Proceedings of the IBM Centre for Advanced Studies Conference (CASCON), pp. 381-390, Toronto, Canada, 8-11 November, 1999.

A Model Independent Source Code Repository

Anthony Cox[†], Charles Clarke[†], Susan Sim[‡]

[†]University of Waterloo, [‡]University of Toronto

Abstract

Software repositories, used to support program development and maintenance, invariably require an abstract model of the source code. This requirement restricts the repository user to the analyses and queries supported by the data model of the repository. In this work, we present a software repository system based on an existing information retrieval system for structured text. Source code is treated as text, augmented with supplementary syntactic and semantic information. Both the source text and supplementary information can then be queried to retrieve elements of the code. No transformations are necessary to satisfy the requirements of a database storage model. As a result, the system is free of many of the limitations imposed by existing systems.

1 Introduction

In order to store computer source code in a database, the source code must be abstracted in some manner so that it satisfies the requirements of the data model supported by the chosen database system. Databases applied to the management of source code are known as *software repositories* and to date, the use of entity-relationship [4], object-oriented [14], graph-based [8], and relational [16] data models has been documented. Software engineers frequently use the term *conceptual model* to refer to the instance of the data model (the data schema) that is used in the repository.

CIA [4] stores a conceptual program model based on the entity-relationship data model. Entities model constructs such as functions and variables, while relationships describe references between the entities. The main problem with this approach is the inefficiency of decomposing the source code into a set of entities and then, during retrieval, having to reconstruct it using numerous queries. This inefficiency was clearly demonstrated by the Omega system [16] where over 100 queries were needed to reconstruct a 5 line function body. CIA avoids the problem by using a coarser grained conceptual model and forfeiting much of the information extractable from the source code.

Another problem resulting from the representation of source code using a data model is the loss of information that may occur if the model is incapable of adequately representing it. Control flow information, easily represented using a graph-based model, is difficult to effectively represent using a relational model.

As an alternative, we propose the usage of a database system explicitly designed for the storage of structured text. Computer source code is written using a text editor and so it should be natural for programmers and software maintainers to query and manage it as text. To store semantic and context-based syntactic information about source code, analysis tools can be employed which generate supplementary text that can be associated with the source code and queried by the database system. It will be shown that using supplementary text permits a significant subset of the information contained in the source file to be stored producing a repository free of the constraints imparted by the data model.

The remainder of this article describes the MultiText Analytical Repository System (MARS) designed for storing large bodies of computer source code and the results of analyses on this source. Using the MultiText database system [6], developed for the management of structured text, we have developed a prototype software repository system significantly different from other implementations. To place our system in context, other repository systems are first examined, to identify their capabilities and limitations. Then, the MultiText database system and its query language, GCL, will be examined to permit their employment in a source code repository system to be understood. After that, the details of their application in MARS will be presented, followed by some concrete examples of possible queries. Finally, we discuss limitations of our initial prototype system and some possible future extensions.

2 Related Work

Several different approaches to creating repositories for computer program source code have been documented. Most of these approaches use the database to store a model of the software. In particular, conceptual source code models have been implemented with the entity-relationship, graph-based, relational, and object-oriented data models.

Perhaps the most widely known of the early repository systems is Omega [16], which stores the entire contents of a source file in a relational database and supports its querying through the use of the QUEL query language. The primary downfall of Omega is the poor performance that was exhibited by the necessity of performing many queries to reconstruct source code entities.

The Rigi system [17] uses GRAS [1], a database designed to represent graph structures, as its central repository to store parsed source code entities, such as function definitions and global variables declarations. As is the case with Omega, many database queries are needed to reconstruct a source code fragment when a fine-grained program model is stored.

CIA [4], considers only the higher level constructs of a C language program. While CIA uses an entity-relationship data model to manage global constructs (those that cross function boundaries), a relational database is employed to access the power of the QUEL query language.

Another repository system designed for high-level coarse-grained models of software is the Software Bookshelf [13]. The system uses the metaphor of a bookshelf, with each source file or high-level entity being represented by a book. Relationships between books are described with a tuple-attribute language and stored in a relational database.

REFINE [2] uses an object-oriented database to store a fine-grained syntactic representation of source code. Patterns are specified and matched against the database in order to perform program transformations for maintenance purposes. Thus, REFINE is oriented to performing transformations as opposed to providing query facilities for source code search and retrieval.

