Supporting Queries on Source Code: A Formal Framework

Santanu Paul * Atul Prakash

Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
email: {santanu,aprakash}@eecs.umich.edu

Contact Addresses:

Prof. Atul Prakash **(Primary Contact)**
Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122.
Office phone: (313) 763-1585
FAX: (313) 763-1503
email: aprakash@eecs.umich.edu

*This research was supported in part by a fellowship from IBM Canada Ltd.

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) that forms the basis of our source code query system. 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 expressing them as equivalent SCA expressions, 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* [1].

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 [1]. 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 [2]. 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: an interactive 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 [3, 4, 5]. The need for sophisticated querying tools for reverse engineering has been articulated by Biggerstaff in terms of a "conceptual `grep`" [6]. The purpose of a source code querying tool is to help a human reverse engineer indulge in *plausible reasoning* [6] or *domain bridging* [7] — 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* [8, 9]. 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 [10], 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 [11] and SCRUPLE [8], and earlier in our work on the Evolution Support Environment System (ESE) [12], 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 [5], CIA [3], and CIA++ [13]. 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 in Rigi [14] or an object-based representation as used in REFINE [1] [15] and in the approach of Heisler et al [16]. 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

[1] REFINE is a trademark of Reasoning Systems

up using an attributed syntax-tree representation whereas the ESE system used a relational representation. In both cases, we felt a definite lack of either a powerful 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* [17] 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 [17], the nested relational model [18], the extended relational model [19], the object model [20, 21, 22, 23], and also in the design of a domain-specific query language for structured office documents [24]. 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 information, control-flow information, etc.

  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 be based on 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 [17], relational calculus [25], NST-Algebra [2][24], 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 detailed 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 [26].

[2]NST stands for *Nested Sequence of Tuples*

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

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* ($\times$) [17]. *Join* ($\bowtie$) is a derived operator of the algebra (composition of σ and $\times$). Each of these operators takes relations

as arguments, and produces 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* [17]. Relational algebra has also been shown to be equivalent to *relational calculus* [25]. 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 [25]. 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 [24].

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 contained 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 [25].

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:

- **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 [18].

  - **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, namely, *components*, *references*, *annotations*, and *methods*.

Components model 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.

References model the associations between objects. In addition to simple cross-referencing information, they offer a way of modeling resource 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.

Annotations are used to store all other relevant information about source code objects. Typical annotations to a source code object are line numbers, metrics, etc.

An attribute of an object can also be a *method* or a function that is computed on-the-fly. Methods are usually computed to obtain reference or annotation information, *during query*

execution. Methods are a standard feature of object-oriented data models [27], 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. [28] 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.

It is important to point out here that new attributes may be *added* to objects *during* a query. These can be thought of as *derived* attributes, and their computation is analogous to the *view generation* problem in relational databases [25]. In section 3.3, we will introduce the **extend** operator, which lets new 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* [29, 30] 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 [31] 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* [32]. 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 objects 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. 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 [33, 34]. 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.

It is necessary to point out here that the SCA query language is *domain independent* with respect to the programming language of the source code. This is an extremely valuable feature (as seen in the case of relational query languages), and essentially means that an implementation of a SCA query processor would work unchanged across different SCA domain models. Basically, SCA provides a set of query operators that can be used to query a graph (or network) of generic objects. Some of these operators are customized to work well in the domain of source code, nevertheless, they are not tied to any specific programming language. Together, the SCA operators constitute an object algebra for source code. The

SCA domain model however, varies from one programming language to another and must be redesigned to reflect the specifics of a given programming language.

The operators of SCA can be classified into different categories based on the data types they operate upon. Table 2 shows SCA operators defined on atomic data types. Operators shown in Table 3 are defined on individual 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.

In the remainder of this section, we will discuss the semantics of SCA operators.

