Object Data Models to Support Source Code Queries:

Implementing SCA within REFINE

Santanu Paul Atul Prakash

Dept. of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI-48105

Abstract

The REFINE [1] object base is being used widely for code analysis and reverse engineering. From the perspective of program querying and interactive program analysis however, REFINE-like object bases offer only general-purpose programming languages in which users must code their program queries. In contrast, **Source Code Algebra** *(SCA) is an object algebra designed to serve as an applicative source code query language. We are currently implementing an SCA-based query processor within the REFINE environment. This paper provides insights into some object data model features which are currently absent in the REFINE framework, and argues that their incorporation will enable certain source code queries to be handled more efficiently. We also argue that the inclusion of these features will greatly simplify the implementation of the SCA query processor.*

1 Motivation

Good systems to support querying on source code are an important part of a toolkit that aids program comprehension and reverse engineering. The purpose of a source code querying tool is to help human reverse engineers indulge in *plausible reasoning* or *domain bridging* [2] — an iterative process of guesswork and verification that leads them to a better understanding of what the source code is doing.

The choice of a suitable data model for programs is a central question in the design of source code querying systems. The most rudimentary "querying" systems include string-searching tools such as `grep`, which treat program source code as a stream of characters (represented in files). Files are also the chosen representation of cross-referencing tools which keep track of identifiers and their corresponding references. Progress in database theory has had significant impact on the evolution of program data models. The relational data model has been used to model program information in the CIA [5] and CIA++ [8] systems. Subsequently, the graph data model has been adopted in SCAN [1], Rigi [10], and other graph parsing-based systems [15] as the "natural" model for representing software structure and resource flow information. The syntax tree-based model (a restricted version of the graph data model) has been used in systems such as SCRUPLE [12]. At the present time, the object-oriented data model, a logical extension of the graph data model reinforced with strong type semantics, has emerged as the more appropriate data model for program information. In particular, the REFINE object database built into the Software Refinery [14] commercial product has been successfully employed in the analysis and reverse engineering of legacy software systems [3, 9].

However, this increase in modeling power is somewhat compromised by the absence of formal as well as flexible query mechanisms in the sophisticated models. For example, in `grep`, a query is specified declaratively using a simple regular expression. In a relational framework, a query is either a relational calculus (or SQL) expression (declarative) or a relational algebra expression (applicative). Either way, a query is essentially non-procedural, i.e., it doesn't have to be programmed explicitly by the user in some elaborate programming language. In contrast, in the graph-based and object-oriented models, there are no simple, flexible query languages. This can be ascribed to the absence of sufficient formalism in these models. Consequently, systems based on these models provide either a pre-programmed menu of query features, or a

[1] Software Refinery, REFINE, DIALECT, REFINE/C, REFINE/COBOL, and INTERVISTA are trademarks of Reasoning Systems

0-8186-5647-6/94 $04.00

general-purpose programming language in which users must explicitly program their queries.

To alleviate the problem of formalism, we have already proposed a *Source Code Algebra* (SCA) as the foundation for building program query systems [13]. The algebra defines an object-oriented model for representing program or 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* [6] as the foundation for relational database systems. As in relational algebra, queries are expressed by writing expressions using the given operators. 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 powerful applicative query language, and opportunities for query optimization.

This paper deals with an experimental effort to extend REFINE with a SCA-based non-procedural query language and processor. Specifically, we are interested in querying and analyzing C programs. In our view, this is an exercise with interesting possibilities. On one hand, REFINE/C, a commercial product based on REFINE tailored to handle C programs, comes with a predefined C data model. REFINE also comes equipped with a powerful, multi-flavored database programming language that can be used for diverse purposes ranging from code transformation to metrics analyses. On the other hand, SCA offers the possibility of an applicative, optimizable query language which REFINE currently lacks. A combination of the two approaches can be achieved by implementing an SCA query processor within the REFINE framework. To achieve this, two steps are necessary. First, the SCA data model must be mapped into a corresponding REFINE data model. Second, an SCA algebraic evaluator must be implemented within the REFINE environment to serve as a query processor. We have already implemented substantial parts of the SCA-based query processor within REFINE. The prototype is operational and many complex program queries can be processed with it.