Paul and Prakash's Source Code Algebra (SCA) [18] relies on the existence of an object-oriented database in order to apply a query algebra specialized for source code. SCA is relationally complete, with extensions to support transitive closure and sequencing operations. The algebra is highly dependent upon an object-oriented data model and only considers syntactic (and not semantic) information found in source code. The actual implementation uses the database from the REFINE system as its repository.

Jarzabek, in his PQL (Program Query Language) system [14] advocates a repository system independent of its implementation. PQL relies on extended Object Modeling Technology (OMT) [20] to describe source files. These OMT models are then translated into entity-relationship data models for storage in a database.

Kamp [15] considers a repository as providing a tradeoff between the complexity of the software model and its efficiency and utility. In GUPRO, he advocates a system which is both easy to use but yet provides powerful query facilities by suggesting that models should store only enough information to answer common queries. GUPRO requires a "user designed conceptual model" described using the entity-relationship data model and stored in objected-oriented database. The query language of GUPRO is GReQL, an SQL-like language.

In CHIME [11] hyperlinks were inserted into source code to permit the use of existing WWW browsing tools. CHIME embeds information about relationships directly in source code, implicitly using a text model for the code. The hyperlinks supplement the text by adding the ability to represent graphs.

Other specialized computer languages, such as HTML, are representable using a graph-based data model. In the case of HTML, the hyperlinks are the arcs, while the WWW pages and their contents, like paragraphs and list-items, are the nodes. The ability to represent HTML with one of the data models used to represent source code suggests that techniques applied to HTML may also be applicable to source code. This is not a new observation, as the Graphlog system has been presented as a tool for both hypertext [7] and for source code [8].

Cross and Hendrix used their GRASP-ML markup language [10] to annotate source code with syntactic and control flow data. The marked up code is then used to generate control structure diagrams – a cross between pseudocode and flow charts. No effort is made to store the generated markup in a database or provide generalized query facilities.

3 The MultiText System

MultiText is a database system designed for structured text — text which has some inherent structure, but for which the structure may not necessarily be explicit. For example, the text of the document you are currently reading has structure, such as paragraphs, sentences, and phrases, however there is no explicit identification of each phrase. Source code is also structured with its declarations, statements and expressions forming the structural elements.

In MultiText, the database is viewed as a sequence of lexemes, or tokens, with each token having a unique index, represented by its offset from the beginning of the database. The definition of a token can be tailored to the specific application of the system, but for the remainder of this paper a token can be considered as roughly equivalent to an identifier in C and will be defined as any contiguous combination of alphabetical or numeric characters including the underscore character. Figure 1 contains a C code fragment with each token identified by a subscript. The value of the subscript is the index of the token.

To identify structure, or represent information extractable from the source code, *supplementary text* is used. Users can consider supplementary text as occurring between tokens in the source text and associated with either the immediately preceding or the immediately following token. We use a notation that is syntactically similar to that found in HTML by surrounding supplementary text in angle brackets. For example, in C programs, all statements could be preceded by the supplementary token <*stmt*> and followed by the token </*stmt*>. Supplementary text is indexed, and hence can be queried on, but is not returned as part of any solution.

A special type of supplementary text called a *named indirection* is also supported by the MultiText system. An indirection is simply an arbitrary link, much like a hyperlink, between two different indexes (locations) in the sequence of tokens composing the database. These links are named so as to permit them to be identified.

In repository systems, the most frequently represented relationship is between definitions and usages of source code elements, such as variables and functions. In MARS, the named indirections which store these relationships are *defined*, linking a variable or function usage to its definition, and *used*, linking a definition to one of its use sites. Indirections permit graph structures to be identified and managed by MultiText. Indirections are not limited to occurring only within a file, but can occur between files, such as is the case of the indirection *included*, linking a `#include` directive to the file included.