3.3.1 Operators for Individual Objects

The operators on individual objects are shown in Table 3 and briefly described below:

$< attribute >$

Given an object and an attribute name of the object, $< attribute > (object)$ returns the value of the given attribute. For example, if f is a file, then the *name* attribute of f is given by $name(f)$.

If the attribute is a *method*, its value must be computed before it is returned.

closure

Given an object and a list of attribute names, **closure** computes a transitive closure, or reachability graph. **closure** finds the set of all objects reachable from the original object using only the named attributes as 'links'. For example, **closure** on the attribute "calls" would result in all the functions directly and indirectly called by a function.

identical

Given two objects, **identical** tests whether they are the same object.

3.3.2 Operators for Collections

The operators on object collections are shown in Table 4 and briefly described below:

select, **pick**

These operators act as filters.

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.

Given a set (sequence) of objects and an algebraic expression that evaluates to a boolean value, **pick** returns a single object for which the expression evaluates to TRUE. If there are multiple candidate objects for which the expression evaluates to TRUE, any one of them may be returned.

project, **extend**, **product**

These are restructuring operators, i.e., they can be used to create new types of objects.

Given a collection of objects and a list of valid attributes of the objects, **project** returns a collection of new objects which contain only the listed attributes. This is extended from the **project** operator in relational algebra. It also offers a way to rename attributes.

Given a collection of objects, a new attribute name, and an algebraic expression, **extend** returns a collection of new objects which have all the attributes of the input objects and also the new attribute whose value is obtained by evaluating the algebraic expression. This is equivalent to the λ operator in NST-Algebra [24], and the **extend** operator in Schek and Scholl's extended relational algebra [35]. **extend** is an extremely powerful operator that allows new attributes to be added to existing type definitions.

Given two collections of objects, **product** returns a set of objects obtained by systematically combining all possible pairs of objects between the two collections. This operator is

similar to the **cartesian product** in relational algebra.

forall, **exists**, **member_of**

These are boolean operators, i.e, they can be used to test the truth or falsehood of assertions.

Given a collection of elements and an algebraic expression that evaluates to a boolean value, **forall** returns TRUE if for all elements in the collection, the boolean expression evaluates to TRUE. This is a derived operator, whose semantics is equivalent to the truth of the expression:

$$\boxed{\textbf{size\_of}(\textbf{select}_{<boolean\ expression>}(< objectcollection >)) = \textbf{size\_of}(< objectcollection >)}$$

Given a collection of elements and an algebraic expression that evaluates to a boolean value, **exists** returns TRUE if for some element in the collection, the boolean expression evaluates to TRUE. This is a derived operator, whose semantics is equivalent to the truth of the expression:

$$\boxed{\textbf{size\_of}(\textbf{select}_{<boolean\ expression>}(< objectcollection >)) \neq 0}.$$

Given a collection of elements and an element, **member_of** returns TRUE if the element belongs to the collection. If the element is an object, the object identity is used to decide membership.

apply, **reduce**

These are higher-order operators, i.e, they involve the application of other operators to a collection of elements.

Given a collection and a unary operator, **apply** returns a new collection obtained by applying the operator to each element in the input set. This is similar to the *lambda abstraction* in *lambda calculus*.

Given a collection and a binary operator, **reduce** returns a new collection obtained by applying the operator recursively over the elements in the input set.

retrieve

Given a collection of objects and an attribute name, **retrieve** returns a collection comprising the values of the specified attribute for each object in the initial collection. This is a derived operator, whose semantics can be expressed as:

$\boxed{\textbf{apply}_{<attribute>}(< objectcollection >)}$

size_of

Given collection, **size_of** returns the size of the collection.

flatten

Given a collection of collections, **flatten** reduces the nesting by merging the member collections into one collection.

3.3.3 Set Operators

Table 5 shows operators that operate exclusively on sets. They are briefly described below:

union, **intersection**, **difference**

The definitions of these operators are derived from their equivalents in relational algebra.

subset_of

Given two sets, **subset_of** returns TRUE if the first is a subset of the second.

set_to_seq

Given a set, **set_to_seq** returns a sequence consisting of the same elements. The choice of the order of elements is arbitrary.