Some important issues related to object-oriented modeling arose in the context of mapping the SCA data model into REFINE. In particular, we found that the REFINE data model lacks *first-class citizen* support for collections such as sets and sequences. It also does not support an inheritance hierarchy of attributes. Furthermore, we also found that unlike some other object-oriented databases, REFINE does not contain automatic support for extending class schemas at runtime. Admittedly, these shortcomings do not cripple REFINE since any missing functionality can be programmed in REFINE (as we have done in the abovementioned cases) using its powerful database programming language. However, as we hope to show in this paper, the lack of these features complicates the implementation of the SCA query processor and makes it harder to handle certain classes of program queries within the REFINE framework.

Section 2 provides a brief description of the SCA machinery required to understand this paper. Section 3 sketches the current features of REFINE and indicates the target environment we are trying to build. Section 5 elaborates on the features that are currently unavailable in REFINE, shows why they are important, and discusses what we have done to implement them. Finally, Section 6 provides a summary and conclusions.

2 SCA: Source Code Algebra

SCA models C source code as an *algebra*. Informally, algebras are mathematical structures that consist of data types (*sorts*) and operations defined on the data types (*operators*). Our objective in designing a Source Code Algebra (SCA) was to model the data types in the source code domain as sorts, and to design source code query primitives as operators. 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 can employ a non-procedural query language to query it. There is one major distinction though: SCA is a *generalized order sorted* algebra while relational algebra is a *one sorted* algebra. In other words, SCA handles a wide variety of data types and supports a *type hierarchy*. In contrast, relational algebra supports only one type, namely *relations*.

2.1 SCA Data Model

2.1.1 Many Data Types

The data types that arise in C source code modeling can be classified into three groups:

- **Atomic data types**: These are the basic data types such as INTEGER, FLOAT, BOOLEAN, CHAR, STRING, etc.
- **Composite data types (Objects)**: Examples of composite data types in C are syntactic ele-

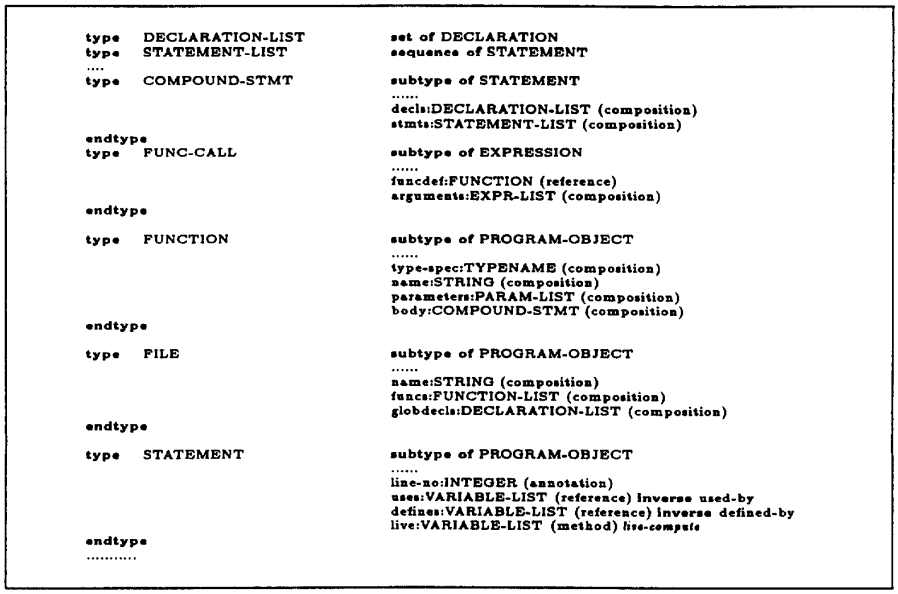

Figure 1: A part of the SCA Domain Model

ments of the language such as **while statement**, **relational expression**, **identifier**, etc.

- **Collection data types**: These are collections of other data types. For example, the type **statement-list** represents a *sequence* of **statement** objects. Similarly, the type **declaration-list** represents a *set* of **declaration** objects.

2.1.2 Type Hierarchy

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*).

SCA incorporates the source code type hierarchy as an integral part of the algebraic framework. The algebra handles the notion of subtyping and inheritance, and supports *substitutability*, an important feature which lets an instance of a subtype be used in place of a supertype.

2.1.3 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*.

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.

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 and can be used to introduce complex and specialized algorithms into the data model.