In figure 2, the code fragment of figure 1 has

/* **constant**$_1$ **function**$_2$ **returning**$_3$ **zero**$_4$ */

int$_5$ **z**$_6$ = **0**$_7$;

int$_8$ **zero**$_9$ () {
  return$_{10}$ (**z**$_{11}$);
}

Figure 1: Tokenized C Source Code Fragment

<*comment*>/* **constant**$_1$ **function**$_2$ **returning**$_3$ **zero**$_4$ */</*comment*>

<*varDef*><*type*>**int**$_5$</*type*> <*name*>**z**$_6$</*name*> = <*const*>**0**$_7$</*const*>;</*varDef*>

<*funDef*><*type*>**int**$_8$</*type*> <*funName*>**zero**$_9$</*funName*> <*params*>()</*params*><*body*>{
  <*stmt*><*keyW*>**return**$_{10}$</*keyW*><*expr*>(<*varRef*>
<*defined loc=5,7*>**z**$_{11}$</*varRef*>)</*expr*>;</*stmt*>
}</*body*></*funDef*>

Figure 2: Annotated C Source Code Fragment

been overlaid with supplementary text for illustrative purposes. In the actual database system, supplementary text is never stored with the source text and is managed by a separate database component. Supplemental text should be thought of as a form of invisible markup used to store information about the source text. The term "supplementary text" is used instead of the more common term "markup" in order to clearly differentiate the two. Supplemental text is a descriptive device only used in queries and, unlike markup, is never an actual physical component of a source file or a query solution.

On examination of the figure, the initial comment can be seen as occurring between the supplemental tokens <*comment*> and </*comment*>. The first of these supplemental tokens could be considered as occurring before the comment, at index $\frac{1}{2}$, and the second, after the comment, at index $4\frac{1}{2}$. In the situation where multiple supplementary tokens occur between two source tokens, all the supplementary tokens share the same index. In the figure, </*comment*>, <*varDef*> and <*type*> all occur at index $4\frac{1}{2}$.

The supplemental token <*defined loc=5,7*> is an instance of a named indirection which indicates that the <*varRef*> is defined in tokens 5 to 7 (**int z = 0**) of the database. It should be noted that the supplemental text chosen for this example is completely arbitrary and that different string such as <*keyword*> instead of <*keyW*> could be used. There is also nothing which prevents supplemental text from occurring between different source tokens.

Supplemental text is determined using other tools prior to constructing the database. Once entered into the database, only the informational content obtained during analysis is available, to the extent of causing seemingly obvious structure, such as document boundaries, to become lost unless explicitly identified with supplementary text.

There is no requirement to store only those tokens that occur in the actual documents entered into the database. It is permissible to store additional strings of *metadata* describing the contents of the database. This metadata can contain attributes of the documents, or of entities within the documents. By way of example, when storing source code, we store the filename, programming language, and a descriptor describing the analysis performed in a <*fileHeader*> area preceding the contents (<*fileBody*>) of source files. Attributes can be managed in the same manner as metadata. Attribute values are stored in an <*attributesArea*> with indirections used to link entities to attribute values.

4 GCL: The Query Language of MultiText

Interaction with MultiText is accomplished using GCL, a query language designed for searching structured text. The basic queries in GCL return either a token, or a contiguous sequence of tokens (a phrase) in the database. For example, the query `"long int"` returns all occurrences of the token `long` followed by the token `int` where any number of non-token characters or supplementary tokens may occur between the two source tokens. Given the query `"x y"` both "`x + y`" and "`x -= *y`", should they occur in the database, are returned as solutions since the `-`, `+`, `=` and `*` characters are not used to form tokens in the current system.

Supplementary text can be queried in a manner identical to source text except that supplementary tokens must be enclosed in angle brackets to differentiate them. A simple query to retrieve the variable reference to `z` from the sample code fragment would be: `"<varRef> z </varRef>"`.

Basic queries can be combined to form more complex queries using one of GCL's binary operators. Figure 3 provides a summary of these operators. The first four operators describe containment relations between the two operand queries. The *before* operator is a form of concatenation for queries and the final two are the Boolean *and* and *or* operators.

Supplementary text can be used in queries analogously to source text providing that it is surrounded by delimiting angle brackets to identify it. A simple query to retrieve all variable references in the database can be formed using two supplemental tokens and the *before* operator as follows: `"<varRef>" ... "</varRef>"`.

To permit common queries to be reused GCL also provides an abstraction facility to name and parameterize queries. The format for such a named abstraction, referred to as a *macro* in GCL, is:

$$name[(args)] = query$$

where any occurrences of *args* in *query* are replaced by the actual arguments when an instantiation occurs. The square brackets around the argument list are a notational mechanism to indicate that the argument list is optional.