3.3.4 Sequence Operators

Table 6 shows operators that operate exclusively on sequences. They are briefly described below:

head, **tail**

Given a sequence of elements, and a number n, **head** returns a sequence consisting of the first n elements, and **tail** returns a sequence consisting of the last n elements.

concat

Given two sequences, **concat** returns a concatenated sequence.

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. The order returned is increasing or decreasing based on whether the parameter $< ord >$ is $'<'$ or $'>'$.

seq_extract

Given a sequence and a regular expression (the $<$ *pattern* $>$), **seq_extract** returns a subsequence of the input sequence which matches the regular expression. Additional constraints about the pattern can be expressed using an assertion (the $<$ *boolean expression* $>$).

For example, a $<$ *pattern* $>$ could be:

```
(while-statement,statement*,if-statement,statement*,while-statement)
```

seq_extract would then return a subsequence of the input sequence which starts and ends with a `while-statement`, and has an `if-statement` somewhere in between.

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.

seq_element

Given a sequence and an integer-valued expression (the $< index >$), **seq_element** returns the sequence element at position $< index >$.

subseq_of

Given two sequences, **subseq_of** returns TRUE if the first is a subsequence of the second.

seq_to_set

Given a sequence, **seq_to_set** returns a set consisting of the same elements.

4 Source Code Queries as SCA Expressions

We now demonstrate the power of SCA by expressing some C source code queries as SCA expressions. This exercise shows the use of SCA as a low level source code query language.

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

$$\boxed{funcs(\mathbf{pick}_{name='analyzer.c}(\text{FILE}))}$$

First, the file `anaylzer.c` is selected, and then its attribute *funcs* (the set of func-

tions defined in the file) is retrieved.

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

$$\boxed{body(\mathbf{pick}_{name='sort}(\text{FUNCTION}))}$$

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

3. *Query: Find the number of iterative statements in the program.*:

$$\boxed{\mathbf{size\_of}(\text{ITERATION-STATEMENT})}$$

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

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

$$\boxed{\mathbf{head}_1(\mathbf{order}_{no\_of\_func,>}(\mathbf{set\_to\_seq}(\mathbf{extend}_{no\_of\_func:=\mathbf{size\_of}(funcs)}(\text{FILE}))))}$$

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.

5. *Query: Find all sequences of two* `if-statements`*, possibly separated by arbitrary statement lists*:

$$\boxed{\mathbf{apply}_{\mathbf{seq\_extract}_{(ifstatement,statement*,ifstatement):TRUE}}(\text{STATEMENTLIST})}$$

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

6. *Query: Create a new view of caller-callee relationships, such that if function A directly calls function B, B is contained in the callee set of A*:

$$\boxed{\mathbf{extend}_{calls:=}\mathbf{retrieve}_{funcdef}(\mathbf{select}_{objecttype='FUNC-CALL}(\mathbf{closure}_{component}))(\text{FUNCTION})}$$

This is an instance of transitive closure computation. **closure** (on component attributes) is used to find all nodes within the definition of a function. From these nodes, only FUNC-CALL nodes are selected, and the respective FUNCTION nodes retrieved. The set of these callee FUNCTION nodes is assigned to the *calls* attribute for each caller function. Note how **extend** has been used to generate a new *view* (calls relationships) of the source code database.

7. *Query: Find all the functions directly or indirectly called by function `sort()`:*

$$\boxed{\mathbf{closure}_{calls}(\mathbf{pick}_{name='sort}(\text{FUNCTION})}$$

This query uses the view defined in the previous query, namely the *calls* relationship between functions. By computing the transitive closure on *calls* links, the entire call graph starting at `sort()` can be easily generated.

