQ(ANY1) ⟶ SEQ(ANY1)
concat	SEQ(ANY1) × SEQ(ANY1) ⟶ SEQ(ANY1)
order	SEQ(ANY1) ⟶ SEQ(ANY1)
seq_extract	SEQ(ANY1) ⟶ SEQ(ANY1)
seq_element	SEQ(ANY1) ⟶ ANY1
subseq_of	SEQ(ANY1) × SEQ(ANY1) ⟶ BOOL
seq_to_set	SEQ(ANY1) ⟶ SET(ANY1)

Table 6: SCA Operators specific to Sequences

3.3.8 forall

forall returns TRUE if for all elements in the collection, the boolean expression evaluates to TRUE.
syntax: $forall_{<boolean\ expression>}(< objectcollection >)$

This is a derived operator, whose semantics is equivalent to the truth of the expression:
$size\_of(select_{<boolean\ expression>}(< objectcollection >)) = size\_of(< objectcollection >)$

3.3.9 exists

exists returns TRUE if for some element in the collection, the boolean expression evaluates to TRUE.
syntax: $exists_{<boolean\ expression>}(< objectcollection >)$

This is a derived operator, whose semantics is equivalent to the truth of the expression:
$size\_of(select_{<boolean\ expression>}(< objectcollection >)) \neq 0$.

3.3.10 seq_extract

The choice of appropriate sequence operators is a topic of current research [37]. Existing sequence manipulation languages provide little or no support for extracting subsequences based on sequence patterns. We have attempted to address this problem by introducing the **seq_extract** operator.
syntax: $seq\_extract_{<pattern>:<boolean\ expression>}(< objectseq >)$
The $< pattern >$ is a regular expression [25]. An example of a pattern could be
`(while-statement,statement*,if-statement,statement*,while-statement)` in which case it would return a subsequence of the input sequence which starts and ends with a `while-statement`, and has a `if-statement` somewhere in between. Additional constraints about the pattern can be expressed using the the $< boolean\ expression >$.

3.3.11 order

order accepts a sequence and returns a sequence ordered by 1) the values of the objects if it is a sequence of atomic data items, or 2) the values of the attribute if the elements are objects.
syntax: $order_{<attribute>,<ord>}(< objectseq >)$
The order returned is increasing or decreasing based on whether the parameter $< ord >$ is $' <'$ or $' >'$.

4 Source Code Queries as SCA Expressions

We now demonstrate the power of SCA by expressing some source code as SCA expressions. This exercise shows the use of SCA as a low level source code query language. We hope to show the feasibility of modeling and querying information related to both program structure as well as program flow within the framework of SCA.

1. *What are the functions defined in the file* `analyzer.c`*?*:

   First, the file `anaylzer.c` is selected, and then its attribute "funcs" (the set of functions defined in the file) is retrieved.

   $retrieve_{funcs}(select_{name=analyzer.c}(FILE))$

2. *Show the body of the function* `sort()`*.*:

   The function `sort()` is selected and its "body" retrieved.

   $retrieve_{body}(select_{name=sort}(FUNCTION))$

3. *Find all the iterative statements in the program.*:

   The objects of type `iteration-statement` are selected. This includes all objects of types `do-statement`, `while-statement`, and `for-statement` (the subtypes of `iteration-statement`).

   $select_{TRUE}(ITERATION - STATEMENT)$

4. *Find the file that has the maximum number of functions*:

   First, the file objects are extended with a new field, namely no_of_func. The set of these new objects is then converted into a sequence and arranged in decreasing order of their no_of_func. The head of this sequence is the file with maximum functions.

   $head_1(order_{no\_of\_func,>}(set\_to\_seq(extend_{no\_of\_func:=size\_of(funcs)}(FILE))))$

5. *Find a sequence of three* `if-statements`*, possibly separated by arbitrary statement lists*:

   The unary operator **seq-extract** operates on every `statement-list` and extracts, wherever applicable, the subsequences that fit the pattern of three `if-statements` with other statements in between.

   $image_{seq\_extract_{(ifstatement,statement*,ifstatement,statement*,ifstatement):TRUE}}(STATEMENT - LIST)$

6. *Find all references to the identifier* `counter`.