To illustrate the use of GCL, figure 4 presents some simple queries which can be posed, provided that the necessary supplementary text has been generated. The use of capital letters in macro names is to improve readability and is not enforced in GCL.

GCL also provides an *indirection operator* to allow queries to follow named references (indirections) between independent sequences within the database. A typical example of a query using indirection would be:

```
defined@("<varRef>"..."</varRef>")
```

This query, which locates the definition of all variables referenced, returns the substrings associated with occurrences of the supplementary token `defined` in a variable reference. When applied to the code of figure 2 the text **int z = 0** is retrieved. Since the name of the indirection must be a supplemental token, it need not be enclosed in angle brackets.

MultiText applies the *shortest substring* rule [5] when solving queries. The rule states that no member of a solution set can contain any other member as a proper substring. This is different to other tools, such as `grep` and `lex`, where the *longest match* rule is employed to partially prevent nested solution elements. The shortest substring rule permits efficient algorithms to be employed for searching the database to find solutions.

The shortest substring rule has significant impact on the solution of queries in GCL. For the binary operators, it is not possible to simply solve each operand and then combine the two solutions so as to meet the operators definition since this may generate solutions which violate the shortest substring criteria. Given the database: `"a b a d c"`, the query `"a" ... "c"` does *not* return: `{"a b a d c", "a d c"}` which is the simple cross product of the operands since one element of the solution is properly a substring of the other element. The actual solution is the set of shortest substrings which satisfies the query, namely the singleton solution: `{"a d c"}`.

Operator	Name	Description
`>`	Contains	LHS that contain the RHS
`/>`	Does Not Contain	LHS that do not contain the RHS
`<`	Is Contained In	LHS that are contained in the RHS
`/<`	Is Not Contained In	LHS that are not contained in the RHS
`...`	Before	LHS occurring before RHS in solution
`^`	And	Both LHS and RHS occur in any order in solution
`|`	Or	One of LHS or RHS occurs in solution

Figure 3: GCL Operators

- A macro to retrieve all function calls in the database:
  `CALLS = ("<funCall>" ... "</funCall>")`
- A parameterized macro to retrieve all calls to *fun*:
  `FUNCALL(fun) = (CALLS > fun)`
- The calls to the `zero` function:
  `FUNCALL("zero")`
- All function definitions and their preceding comments:
  `("<comment>" ... "</comment>") ... ("<funDef>" ... "</funDef>")`
- All iteration statements:
  `("<stmt>" ... "</stmt>") > ("for" | ("do" ^ "while"))`

Figure 4: Sample GCL Queries

5 A MultiText Based Repository System

The information content of source code has many different aspects. One classification of this information is to use the hierarchy suggested by Perry's Interconnection Models [19]. At the top level, only the largest, most general syntactic constructs can be considered, such as functions, modules and files (the unit interconnection model). Next comes the syntactic interconnection model which adds the identification of abstract syntax (statements, expressions and local declarations) of the source file. Finally, comes the semantic interconnection model which extends the syntactic models with semantic information. However, unlike the definition provided by Perry where semantics are represented by axiomatic specifications, other aspects of source code can also be included in this model. For example type and scope information can be identified and considered as part of the semantic content of source code.

When implementing a software repository, it is usually necessary to determine which of these interconnection models one is going to support. However in MARS, the selection of the model is only dependent on the effort that is put into analyzing the source code to extract information. It is possible to store all of this information in a MultiText database and retrieve it with equal efficiency. The only limits on the semantic information stored is the ability to represent it using supplementary text. Too much information for a single source file is not a problem. If a query returns too large a solution, it is because the query was not precise enough. Increasing the database size so that users can pose more exact queries has virtually no effect on solution time and is therefore an advantage, not a weakness. The effect of database size on query solution time will be discussed in section 7.

A program analyzer for the generation of supplementary text can be as simple as a lexical scanner, or as complicated as a complete language interpreter, depending upon the difficulty of the desired analysis. Currently, we have built a parser (using YACC) for the C language and can extract both syntactic (an AST) and type information from a syntactically correct program. Supplementary text generated by our analyzer is stored in a set of files complementing the original source text. Both sets of files are then used to construct a repository. If, a higher level (coarser) program model is desired, it is possible to delete any unwanted supplementary text prior to constructing the repository, resulting in a smaller repository.