8. *Query: Which functions in `analyzer.c` are called by functions in `main.c`?*:

$$\textbf{intersection}(funcs(\textbf{pick}_{name='analyzer.c}\ (\text{FILE})),\ \textbf{flatten}(\textbf{apply}_{\textbf{closure}_{calls}}\ (funcs(\textbf{pick}_{name='main.c}\ (\text{FILE})))))$$

First, the functions in file `analyzer.c` are selected. Second, the functions in file `main.c` are selected, and for each function, the set of functions it calls are obtained (using the **apply** and **closure** operators). The result is a set of sets of functions. The **flatten** operator flattens the nested set into a single set of functions called from the file `main.c`. The **intersection** operator is then used to locate only those functions that are defined in `analyzer.c`.

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 operators in SCA are extended from relational and extended relational algebras, for which optimizations already exist [21, 35]. 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 pursuing 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 using REFINE, an object-oriented source code database system [36]. Several key components such as the parser and the query processor for handling objects, col-

lections, and sets have been built. Sequence algorithms were tested in the SCRUPLE system and need to be incorporated into the query processor. Performance results for operating on sequences were promising and are available in [8].

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 [18, 19]. 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 [20], Osborn's algebra [21], Straube and Ozsu's algebra [23], and Shaw and Zdonik's algebra [22]. 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.

Unlike set algebras, the field of sequence algebras is in its infancy. The NST-algebra [24] 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 (command-line, 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 those in the relational model.

Acknowledgments

We thank Erich Buss, John Henshaw, and Jacob Slonim at the Centre for Advanced Studies, IBM Canada Ltd., for their support for this work. We also thank the anonymous referees of this paper for their valuable suggestions.

References

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

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

[3] 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.

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

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

[6] 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.

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

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

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

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

[11] 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.

[12] C.V. Ramamoorthy, Y. Usuda, A. Prakash, and W.T. Tsai. The Evolution Support Environment System. *IEEE Transactions on Software Engineering*, 16(11):1225–1234, November 1990.

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

[14] H.A. Muller, M.A Orgun, S.R. Tilley, and J.S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[15] 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.

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

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

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

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

[30] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Vol:IV*, chapter 5. Prentice-Hall, Englewood Cliffs, N.J., 1978.

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

[32] K. Bruce and P. Wegner. An Algebraic Model of Subtype and Inheritance. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, chapter 5. ACM Press, 1990.

[33] 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.

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

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

[36] Reasoning Systems, Palo Alto, CA. *REFINE User's Guide*, 1989.

Operator	Signature	Description
union,difference intersection	RELATION $\times$ RELATION $\longrightarrow$ RELATION	Standard set operators
select	RELATION $\longrightarrow$ RELATION	Returns a subset of the tuples based on a boolean condition
project	RELATION $\longrightarrow$ RELATION	Returns a relation with only the specified attributes
cartesian product	RELATION $\times$ RELATION $\longrightarrow$ RELATION	Combines the tuples in two relations exhaustively
join (natural)	RELATION $\times$ RELATION $\longrightarrow$ RELATION	Cartesian product followed by select

Table 1: Relational Algebra Operators

Operator	Semantics
$+,-,*,/$	arithmetic operators
and,**or**,**not**	boolean operators
$=,<,>,\leq,\geq$	relational operators

Table 2: SCA Operators for Atomic Data Types

Operator	Description
$< attribute >$	*signature*: COMP $\longrightarrow$ ANY *syntax*: $< attribute > (< object >)$
closure	*signature*: COMP $\longrightarrow$ SET(COMP) *syntax*: $\mathbf{closure}_{<attributelist>}(< object >)$
identical	*signature*: COMP1 $\times$ COMP1 $\longrightarrow$ BOOL *syntax*: **identical** $(< object1 >, < object2 >)$