   Each identifier object has an attribute named `ide-references` that points to the set of expressions that refer to it.

   $retrieve_{id-references}(select_{name=counter}(IDENTIFIER))$

7. *Identify the set of all functions that are directly or indirectly invoked by the function* `sort()`.:

   This is a clear instance of transitive closure based on the 'calls' relation.

   $closure_{calls}(select_{name=sort}(FUNCTION))$

8. *Which functions in* `until.c` *are called by functions in* `main.c`*?*

   First, the functions in file `main.c` are selected. For each function, the set of functions it calls are obtained (using the **image** operator). The result is a set of set of functions. The **flatten** operator flattens the nested set into a single set of functions. The **select** operator then picks out only those that are defined in `until.c`.

   $select_{in-file=until.c}(flatten(retrieve_{calls}(select_{in-file=main.c}(FUNCTION))))$

9. *Find all subsequent uses of the variable defined in statement* `s`.

   The variable defined in `s` is selected, and then all its uses are retrieved.

   $flatten(retrieve_{used-by}(retrieve_{defines}(select_{s}(STATEMENT))))$

5 Incorporating SCA into a Reverse Engineering System

Figure 3 shows how SCA would fit into the design of a query system. Source code files are processed using tools such as parsers, static analyzers, etc. and the necessary information (according to the SCA data model) is stored in a repository. A user interacts with the system, in principle, through a variety of high-level languages, or by specifying SCA expressions directly. Queries are mapped to SCA expressions, the SCA optimizer tries to simplify the expressions, and finally, the SCA evaluator evaluates the expression and returns the results to the user.

We expect that many source code queries will be expressed using high-level query languages or invoked through graphical user interfaces. High-level queries in the appropriate form (e.g., graphical, command-line, relational, or pattern-based) will be translated into equivalent SCA expressions. An SCA expression can then be evaluated using a standard SCA evaluator, which will serve as a common query processing engine. The analogy from relational database systems is the translation of SQL to expressions based on relational algebra.

Where high-level queries available to the user are not sufficiently expressive, the SCA itself can be used as a low-level source code query language. Users familiar with SCA can exploit the power of the algebra by expressing queries directly as SCA expressions, thus bypassing the high-level query interface. Queries that involve structural as well as flow information are ideal candidates for such treatment.

An obvious issue in the above architecture is whether SCA expressions can be evaluated efficiently. While the study of SCA optimization is currently in progress, we have strong grounds to believe that important performance gains can be achieved using our approach. One reason is that many of the set

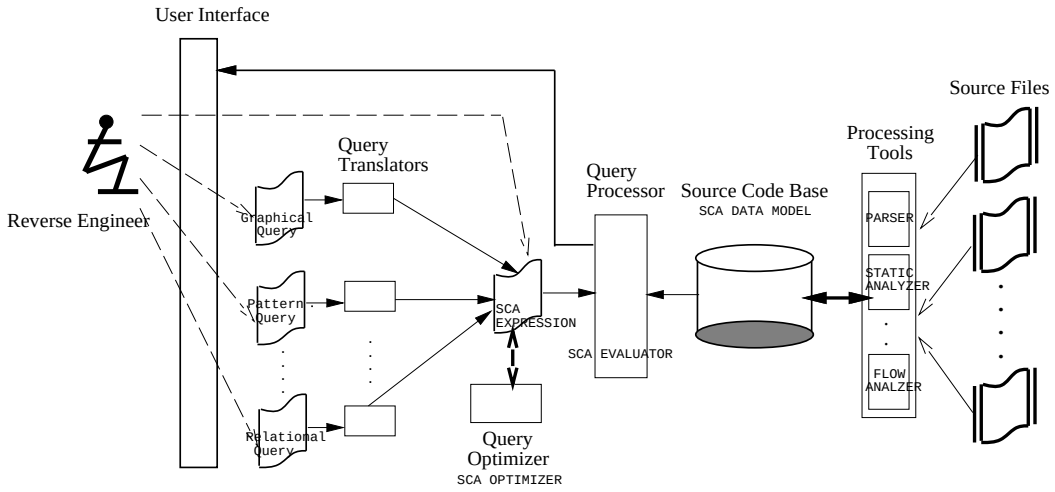


Figure 3: SCA-based Source Code Query System

operators in SCA are extended from relational and extended relational algebras, for which optimizations already exist [31, 39]. Furthermore, many sequence operators introduced in SCA (such as **seq-extract**) can be implemented using efficient algorithms developed in our work on the SCRUPLE system.