Figure 5 contains a short C program which, somewhat like the Unix `unexpand` command, converts whitespace to tabs and figure 6 overlays it with the supplemental text that is generated by our analyzer. Source code entered into MARS is typically preprocessed first, but we chose not to do so in this example for the purpose of simplification. As a result, forward definitions are not available and thus *defined* indirections can not be generated for `getchar`, `putchar` and `EOF`. Declarations in C have the potential to be very complex. To create a supplemental text schema that is consistent for all declarations, only the declaration name and its declaration specifier are identified. Other components such as type specifiers, type modifiers and storage class specifiers are ignored.

The MultiText data model treats all data in a uniform manner, viewing it as a series of tokens separated by supplementary tokens and non-word characters. It does not matter if two token sequences came from source files written in the same language — both are merely indexable sequences to be stored in the database. Thus, source files from many different languages can be entered into the same repository and queried using the same tools. Documentation such as Unix `man` pages, texinfo, Postscript and LaTeX files can also be stored in the repository. By storing the entire contents of the input files, constructs which other systems do not explicitly include in the conceptual model, such as comments, are accessible to be queried.

There is no requirement to use any specific schema, or even a single schema for the supplementary text generated for a particular source file. Multiple analyses, each generating different supplementary text, can be performed and the results merged when the database is constructed. The supplementary text generated by our analyzer and shown in figure 6 is only one of many possible schemas for the C program-

```
/* Quasi$_5$ unix$_6$ unexpand$_7$ function$_8$ */

#include$_9$ <stdio$_{10}$.h$_{11}$>

char$_{12}$ spc_to_tab$_{13}$ (char$_{14}$ c$_{15}$)
{
  return$_{16}$ ((c$_{17}$ != \x20$_{18}$) ? c$_{19}$ : '\t'$_{20}$);
}

int$_{21}$ main$_{22}$ (int$_{23}$ argc$_{24}$, char$_{25}$ *argv$_{26}$[])
{
  char$_{27}$ c$_{28}$;

  do$_{29}$ {
    c$_{30}$ = getchar$_{31}$();
    c$_{32}$ = spc_to_tab$_{33}$ (c$_{34}$);
    putchar$_{35}$ (c$_{36}$);
  } while$_{37}$ (c$_{38}$ != EOF$_{39}$);

  return$_{40}$ (0$_{41}$ );
}
```

Figure 5: A Short C Program

ming language. The flexibility imparted by the lack of requirements on the supplementary text schema allows the system to be used in a variety of ways. The only necessary requirement is that either the query tools or the repository user be aware of the actual strings which make up the supplementary text so that queries can be constructed. It is not even necessary to be aware of the entire schema, one only needs to know enough to formulate a query to solve the problem at hand.

An initial prototype of MARS has been implemented and tested on the source code for the Apache web server and the GNU C compiler. For testing purposes a simple two window system to enter queries and view their results has been provided. Although usable, we do not consider this to be more than a temporary interface, to be replaced as we explore other applications. One alternative is to link MARS to an existing system such as the Portable Bookshelf (PBS) [22] a variant of the Software Bookshelf [13]. While providing powerful browsing facilities for inspecting source code, the PBS does not have equivalent searching facilities. Extending the PBS with the ability to perform GCL queries complements its existing capabilities. The addition of MARS to the PBS is detailed more fully in [21].