Table 3: SCA Operators for Individual Objects

Operators	Description
select	*signature*: COLLECTION(ANY1) $\longrightarrow$ COLLECTION(ANY1) *syntax*: **select**$_{<boolean\ expression>}(< objectcollection >)$
pick	*signature*: COLLECTION(ANY1) $\longrightarrow$ ANY1 *syntax*: **pick**$(< objectcollection >)$
project	*signature*: COLLECTION(COMP1) $\longrightarrow$ COLLECTION(COMP2) *syntax*: **project**$_{<old\ attribute\ list, new\ attribute\ list>}(< objectcollection >)$
extend	*signature*: COLLECTION(COMP1) $\longrightarrow$ COLLECTION(COMP2) *syntax*: **extend**$_{<attribute:=algebraic\ expression>}(< objectcollection >)$
product	*signature*: COLLECTION(COMP1) $\times$ COLLECTION(COMP2) $\longrightarrow$ SET(COMP3) *syntax*: **product**$(< objectcollection1 >, < objectcollection2 >)$
forall	*signature*: COLLECTION(ANY) $\longrightarrow$ BOOL *syntax*: **forall**$_{<boolean\ expression>}(< objectcollection >)$
exists	*signature*: COLLECTION(ANY) $\longrightarrow$ BOOL *syntax*: **exists**$_{<boolean\ expression>}(< objectcollection >)$
member_of	*signature*: ANY1 $\times$ COLLECTION(ANY1) $\longrightarrow$ BOOL *syntax*: **member_of**$(< object >, < objectcollection >)$
apply	*signature*: COLLECTION(ANY1) $\longrightarrow$ COLLECTION(ANY2) *syntax*: **apply**$_{<operator>}(< objectcollection >)$
reduce	*signature*: COLLECTION(ANY) $\longrightarrow$ ANY *syntax*: **reduce**$_{<operator>}(< objectcollection >)$
retrieve	*signature*: COLLECTION(COMP) $\longrightarrow$ COLLECTION(ANY) *syntax*: **retrieve**$_{<attribute>}(< objectcollection >)$
size_of	*signature*: COLLECTION(ANY) $\longrightarrow$ INT *syntax*: **size_of**$(< objectcollection >)$
flatten	*signature*: COLLECTION(COLLECTION(COMP1)) $\longrightarrow$ COLLECTION(COMP1) *syntax*: **flatten**$(< objectcollection >)$

Table 4: SCA Operators for Collections

Operator	Description
union	*signature*: SET(ANY1) $\times$ SET(ANY1) $\longrightarrow$ SET(ANY1) *syntax*: **union**($< set >, < set >$)
intersection	*signature*: SET(ANY1) $\times$ SET(ANY1) $\longrightarrow$ SET(ANY1) *syntax*: **intersection**($< set >, < set >$)
difference	*signature*: SET(ANY1) $\times$ SET(ANY1) $\longrightarrow$ SET(ANY1) *syntax*: **difference**($< set >, < set >$)
subset_of	*signature*: SET(ANY1) $\times$ SET(ANY1) $\longrightarrow$ BOOL *syntax*: **subset_of**($< set >, < set >$)
set_to_seq	*signature*: SET(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: **set_to_seq**($< set >$)

Table 5: SCA Operators specific to Sets

Operator	Description
head	*signature*: SEQ(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: $\mathbf{head}_{<n>}(<objectseq>)$
tail	*signature*: SEQ(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: $\mathbf{tail}_{<n>}(<objectseq>)$
concat	*signature*: SEQ(ANY1) $\times$ SEQ(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: $\mathbf{concat}(<objectseq>, <objectseq>)$
order	*signature*: SEQ(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: $\mathbf{order}_{<attribute>,<ord>}(<objectseq>)$
seq_extract	*signature*: SEQ(ANY1) $\longrightarrow$ SEQ(ANY1) *syntax*: $\mathbf{seq\_extract}_{<pattern>:<boolean\ expression>}(<objectseq>)$
seq_element	*signature*: SEQ(ANY1) $\longrightarrow$ ANY1 *syntax*: $\mathbf{seq\_element}_{<index>}(<objectseq>)$
subseq_of	*signature*: SEQ(ANY1) $\times$ SEQ(ANY1) $\longrightarrow$ BOOL *syntax*: $\mathbf{subseq\_of}(<objectseq>, <objectseq>)$
seq_to_set	*signature*: SEQ(ANY1) $\longrightarrow$ SET(ANY1) *syntax*: $\mathbf{seq\_to\_set}(<objectseq>)$