Obviously, the above is only an outline of the ideas required to incorporate the framework in a query system. Open issues exist and a more complete discussion would go beyond the scope of the paper. However, what we have attempted to do is show that the model is worth pursuring because of the following merits. First, given a query processor based on the model, it would ease the design of source code query systems. Second, techniques used for optimizing queries in other algebraic models can also be applied to our model. And third, whenever available query optimizers are not good enough, our model allows designers to incorporate specialized code analysis algorithms easily into the model using method attributes.

To further investigate design and implementation issues, a prototype of the current system is being built on top of Gemstone, an object-oriented database system [7]. Several key components such as parser and query processor for handling sequences have been built and were tested in the SCRUPLE system for pattern matching. Performance results for operating on sequences were promising and are available in [34].

6 Comparison of SCA with other Query Algebras

The most well-known query algebra is the relational algebra. Query languages for the nested and extended relational models have also been developed by relaxing the first normal form restriction of relational algebra [21, 22, 23, 38]. The primary data type in these models is the *relation*, which is a *set* of tuples.

Inspired by the relational model, some object-oriented database systems have attempted to develop object algebras to serve as a basis for their query languages. Some of these algebras are the PDM algebra [29], Osborn's algebra [31], Straube and Ozsu's algebra [41], and Shaw and Zdonik's algebra [40]. The object algebras treat all their data types as first class objects, and compared to relational algebra, permit considerably more orthogonality between objects and type constructors. Object algebras differ from one another in the range of their supported types and, more importantly, in their operators. One of the major drawbacks of these algebras is that they fail to provide modeling and operator support for data type collections such as sequences. Like relational algebra, object algebras are essentially set-oriented.

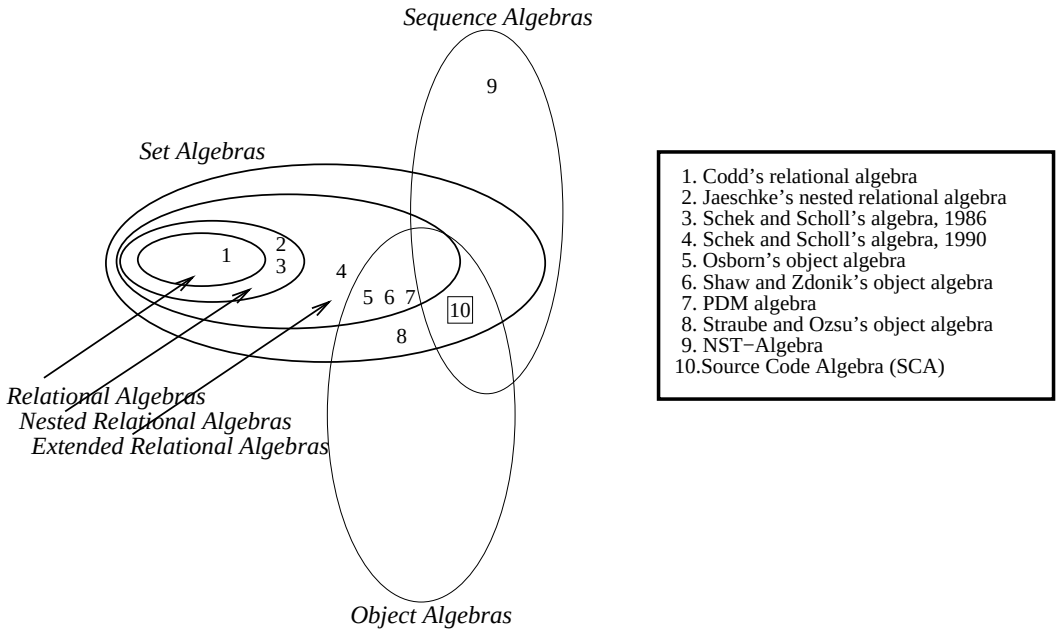


Figure 4: SCA in comparison with other Query Algebras

Unlike set algebras, the field of sequence algebras is in its infancy. The NST-algebra (*Nested Sequence of Tuples*) [19] is a many-sorted algebra used as a query language for structured office documents, a domain where nested sequences arise naturally. Documents are modeled as *nested sequences of tuples* (NST). However, there is no support for extracting subsequences in NST-algebra.