6 Applications of the System

One common, though highly informal, software reuse technique consists of locating some source code which does approximately what is desired and then modifying it so that it is suitable for the new application. MARS is an ideal tool for supporting this technique since it permits users to describe source code in terms of its content. In one possible scenario, a programmer wishes to setup a Unix socket connection to another machine. Knowing that the `socket` system call is required, the programmer consults the appropriate `man` page and learns that the `gethostbyname` and `listen` system calls must be used. Unfortunately, these calls have complicated parameters. What is needed is an example code fragment which can be copied and modified for the current program. Code which locates suitable sample code fragments,

```
<file><fileHeader>
  <fileName>unexpand_1.c_2<fileName>
  <language>C_3</language>
  <schema>MARS_4</schema>
</fileHeader><fileBody>

<comment>/* Quasi_5 unix_6 unexpand_7 function_8 */</comment>

<include>#include_9 <<fileName>stdio_10.h_11</fileName>></include>

<funDef><declr>char_12 <funName>spc_to_tab_13</funName>
  <params>(char_14 <name>c_15</name>)</params></declr>
<body>{
  <stmt><keyW>return_16</keyW>
    <expr>(<cond><guard>(<varRef><defined loc=14,15>c_17</varRef> !=
      <const>\x20_18</const>)</guard>
    ? <then><varRef><defined loc=14,15>c_19</varRef></then>
    : <else><const>'\t'_20</const></else></cond>)</expr>;</stmt>
}</body></funDef>

<funDef><declr>int_21 <funName>main_22</funName>
  <params>(int_23 <name>argc_24</name>,
    char_25 *<name>argv_26</name>[])</params></declr>
<body>{
  <decl><declr>char_27 <name>c_28</name></declr>;</decl>
  <stmt><keyW>do_29</keyW> <block>{
    <expr><varRef><defined loc=27,28>c_30</varRef> =
      <funCall>getchar_31()</funCall></expr>;
    <expr><varRef><defined loc=27,28>c_32</varRef> =
      <funCall><defined loc=12,20>spc_to_tab_33
        (<varRef><defined loc=27,28>c_34</varRef>)</funCall></expr>;
    <expr><funCall>putchar_35
      (<varRef><defined loc=27,28>c_36</varRef>)</funCall></expr>;
  }</block> <keyW>while_37</keyW>
    <expr>(<varRef><defined loc=27,28>c_38</varRef> !=
      <macroCall>EOF_39</macroCall>)</expr>;</stmt>

  <stmt><keyW>return_40</keyW> <expr>(<const>0_41</const>)</expr>;</stmt>
}</body></funDef>
</fileBody><file>
```

Figure 6: Supplemental Text Generated for a C Program

which in this case is a set of function definitions, can be found in the query of figure 7.

During maintenance activities, it is frequently necessary to locate a function's definition when one encounters a function call. Programmers familiar with `grep` employ a programming style that causes function definitions to span several lines so that the function name occurs as the first element of one of the lines. Thus, for the definition of the function `foo` the regular expression `^foo` applied to all the source files will locate the desired definition. In GCL, it is possible to write a simple query which performs the same task, but which does not require special formatting and will not return any *false positive* results as did `grep`. The GCL query to locate `foo`'s definition would be:

```
  ("<funDef>" ... "</funDef>") >
(("<funName>" ... "</funName>") >
                "foo")
```

Another technique supported by MARS, and which can not be performed in other repository systems, is searching the contents of comments. Programmers often use comments to indicate sections of code that are incomplete or in need of improvement by preceding the code with a comment containing a string like `TODO`. Using `grep` only the line containing the string can be retrieved, but with the GCL query:

```
("<comment>" ... "</comment>") > "TODO"
```

the entire contents of the comment can be retrieved.

The results of one query often lead to the formation of another. Consider the case where a variable has an incorrect value. Query, `Q1`, is formed and used to identify all possible variable references in assignment or increment expressions. The maintainer may then wish to see all the definitions of the references found by `Q1` to eliminate those of variables with the same name in other scopes. This can be easily done with the query:

```
defined@Q1
```

7 Discussion

Current approaches to source code repositories have several limitations which we believe that our system avoids: poor scalability and performance, loss of context, language dependence, and schema imposed limitations. Each of these issues will now be discussed.

MultiText has been used successfully with large databases of approximately 100 gigabytes [9], as part of the annual National Institute of Standards Text Retrieval Experiments. The syntactic and type information obtained by our current analyzer approximately doubles the database size permitting up to 50 gigabytes of source code to be stored without exceeding the tested capacity of the MultiText system upon which MARS is based. On a smaller database consisting of one gigabyte of source code, complex queries can be routinely solved in less than a second using a desktop PC. This smaller database is still larger than most software systems, indicating the applicability of MARS as an efficient code exploration and maintenance tool. Furthermore, the ability to efficiently deal with large amounts of data eliminates any need to limit the amount of information obtained by analysis. The continuous operation of a MultiText news archive at the University of Waterloo for a multi-year period has lead to a stable, robust and well tested system. As a result of these properties, we believe MARS can be used for large collections of source code without encountering any scalability problems.