2.2 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 an algebra of atomic types, objects, and collections. We have used and extended operators from pre-existing object algebras for set operations, generalizing them

Operators	Description
< attribute >	signature: COMP ⟶ ANY syntax: < attribute > (< object >) semantics: Returns the value of the specified attribute
select	signature: COLLECTION(ANY1) ⟶ COLLECTION(ANY1) syntax: select<boolean expression>(< objectcollection >) semantics: Chooses a subcollection based on a condition
project	signature: COLLECTION(COMP1) ⟶ COLLECTION(COMP2) syntax: project<attributelist>(< objectcollection >) semantics: Retains only the specified attributes
extend	signature: COLLECTION(COMP1) ⟶ COLLECTION(COMP2) syntax: extend<attribute:=algebraic expression>(< objectcollection >) semantics: Adds a new attribute to each object
retrieve	signature: COLLECTION(COMP) ⟶ COLLECTION(ANY) syntax: retrieve<attribute>(< objectcollection >) semantics: Retrieves a specified attribute
closure	signature: COMP ⟶ SET(COMP) syntax: closure<attributelist>(< object >) semantics: Finds all objects reachable using listed attributes
apply	signature: COLLECTION(ANY1) ⟶ COLLECTION(ANY2) syntax: apply<operator>(< objectcollection >) semantics: Applies a unary operator to each element
product	signature: COLLECTION(COMP1) x COLLECTION(COMP2) ⟶ SET(COMP3) syntax: product(< objectcollection1 >, < objectcollection2 >) semantics: Cartesian product of two collections
flatten	signature: COLLECTION(COLLECTION(COMP1)) ⟶ COLLECTION(COMP1) syntax: flatten(< objectcollection >) semantics: Removes a level of nesting
pick	signature: COLLECTION(ANY1) ⟶ ANY1 syntax: pick<boolean expression>(< objectcollection >) semantics: Picks an element out of a singleton collection
size_of	signature: COLLECTION(ANY) ⟶ INT syntax: size_of(< objectcollection >) semantics: Returns the size
reduce	signature: COLLECTION(ANY) ⟶ ANY syntax: reduce<operator>(< objectcollection >) semantics: Applies a binary operator recursively to the collection
forall	signature: COLLECTION(ANY) ⟶ BOOL syntax: forall<boolean expression>(< objectcollection >) semantics: True if all objects satisfy the condition
exists	signature: COLLECTION(ANY) ⟶ BOOL syntax: exists<boolean expression>(< objectcollection >) semantics: True if even one object satisfies the condition
member_of	signature: COLLECTION(ANY1) x ANY1 ⟶ BOOL syntax: member_of(< collection1 >, element) semantics: True if element is a member

Table 1: SCA Operators for Objects and Collections

to operate on sequences wherever possible, and proposed appropriate operators for sequences. 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. A detailed description of SCA operators can be found in [13].

Table 1 shows SCA operators defined on objects and object collections. Operators specific to sets and sequences are shown in Tables 2 and 3 respectively.

Using these operators, users can express a wide variety of queries. For example, consider the SCA expression:

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

This expression evaluates to 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 file objects is then converted into a sequence and arranged in decreasing order of no_of_func. The head of this sequence is the file with maximum functions.

Operator	Description
union	signature: SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) syntax: union(< set >, < set >) semantics: Union of two sets
intersection	signature: SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) syntax: intersection(< set >, < set >) semantics: Intersection of two sets
difference	signature: SET(ANY1) x SET(ANY1) ⟶ SET(ANY1) syntax: difference(< set >, < set >) semantics: Difference of two sets
subset_of	signature: SET(ANY1) x SET(ANY1) ⟶ BOOL syntax: subset_of(< set >, < set >) semantics: True if one set is a subset of another
set_to_seq	signature: SET(ANY1) ⟶ SEQ(ANY1) syntax: set_to_seq(< set >) semantics: Produces a random sequence from the set

Table 2: SCA Operators specific to Sets

Operator	Description
head	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* head$_{<n>}$(< *objectseq* >) *semantics:* Returns a sequence consisting of the first *n* elements.
tail	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* tail$_{<n>}$(< *objectseq* >) *semantics:* Returns a sequence consisting of the last *n* elements.
concat	*signature:* SEQ(ANY1) x SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* concat(< *objectseq* >, < *objectseq* >) *semantics:* Returns a concatenation of two sequences.
order	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* order$_{<attribute>,<ord>}$(< *objectseq* >) *semantics:* Returns a sequence ordered by the attribute values
seq_extract	*signature:* SEQ(ANY1) ⟶ SEQ(ANY1) *syntax:* seq_extract$_{<pattern>:<condition>}$ (< *objectseq* >) *semantics:* Extracts a subsequence that fits the pattern
seq_element	*signature:* SEQ(ANY1) ⟶ ANY1 *syntax:* seq_element$_{<index>}$ (< *objectseq* >) *semantics:* Returns the indexed element of the sequence.
subseq_of	*signature:* SEQ(ANY1) x SEQ(ANY1) ⟶ BOOL *syntax:* subseq_of(< *objectseq* >, < *objectseq* >) *semantics:* True if it is a subsequence
seq_to_set	*signature:* SEQ(ANY1) ⟶ SET(ANY1) *syntax:* seq_to_set(< *objectseq* >) *semantics:* Returns a set consisting of the sequence elements