Figure 4 shows the world of query algebras using a Venn diagram. Different query algebras are positioned in the diagram based on the data types supported by them. It shows that the relational algebras (1,2 and 3) fall within the larger class of set algebras. It shows that Osborn's object algebra (5) supports objects and sets, but does not support sequences. Similarly, NST-Algebra supports sequences, but does not support sets.

Since SCA (10) is essentially an algebra of objects, sets, and sequences, it belongs to the intersection set of object, set, and sequence algebras.

7 Conclusion

We began this paper by presenting the requirements of a source code query system. A useful source code query system must model information pertaining to program structure (global as well as fine-grained) and program flow in a seamless manner. A powerful query language should then be used to extract the information present in the model.

We introduced Source Code Algebra (SCA), a formal framework that models source code as an algebra of objects. Our solution views source code as a domain of strongly-typed objects (and their collections) with attributes, and supports type hierarchies as an integral part of the model. A set of well-defined algebraic operators are defined to extract information from the model. Theoretically, SCA belongs to the class of generalized order-sorted algebras.

Modeling source code as an algebra has important benefits in terms of query languages. We have shown,

with examples, how SCA can be used as a low-level source code query language. Queries written in high-level query languages (commandline, graphical, pattern-based, etc.) can also be processed by mapping them to equivalent SCA expressions and evaluating them using a standard SCA evaluator. Since SCA expressions can be simplified using rules of algebraic transformation, source code queries mapped to SCA expressions can be optimized. From a theoretical point of view, high level query languages built on top of SCA will have well-defined semantics.

The implementation of a prototype source code query system based on SCA is in progress. The prototype will allow us to investigate issues such as SCA optimization strategies, and view generation mechanisms similar to the relational model.

References

[1] R. Al-Zoubi and A. Prakash. Software Change Analysis via Attributed Dependency Graphs. Technical Report CSE-TR-95-91, Dept. of EECS, University of Michigan, May 1991. Also in *Software Maintenance*, to appear.

[2] J. Ambras and V. O'Day. Microscope: A Program Analysis System. In *Proc. of the 20th Hawaii International Conference on System Sciences*, pages 460–468, 1987.

[3] T. Biggerstaff, B.G. Mitbander, and D. Webster. The Concept Assignment Problem in Program Understanding. In *Proc. of the 15th International Conference on Software Engineering*, pages 482–498, 1993.

[4] G. Birkhoff and D. Lipson. Heterogeneous Algebras. *Journal of Combinatorial Theory*, 8:115–133, 1970.

[5] R. Brooks. Towards a Theory of Comprehension of Computer Programs. *International Journal of Man Machine Studies*, 18:543–554, 1983.

[6] K. Bruce and P. Wegner. *Advances in Database Programming Languages*, chapter An Algebraic Model of Subtype and Inheritance. ACM Press, 1990.

[7] P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):50–77, October 1991.

[8] Y. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[9] E. Chikofsky. State-of-Art Talk on Reverse Engineering. Invited Talk at the 15th International Conference on Software Engineering, Baltimore, Maryland., May 1993.

[10] L. Cleveland. A Program Understanding Support Environment. *IBM Systems Journal*, 28(2):324–344, 1989.

[11] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] T.A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[13] M. Atkinson et al. The Object-Oriented Database System Manifesto. Technical Report ALTAIR TR 30-89, GIP ALTAIR, LeChesnay, France, 1989.

[14] M. Stonebraker et al. Third-generation database system manifesto. *ACM SIGMOD Record*, 19(3), 1990.

[15] S. Ginsburg and X. Wang. Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data. In *Proc. of the 11th ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 293–300, 1992.

[16] J. Goguen and J. Meseguer. Extensions and Foundations of Object-oriented Programming. Technical report, SRI, 1986.