Containment is one of the fundamental properties of an abstract syntax tree (AST) since a node is frequently constructed such that it contains a set of subtrees. In C, an `if` node from an AST always contains a *guard expression*, a *then statement* and, optionally, an *else statement*. On examining the text of a C program, the text of these subtrees is always a substring of the text which forms the `if` statement. Thus, the source code naturally provides the containment relations which other repository systems must artificially manage. Furthermore the containment operators of GCL provide an intuitive and effective mechanism for describing containment relationships. It is the distribution of containment relationships, which occurs when source code is broken down into a set of entities, relations or graph-nodes that is responsible for the poor performance documented in repositories such as Omega [16]. MultiText, by using a fundamentally different approach to manag-

```
("<funDef>" ... "</funDef>") >
  (FUNCALL("socket") ^
    (FUNCALL("listen") ^ FUNCALL("gethostbyname")))
```

Figure 7: Query to Retrieve a Socket Example

ing source code provides powerful facilities for managing containment relationships and avoids performance problems these relationships can cause.

As discussed in an earlier section, each document can have its own data schema, since supplementary text is added by an auxiliary tool prior to the entering of the data into the database. This characteristic provides a system which allows multiple formats for the data and eliminates any system dependency on the data schema. We have shown how graph structures, relationships, entities and attributes can be described using supplementary text thereby providing equivalent representational capability to other data models.

Different definitions for identifiers in different languages can be a problem in MARS. In Scheme, `<=?--` is a valid identifier that does not get stored in the repository's index since, according the the current definition of a token, it contains only non-token characters. As it is not indexed, it can not be queried upon. Furthermore, supplementary text can not be used to indicate that an identifier occurred since there is no token for the supplementary text to surround. In Fortran, where whitespace is ignored, multiple tokens will be identified in places where only a single source language lexeme (such as an identifier) occurs. The generation of multiple tokens thus prevents a user from querying using the actual identifier since it is not an indexed term. The solution to these problems is to change the definition of a token. We do not anticipate this to be a problem, but have not yet investigated the impact of having different token definitions for different input files.

The shortest substring model employed by the MultiText system can cause problems when dealing with highly nested data. Recursive grammar constructs in a language parser are indicative of nested data. Currently, we must either generate a different supplementary token for each level of nesting or only generate supplementary text for a predetermined nesting level. In C programs, statements often contain other statements, and so the existence of

<stmt> token* </stmt>

patterns nested within each other is frequent. We have not yet attempted to implement solutions to this limitation since we are unsure of what the desirable behaviour should be. Do programmers want solutions composed of the innermost statements (shortest statements but many of them), the outermost (longer statements but few of them), or some tradeoff between solution element size and number of elements?

In other repository systems, the results of a query come back in some ordering based upon how they are physically stored in the database. While MultiText returns solutions in the order that they appear in the database, it can also order them according to their expected relevance using algorithms employed for information retrieval. The ability to rank the solutions to queries is a new area of investigation when applied to source code and so it is unknown as to its effectiveness. Does the ordering of query solutions by relevance aid software maintainers?

Chang and Eastman [3] used the SMART information retrieval system to build a code library on which queries identified potential program units suitable for reuse. Their technique depended upon the identification of "reuse-related attributes" and their storage in the system. MARS can also perform this role if the attributes are generated and stored as supplementary text. SMART also has the ability to rank query results but the different role in which it was used did not investigate ranking during maintenance and development activities.

We are also investigating the use of supplementary text for the representation of control and dataflow information. One of the possible tasks which is desirable in a software analysis environment is that of program slicing [23]. Our initial results at the use of supplementary text, including indirections, for the storage of a control flow graph have been encouraging, but to formulate a query which recreates a program slice, a program dependence graph [12] is necessary. Though we are confident that supplementary text can be used to "overlay" a program dependence graph on the source code, the efficiency and effectiveness of this representation is an open problem.

Cross and Hendrix [10] have suggested that markup is suitable for the representation of *program plan* [24] information. Should further research indicate that this is possible, MARS would provide a flexible system for exploring the retrieval of specific program plans using the GCL query language.