Table 6: SCA Operators specific to Sequences

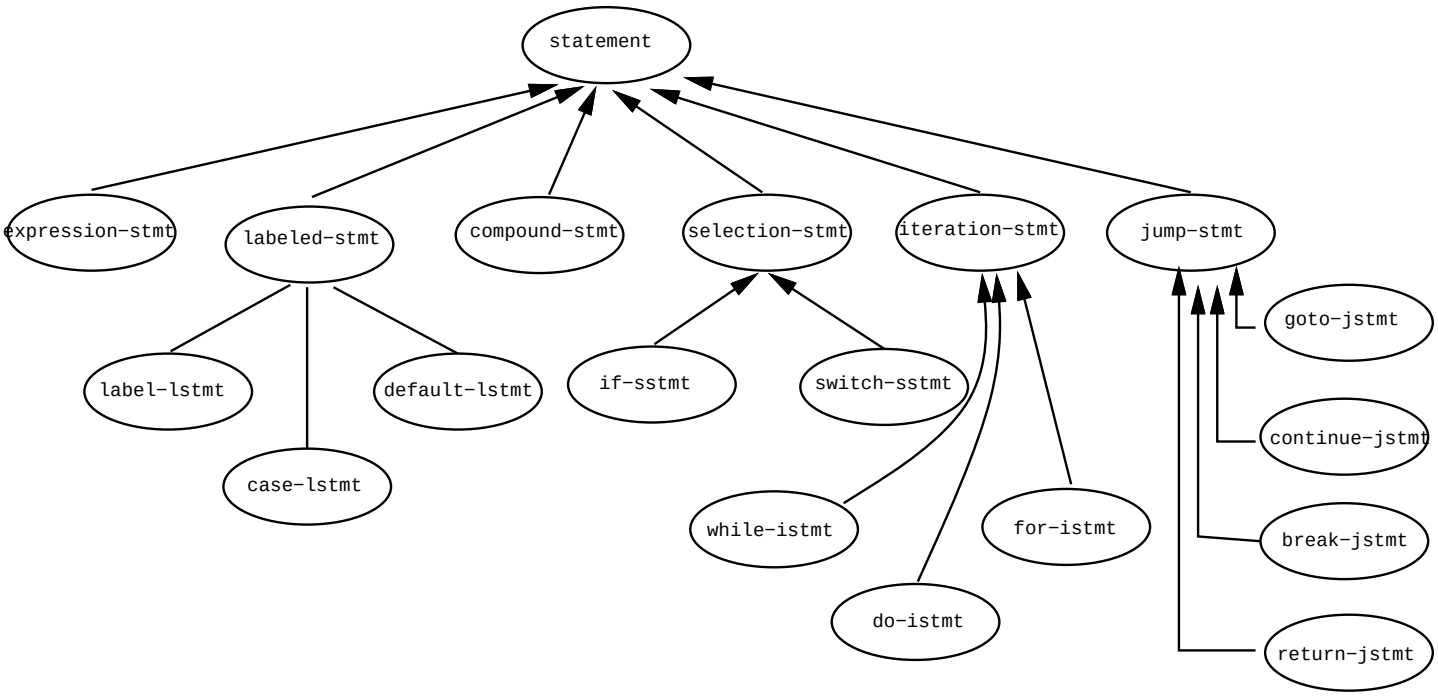

Figure 1: Type Hierarchy for C Statements

```
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
                        ......
                        funcdef:FUNCTION
                        arguments:EXPR-LIST
endtype

type   FUNCTION         subtype of PROGRAM-OBJECT
                        ......
                        type-spec:TYPENAME
                        name:STRING
                        parameters:PARAM-LIST
                        body:COMPOUND-STMT
endtype

type   FILE             subtype of PROGRAM-OBJECT
                        ......
                        name:STRING
                        funcs:SET(FUNCTION)
                        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
                        live:SET(VARIABLE) method live-compute
endtype
.....
```