[17] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *Current Trends in Programming Methodology*, volume IV, chapter An Initial Algebra Approach to the specification, correctness, and implementation of abstract data types. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[18] J.E. Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems: The Journal of the USENIX Association*, 5(1):5–67, Winter 1992.

[19] R.H. Guting, R. Zicari, and D.M. Choy. An Algebra for Structured Office Documents. *ACM Transactions on Office Information Systems*, 7(4):123–157, 1989.

[20] K. Heisler, Y. Kasho, and W.T. Tsai. A Reverse Engineering Model for C Programs. *Information Sciences*, 68:155–193, February 1993.

[21] G. Jaeschke. An Algebra of power set type relations. Technical Report TR 82.12.002, IBM Heidelberg Scientific Center, Heidelberg, Germany, 1982.

[22] G. Jaeschke. Nonrecursive Algebra for relations with relation-valued attributes. Technical Report TR 85.03.001, IBM Heidelberg Scientific Center, Heidelberg, Germany, 1985.

[23] G. Jaeschke. Recursive Algebra for relations with relation-valued attributes. Technical Report TR 85.03.002, IBM Heidelberg Scientific Center, Heidelberg, Germany, 1985.

[24] K. Kennedy. *Program Flow Analysis: theory and applications*, chapter 1. Prentice-Hall, 1981.

[25] B.W. Kernighan and R. Pike. *The UNIX Programming Environment.* Prentice-Hall, 1984.

[26] G.B. Kotik and L.Z. Markosian. Automating Software Analysis and Testing Using a Program Transformation System. In *Proceedings of ACM SIGSOFT*, pages 75–84, 1989.

[27] W. Kozaczynsky, J. Ning, and A. Engberts. Program Concept Recognition and Transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[28] M.A. Linton. Implementing Relational Views of Programs. In *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium*, May 1984. Practical Software Development Environment.

[29] F. Manola and U. Dayal. PDM: an Object-oriented Data Model. In *Proc. of Intl. Workshop on Object-oriented Database Systems*, pages 18–25, September 1986.

[30] H.A. Muller, B.D. Corrie, and S.R. Tilley. Spatial and Visual Representations of Software Structures: A model for reverse engineering. Technical Report TR-74.086, IBM Canada Ltd., April 1992.

[31] S.L. Osborn. Identity, Equality and Query Optimization. In *2nd Intl. Workshop on Object-oriented Database Systems*, pages 346–351. Springer-Verlag, September 1988.

[32] S. Paul. *Theory and Design of Source Code Search Systems.* PhD thesis, University of Michigan, 1994. In preparation.

[33] S. Paul and A. Prakash. Source Code Retrieval Using Program Patterns. In *Proc. of the 5th International Conference on Computer Aided Software Engineering*, pages 95–105, 1992.

[34] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 1994. Accepted for publication.

[35] C.V. Ramamoorthy, Y. Usuda, A. Prakash, and W.T. Tsai. The Evolution Support Environment System. *IEEE Transactions on Software Engineering*, pages 1225–1234, November 1990.

[36] C. Rich and R. Waters. *The Programmer's Apprentice*. Addison-Wesley, Baltimore, Maryland, 1990.

[37] J. Richardson. Supporting Lists in a Data Model. In *Proc. of the 18th VLDB Conference*, pages 127–138, 1992.

[38] H.J. Schek and M.H. Scholl. An Algebra for the relational model with relation-valued attributes. *Information Systems*, 11:137–147, 1986.

[39] H.J. Schek and M.H. Scholl. A Relational Object Model. In *3rd Intl. Conference on Database Theory*, pages 89–105. Springer-Verlag, 1990.

[40] G.M. Shaw and S.B. Zdonik. An Object-oriented Query Algebra. *Bulletin of IEEE technical committee on Data Engineering*, 12(3):29–36, 1989.

[41] D.D. Straube and M.T. Ozsu. Queries and Query processing in Object-oriented Database Systems. *ACM Transactions on Information Systems*, 8(4), October 1990.

[42] J.D. Ullman. *Principles of Database Systems*. Computer Science Press International, Rockville, Maryland, 1990.

[43] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, June 1984.

[44] E. Yourdon. RE-3. *American Programmer*, 2(4):3–10, April 1989.