8 Conclusion

By supplementing the textual content of source code with additional structural and semantic information, it is possible to enter source code into the MultiText database system, creating a versatile, easily queried software repository. We have found the mechanism of supplementary text to be suitable for annotating constructs found in source code in order to identify its syntax and semantics. This technique creates a seamless system capable of managing heterogeneous collections of files, each of which may have been analyzed differently. Code written in a variety of languages can be combined in the same repository and intermixed with any desired documentation providing software maintainers with access to additional resources. The storage of the complete source file negates the need to abstract it to satisfy some specific data model, therefore making its entire content available for querying and retrieval. By applying techniques for the management of large collections of structured text to the problem of creating a software repository, MARS provides a new perspective on the management of computer source code.

Acknowledgments

Support for this work has been provided by NSERC, CITO, CSER and IBM CAS.

About the Authors

Anthony Cox is a PhD student in the Department of Computer Science at the University of Waterloo. His research interests include programming language implementation, software tools and development environments. His email address is amcox@plg.uwaterloo.ca.

Charles Clarke is an Assistant Professor in the Department of Computer Science at the University of Waterloo. His research interests include information retrieval, programming language implementation, software tools and distributed applications. His email address is claclarke@plg.uwaterloo.ca.

Susan Sim is a PhD student in the Department of Computer Science at the University of Toronto working in software engineering with a focus on the maintenance of large legacy systems. Her email address is simsuz@cs.toronto.edu.

References

[1] T. Brandes and K. Lewerentz. GRAS: A non-standard database system within a software development environment. In *Workshop on Software Engineering Environments for Programming-in-the-large*, pages 113–121. GTE Laboratories Inc., June 1985.

[2] Scott Burson, Gordon Kotik, and Law rence Markosian. A program transformation approach to automating software re-engineering. In *14th Annual International Computer Software and Applications Conference*, pages 314–322. IEEE, October 1990.

[3] Yuk Fung Chang and Caroline Eastman. An information retrieval system for reusable software. *Information Processing and Management*, 29(5):601–614, 1993.

[4] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C information

abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[5] Charles Clarke and Gordon Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19:414–426, 1997.

[6] Charles Clarke, Gordon Cormack, and Forbes Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–65, 1995.

[7] Mariano Consens and Alberto Mendelzon. Expressing structural hypertext queries in graphlog. In *Second ACM Conference on Hypertext*, pages 269–292. ACM, November 1989.

[8] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *International Conference on Software Engineering*, pages 138–156, May 1992.

[9] Gordon Cormack, Christopher Palmer, Michael Van Biesbrouck, and Charles Clarke. Deriving very short queries for high precision and recall. In *Seventh Text REtrieval Conference (TREC-7)*, Gaithersburg, Maryland, November 1998.

[10] James H. Cross and T. Dean Hendrix. Using generalized markup and SGML for reverse engineering graphical representations of software. In *Second Working Conference on Reverse Engineering*, pages 2–6. IEEE, July 1995.

[11] Premkumar Devanbu, Yih-Farn Chen, E. Gansner, Hausi Muller, and J. Martin. CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *21st International Conference on Software Engineering*, May 1999.

[12] Jeanne Ferrante, Karl Ottenstein, and Joe Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[13] P. Finnigan, Ric Holt, I. Kalas, S. Kerr, K. Kontogiannis, Hausi Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[14] Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, March 1998.

[15] Manfred Kamp. Managing a multi-file, multi-language software repository for program comprehension tools: A generic approach. In *Sixth International Workshop on Program Comprehension*, pages 64–71. IEEE, June 1998.

[16] Mark Linton. Implementing relational views of programs. In *SIGSOFT Symposium on Practical Software Development Environments*, pages 132–140. ACM, April 1984.

[17] Hausi Muller and Karl Klashinsky. Rigi: A system for programming-in-the-large. In *10th International Conference on Software Engineering*, pages 80–86, April 1988.

[18] Santanu Paul and Atul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, March 1996.

[19] Dewayne Perry. Software interconnection models. In *9th International Conference on Software Engineering*, pages 61–69, March 1987.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[21] Susan Sim, Charles Clarke, Ric Holt, and Anthony Cox. Browsing and searching software architectures. In *International Conference on Software Maintenance*, Oxford, England, September 1999.

[22] Vassilios Tzerpos, Ric Holt, and Gary Farmaner. Web-based presentation of hierarchic software architecture. In *19th In-*

ternational Conference on Software Engineering. ACM, 1997.

[23] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[24] Linda Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45:113–168, September 1990.