Table 3: SCA Operators specific to Sequences

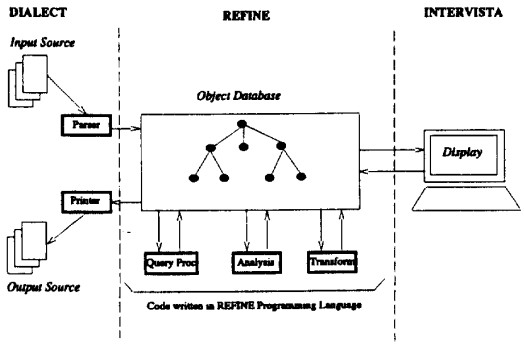

Figure 2: Software Refinery Architecture

3 Software Refinery

The Software Refinery toolkit [14] is a powerful reengineering system. It has gained considerable popularity amongst reengineering practitioners and academics as a convenient vehicle for both experimentation and industrial strength reverse engineering and reengineering. The toolkit contains an object database (called REFINE), a database programming language (also called REFINE), a parser-cum-printer generator (DIALECT), and a GUI development environment (INTERVISTA). The schematic architecture (adapted from [3]) is shown in Figure 2. We cover the object database and the programming language in this section.

3.1 REFINE Object Base

The REFINE object base is an object-oriented database that models the objects in an application domain and the relationships between the objects. Users can define their own classes and attributes and create schemas (also known as a domain model specification).

Typically, a program is represented in REFINE as an abstract syntax tree whose nodes are objects. The relationships between these objects are modeled as attributes.

The object base also supports a class hierarchy. For example, the most general class may be `program-object`. Other classes such as `program`, `function`, etc. are subclasses of `program-object`. Similarly, `while-statement` is a subclass of `statement`. A class hierarchy is semi-lattice structure that supports the inheritance relationship between the different classes.

3.2 REFINE database programming language

A powerful database programming language, also known as REFINE, is at the heart of Software Refinery. The language is multi-flavored, i.e., it supports a wide variety of programming styles and features including procedural, functional, rule-based, and pattern-based constructs. The syntax of REFINE is lisp-like. The procedural and functional constructs are useful in writing analysis routines. The rule-based and pattern-based constructs make it easy to write transformation routines for code reengineering and restructuring. Some of the other highlights of the REFINE language are:

- Support for high-level data types such as sets, sequences, trees and tuples, and their operations.
- Support for syntactic pattern matching.
- Flexible escape to Lisp, so users may write code in Lisp as and when necessary.

4 Implementing the SCA-based Query Processor

Our goal is to implement an SCA-based query processor within the REFINE environment. Figure 2 shows that currently, all processing tasks such as query processing, code analysis, and transformation must be programmed in REFINE using the database language. Our objective is to create the scenario shown in Figure 3, so that query processing and some analyses on code can be performed by an applicative query language. To reiterate, such an approach facilitates

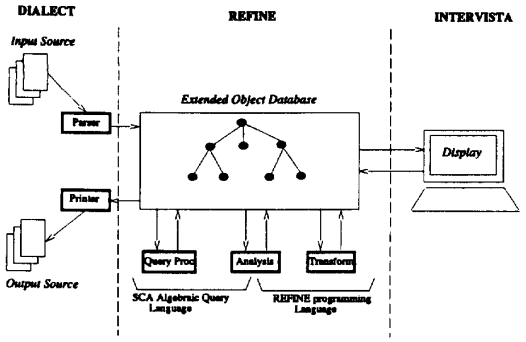

Figure 3: Modified Architecture

interactive program querying and promotes data independence, well-defined query semantics, and query optimization.

The implementation of our query processor contains two parts. First, a parser for SCA expressions must be constructed. The SCA parser converts SCA expressions written by a user into equivalent SCA expression trees. Finally, an evaluator of SCA expression trees must be implemented using the REFINE database programming language.