Figure 2: A part of the SCA Domain Model

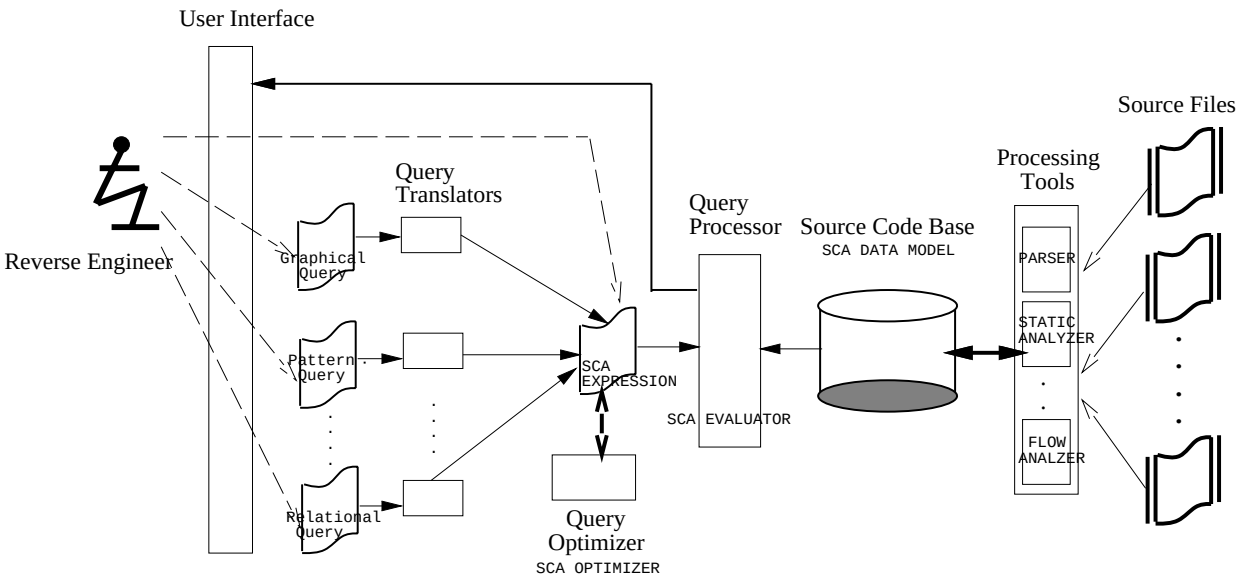

Figure 3: SCA-based Source Code Query System

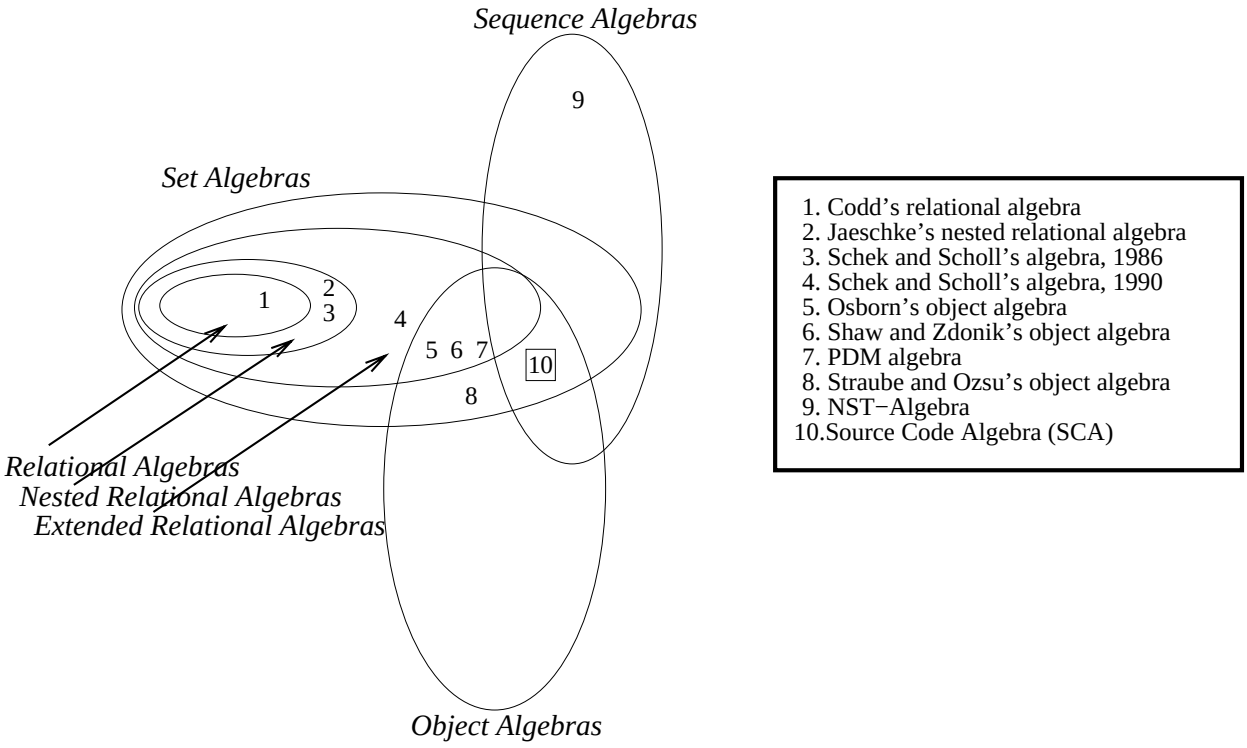

Figure 4: SCA in comparison with other Query Algebras

FOOTNOTES

- No. 1: Page 4. REFINE is a trademark of Reasoning Systems.
- No. 2: Page 9. NST stands for *Nested Sequence of Tuples*.

MATHEMATICAL SYMBOLS

σ (Sigma) (often appears with subscripts: page 10,11)
$\cup$ (Union)
π (Pi)
$\times$ (Times)
$\bowtie$ (Join)
$\in$ (in)
$\equiv$ (Equivalent)
$\longrightarrow$ (maps to)

AUTHOR BIOGRAPHIES

Santanu Paul

Santanu Paul received his B.Tech degree in Computer Science from the Indian Institute of Technology, Madras, in 1990 and an M.S. in Computer Science and Engineering from the University of Michigan in 1992.

At present, he is a Ph.D. candidate at the University of Michigan, Ann Arbor. His interests include databases, reverse engineering, and multimedia systems. Santanu Paul received an IBM Canada Graduate Research Fellowship during 1991-93 and a Rackham Predoctoral Fellowship for 1994-95. He is a student member of the IEEE Computer Society. He can be reached at the Software Systems Research Laboratory, Dept. of EECS, University of Michigan, Ann Arbor, MI-48109, and through email at santanu@eecs.umich.edu.

Atul Prakash

Atul Prakash received his B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, New Delhi in 1982, and M.S. and Ph.D. degrees in Computer Science from the University of California at Berkeley in 1984 and 1989 respectively.

Since 1989, he has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, where currently he is an Assistant Professor. His research interests include toolkits and architectures for supporting computer-supported cooperative work, support for reengineering of software, and parallel simulation. He is a member of the ACM and the IEEE Computer Society. He can be reached at the Software Systems Research Laboratory, Dept. of EECS, University of Michigan, Ann Arbor, MI-48109, and through email at aprakash@eecs.umich.edu.