At this point, the SCA parser is complete and evaluator is nearing completion. The evaluator has already been used to process many SCA queries. Details of the queries that have been processed is beyond the scope of this paper.

5 Some Issues in Object-oriented Modeling

We present three important object-oriented modeling features that are unavailable in REFINE. We contend that REFINE stands to benefit from the inclusion of these features, both in terms of its modeling as well as query capabilities. We also contend that such features significantly simplify the implementation of the SCA query processor.

5.1 Collections as Classes

The REFINE data model does not support collections as *first class citizens*. REFINE classes are definable on objects, but not on collections of objects. For example, REFINE/C provides a class DECLARATION (actually DECLARATION-OBJECT but we use DECLARATION as a shorthand), the extent of which is the set of all declaration objects present in a given program. However, REFINE/C does not provide a class DECLARATION-LIST, the extent of which would be the set of all declaration *lists* in the program. On the other hand, the SCA data model treats collection data types at par with objects.

In the case of source code, collections are extremely important. Source code contains syntactic entities such as statement lists and declaration lists with the semantics of sequences and sets respectively. Queries on these collections must be supported. For example, consider the simple queries:

1. *Find all the declarations in the program.*
2. *Find all the declaration lists in the program.*

While these queries may appear contrived in isolation, such simple queries frequently occur in the context of more complex queries. For example, if the user wished to find all declaration lists with more than n declaration instances, or all declaration lists that contain m or more declarations of type T, then query 2 would need to be evaluated as the first step. Such queries apply equally well to parameter lists, statement lists, etc.

Declaration lists can occur as attribute values of objects such as FILE objects, COMPOUND STATEMENT objects, etc. In the SCA framework, since both `declaration` as well as `declaration-list` are supported as first class data types, the equivalent (simple) expressions for these queries are:

1. $select_{TRUE}(DECLARATION)$
2. $select_{TRUE}(DECLARATION - LIST)$

The REFINE implementation for the first query is simple: `instances-of`(DECLARATION). For the second query however, mapping the SCA collection data types to REFINE classes becomes the first issue. The query implementation follows from the mapping.

There are two ways to deal with the mapping problem. REFINE permits collection-valued attributes. For example, the type of an attribute may be SET(DECLARATION). One way is to create dummy classes for collections. For example, we can define a new class in REFINE/C called DECLARATION-LIST with exactly one attribute, say *list-attr*, which points to a declaration list. In this way, all declaration lists in the system can be associated with a dummy object of the class DECLARATION-LIST (through one level of indirection imposed by the attribute). To handle queries such as query 2, a query processor needs to identify collections as special classes, and transparently access the value of *list-attr* for each member of the dummy class DECLARATION-LIST.

The second way to deal with the mapping problem is to retain the attribute types as SEQ(STATEMENT), SET(DECLARATION), etc. For example, FILE has an attribute **globdecls** with type SET(DECLARATION) and COMPOUND-STATEMENT has an attribute **decls** with type SET(DECLARATION). Similarly, there may be many other attributes in the complete schema definition with the same type. To find all instances of **declaration-list** using REFINE, one has to write procedural code to navigate through objects, locate attributes with type SET(DECLARATION), and find their values.

The same query would have been much simpler if REFINE treated collections as *first class citizens*. In other words, collections ought to be treated as classes by themselves, at par with classes for single objects (STATEMENT, DECLARATION, etc.). The new classes would be STATEMENTLIST, DECLARATIONLIST, and so on. Finding all instances of **declaration-list** would then be equivalent to **instances-of**(DECLARATIONLIST), i.e., finding the members of the class, which is a simple class operation. The SCA expression would be:

$$select_{TRUE}(DECLARATIONLIST)$$

The result of the query would be a set of declaration lists.

5.2 Attribute Hierarchy or Grouping

The REFINE data model, with one exception, does not support attribute groups. On the other hand, the SCA data model supports four different kinds of attributes. These are *composition*, *reference*, *annotations*, and *methods*. This grouping of attributes can have important pay-offs in terms of query processing. Consider the query: *Find all function calls within the body of function* **foo**. Clearly, starting from object **foo**, the idea is to recursively explore its syntax subtree and filter out objects belonging to class FUNC-CALL. This can be efficiently done by pursuing only the composition attributes, as opposed to all possible attributes. Consider a different query: *Find all statements with which statement s shares a data-flow relationship*. Here, starting from *s*, we wish to traverse only the data-flow attributes (a subgroup of reference attributes) to locate target statements. Clearly, the ability to traverse attributes selectively is an important one and can be achieved using attribute groups. For example, the SCA expression for finding the function calls within **foo** is:

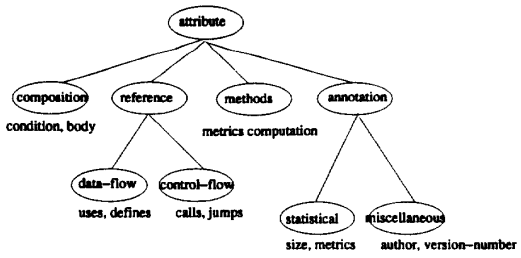

Figure 4: Attribute Grouping

$$select_{objectclass=FUNC-CALL}(closure_{composition}(pick_{name="foo"}(FUNCTION)))$$

Currently, REFINE supports composition attributes as a group. This is done using the notion of a *tree attribute*, effectively allowing the schema designer to register certain attributes as syntax tree attributes. A bunch of tree traversal operations are provided to navigate the syntax trees. There is no equivalent support for reference attributes. Nor can the schema designer decide a hierarchy of attribute groups of his or her choice. To implement the SCA attribute groups, explicit book-keeping is required to keep track of attributes.

This issue is resolved in object-oriented databases which support *attribute hierarchies*, and other object-oriented systems such as Telos [11], a development environment for information systems, which treats attributes as first class citizens. The ideal scenario is depicted in Figure 4. In addition to the four attribute groups, a schema designer may define other attribute groups as subgroups of previously-defined groups. An attribute group in the hierarchy represents a union of all attributes defined below it.

5.3 Extension of Class Schemas

REFINE does not permit the dynamic extension of class schemas. All class schemas are defined at compile time. This affects *view generation*, which is a standard feature in relational databases [7]. Views offer a mechanism to *compute* information not explicitly stored in the database. For example, the **join** operation in relational algebra computes new relations (or tables) from existing ones. In object-oriented databases, new views can be created by performing computations on existing attribute values of objects and storing the results as new attributes. Consider the query: *Show the five largest functions in terms of size (in lines)*. First, we

must compute the size of each function, then order them in decreasing order of their sizes, and finally choose the top five. To be able to order the functions, we need to explicitly *store* the size associated with each function. This should be done by extending the class schema associated with FUNCTION with a new attribute, say, *func_size*. For each function, the value of this new attribute is given by *end_line* minus *start_line*.

Currently, the only way to add a new attribute to a class schema is by explicitly compiling a new attribute. During query execution, the need to generate new attributes may arise naturally and each time, the query processor must suspend execution, re-compile the class definition with a new attribute, compute the value of the new attribute for each class member, and then resume query processing.

A simple class extension feature can solve this problem. In SCA, we have defined an **extend** operator, whose very purpose is to add new attributes to objects and compute them on-the-fly. The syntax of the operator is:

$$\textbf{extend}_{<attribute:=expression>}(< objectcollection >)$$

6 Summary

In this paper, we have discussed the idea of providing an algebraic query language interface for REFINE object bases. The idea is to provide a powerful, interactive query language for source code analysis and program comprehension, which also lends itself well to optimization. SCA is an object algebra suited to this purpose.

REFINE is a powerful and widely-used object-oriented database for programs. We have attempted to identify object-oriented modeling features that are important for querying on source code, but are currently lacking in REFINE. The three features we identify are support for collections as classes, attribute hierarchies, and support for dynamic class extension. Some general-purpose object-oriented databases such as GEMSTONE [4] currently do support some of these features, but they lack REFINE's specialized support for tasks such as parsing, program transformation, code pattern matching, etc., making them difficult to use for analyzing programs. We have shown that the inclusion of the above features is desirable in object-oriented database systems used for representing and analyzing programs in order to efficiently implement powerful languages for making queries on source code.

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.

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

[3] E. Buss and J. Henshaw. A Software Reverse Engineering Experience. In *Proc. of the CAS Conference*. IBM Canada Ltd. Laboratory, Centre for Advanced Studies, 1991.

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

[5] 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.

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

[7] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1986.

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

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

[10] 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.

[11] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[12] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, pages 463–475, June 1994.

[13] S. Paul and A. Prakash. Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*, September 1994. Special Issue on Reverse Engineering.

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

[15] L.M. Wills. *Automated Program Recognition by graph parsing*. PhD thesis, MIT, 1992.